Porting Linux to the ρ-VEX reconfigurable VLIW softcore

Joost J. Hoozemans

Abstract

This thesis describes the design and implementation of an FPGA-based hardware platform based on the ρ-VEX VLIW softcore and the adaption of a Linux 2.0 no_mmu kernel to run on that platform.

The ρ-VEX is a runtime reconfigurable VLIW softcore processor. It supports various configurations that allow programs to run faster or more efficient. The ρ-VEX core can switch between different configurations while it is running. Reconfigurations are typically performed by a software program that is running on a different processor.

We discuss the concept of using an Operating System, running on the core itself, that monitors the execution of its tasks and orchestrates core reconfigurations during task switches. In addition to using statically found optimal configurations, performance counters could be added to the core that measure how efficient a program is running on the current core configuration. The OS could use that data to evaluate if another configuration would be beneficial. The implementation of the hardware platform and the porting of the Linux kernel represent the first steps in working towards that final goal.

To support our Linux port, a vectored trap controller has been designed. Additionally, a debugging environment has been created by designing a hardware debug unit, implementing an RSP server program and adding ρ-VEX support to the GNU debugger (GDB).
Porting Linux to the ρ-VEX reconfigurable VLIW softcore

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Joost J. Hoozemans
born in Delft, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Porting Linux to the $\rho$-VEX reconfigurable VLIW softcore

by Joost J. Hoozemans

Abstract

This thesis describes the design and implementation of an FPGA-based hardware platform based on the $\rho$-VEX VLIW softcore and the adaptation of a Linux 2.0 no_mmu kernel to run on that platform.

The $\rho$-VEX is a runtime reconfigurable VLIW softcore processor. It supports various configurations that allow programs to run faster or more efficient. The $\rho$-VEX core can switch between different configurations while it is running. Reconfigurations are typically performed by a software program that is running on a different processor.

We discuss the concept of using an Operating System, running on the core itself, that monitors the execution of its tasks and orchestrates core reconfigurations during task switches. In addition to using statically found optimal configurations, performance counters could be added to the core that measure how efficient a program is running on the current core configuration. The OS could use that data to evaluate if another configuration would be beneficial. The implementation of the hardware platform and the porting of the Linux kernel represent the first steps in working towards that final goal.

To support our Linux port, a vectored trap controller has been designed. Additionally, a debugging environment has been created by designing a hardware debug unit, implementing an RSP server program and adding $\rho$-VEX support to the GNU debugger (GDB).

Laboratory : Computer Engineering
Codenumber : CE-MS-2014-01

Committee Members :

Advisor: Stephan Wong, CE, TU Delft
Chairperson: Stephan Wong, CE, TU Delft
Member: Koen Bertels, CE, TU Delft
Member: Arjan van Genderen, CE, TU Delft
Member: Luigi Carro, Instituto de Informática, UFRGS

Member: Johan Pouwelse, PDS, TU Delft
Dedicated to my dear parents.
# Linux

## 7.1 Introduction

- 7.1.1 Operating System components
- 7.1.2 GNU/Linux distributions

## 7.2 uCLinux

- 7.2.1 uCLinux Kernel versions
- 7.2.2 Standard C library
- 7.2.3 Filesystem
- 7.2.4 User space programs
- 7.2.5 Limitations

## 7.3 The build system

## 7.4 Porting Linux

- 7.4.1 Steps
- 7.4.2 The boot process
- 7.4.3 File listings
- 7.4.4 Code examples

## 7.5 Conclusion

### 8 Evaluation

## 8.1 Hardware platform evaluation

- 8.1.1 Core
- 8.1.2 AMBA bus and main memory
- 8.1.3 Caches
- 8.1.4 Debugger Unit

## 8.2 Linux evaluation

- 8.2.1 Level of functionality
- 8.2.2 Kernel image
- 8.2.3 Task switches
- 8.2.4 Interrupt latency

## 8.3 Development Environment

## 8.4 Conclusion

### 9 Conclusions

## 9.1 Summary

## 9.2 Main contributions

## 9.3 Future work

### Bibliography
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The performance - flexibility tradeoff</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Schematic representation of an FPGA architecture</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Design overview of the $\rho$-VEX VLIW softcore</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>rvex_system module containing the core and supporting hardware.</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>The ML605 development board</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Schematic representation of the ML605 board running our GRLIB-based platform</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Simplified design schematic of the debug unit</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Connecting the HOST machine with the debug unit via a USB JTAG cable</td>
<td>42</td>
</tr>
<tr>
<td>5.3</td>
<td>JTAG TAP controller state diagram</td>
<td>43</td>
</tr>
<tr>
<td>5.4</td>
<td>Xilinx Impact showing the Boundary-scan chain of the ML605</td>
<td>44</td>
</tr>
<tr>
<td>7.1</td>
<td>Architecture-specific files in the arch/VEX directory</td>
<td>66</td>
</tr>
<tr>
<td>7.2</td>
<td>Architecture-specific files in the include/asm-VEX directory</td>
<td>66</td>
</tr>
<tr>
<td>7.3</td>
<td>Assembly code to configure the $\rho$-VEX’s IPL</td>
<td>67</td>
</tr>
<tr>
<td>7.4</td>
<td>Entry code</td>
<td>67</td>
</tr>
<tr>
<td>7.5</td>
<td>Macros used to define different types of trap vectors</td>
<td>68</td>
</tr>
<tr>
<td>8.1</td>
<td>The AMBA bus transfer delay for memory accesses is 10 cycles</td>
<td>70</td>
</tr>
<tr>
<td>8.2</td>
<td>Loading a binary image to the platform using GRMON</td>
<td>71</td>
</tr>
<tr>
<td>8.3</td>
<td>Linux boot messages</td>
<td>72</td>
</tr>
<tr>
<td>8.4</td>
<td>Components of the binary Kernel image</td>
<td>73</td>
</tr>
<tr>
<td>8.5</td>
<td>Task switch</td>
<td>75</td>
</tr>
<tr>
<td>8.6</td>
<td>Waveform of the activation of a software trap</td>
<td>76</td>
</tr>
<tr>
<td>8.7</td>
<td>Schematic representation of the full system</td>
<td>76</td>
</tr>
</tbody>
</table>
List of Tables

1.1 VEX Design parameters ................................................. 3
3.1 Overview of the Requirements and the necessary modifications for the ERA platform ...................................... 18
3.2 Overview of the Requirements and the necessary modifications for the GRLIB platform ..................................... 19
3.3 The hardware platform’s Memory Map ................................. 20
3.4 Cache characteristics ...................................................... 23
3.5 Control registers .......................................................... 24
4.1 VEX Control Registers .................................................. 33
4.2 The VCR0 register ........................................................ 33
4.3 The VCR1 register ........................................................ 34
5.1 Debug control registers ................................................... 38
5.2 The Debug Control register ............................................. 38
5.3 GDB register number with corresponding actual CPU registers .................................................. 39
5.4 JTAG signals ............................................................... 42
8.1 Execution time of Powerstone benchmark programs measured in cycles ........................................... 69
8.2 Instruction cache miss rates .............................................. 70
8.3 Data cache miss rates ...................................................... 71
Acknowledgements

I would like to thank a number of people, each having had an impact on this project in their own way. First, my thesis advisor Stephan Wong for giving me the opportunity to work on this subject, and for having great confidence in me throughout the process. My roommates at EWI, Mark and Gustavo, thanks for the gezelligheid! Wouter, thank you for proofreading my drafts and providing valuable suggestions. Many friends and fellow students, thank you for stimulating me to work hard and obtain the best possible results, for sharing many laughs and priceless moments, for tying me down to the bar when I want to leave too early, and in general just for being there.

Furthermore, I would like to thank:
My wonderful girlfriend Olga, whose beauty is not the only thing I admire about her.
My dear mother, who seems to light up every time she sees me and who has been nothing but supportive.
My dear father, who inspires me to always keep trying to develop myself as a person, and who has given me limitless patience and support during my studies.

Joost J. Hoozemans
Delft, The Netherlands
August 17, 2013
Introduction

This thesis describes a Msc project that aims to add Operating System support to the \( \rho \)-VEX reconfigurable VLIW softcore processor. In this chapter, we introduce the most important concepts and provide the necessary context for understanding the project. After this, the problem statement is defined and the project goals will be introduced. This thesis will contain many topics that will be extremely familiar to embedded Linux developers. A great effort has been made to make it more comprehensible for a broader audience of more general (Computer) Engineers, while trying not to over-elaborate.

1.1 Context

The computer industry is always driving to design new systems that increase performance and decrease power consumption (maximizing power and chip area efficiency). Technology scaling is an important factor in creating faster and more energy-efficient chips [1]. This is out of the scope of this project. Another source of increasing performance and efficiency is innovations in microarchitecture [2] [3]. An introduction to the traditional approach for improving processor performance is given in Section 1.1.1. This thesis uses a \( \rho \)-VEX processor, that is based on an innovative flexible microarchitecture that aims to deliver high performance and to be power and area efficient.

In recent years, a number of developments have had a profound impact on the field of Computer Engineering, including:

- The transition from multi-core to many-core architectures
- The advent of 3D-stacked chip technology
- Reconfigurable Computing

Each of these new technologies provide new opportunities for designers to create faster and more efficient systems. This thesis falls into the third category: Reconfigurable Computing (RC). Reconfigurable computing is introduced in Section 1.1.3.

1.1.1 Classical processor design

Classic processor design practice revolved around improving performance by increasing clock frequencies and improvements in processor architectures. These architectural improvements aim to exploit properties that most computer programs exhibit, one of the most important being Instruction-Level Parallelism (ILP) (see [4] for an introduction to computer architecture). ILP is a measure of how many instructions can be executed in parallel by the processor.
CHAPTER 1. INTRODUCTION

Instructions can be executed in parallel if there is no dependency between them. Programs often run sequences of calculations where subsequent operations need the result of previous operations. In that case, the ILP of the program is low. In other cases, many calculations can be performed independently. Modern processors are able to execute multiple instruction in a single clock cycle, this is called “multiple issue”. These processors have multiple functional units that can perform an operation at the same time, thereby allowing parallel execution of instructions. There are 2 global approaches to multiple issue processor designs:

- Superscalar processors
- Very Long Instruction Word (VLIW) processors

This thesis discusses a VLIW processor design. The architecture is introduced in Section 1.1.2 and our implementation of this architecture is presented in Section 1.1.4.

Superscalar processors use complex circuitry to analyze the instruction dependencies to decide which ones can be executed simultaneously. Most modern desktop processors use this design approach. VLIW processors leave the dependency analysis to the compiler. The processor will have a certain “issue-width” that determines how many instructions it can execute per clock cycle. The compiler explicitly places instructions that can be executed simultaneously in a bundle (the very long instruction). This way, a VLIW processor does not need the complex circuitry thereby saving area and power (and this advantage grows as the issue-width increases). The compiler can use more advanced techniques to find parallelism in the program (in part because it can do much higher-level code transformations such as loop unrolling), potentially exploiting more ILP compared to the hardware circuitry of a superscalar processor.

Traditional drawbacks that have prevented VLIWs from becoming mainstream are binary code compatibility and low efficiency of the instruction encoding for programs with low ILP. A concept that addresses these problems is discussed in Section 2.2.2.

The next section introduces a VLIW architecture that aims to expand the advantages of a VLIW processor by introducing architectural flexibility.

1.1.2 VLIW Example - VEX

In [5], a VLIW architecture called VEX is introduced. It consists of an Instruction Set Architecture (ISA), a C compiler and a compiled simulation system. It was created mainly for academic purposes - it is a showcase for “architecture exploration”.

The architecture has been designed to be scalable and flexible. It has a number of “parameters” that can be varied: for instance the number and type of functional units, instruction latencies and number of registers (see Table 1.1). VEX can therefore be seen

---

1 For a VLIW, an instruction always contains as many operations as the issue-width. These instructions are called “bundles”. The individual operations (that roughly correspond with a classic CPU instruction) are called “syllables”.

2 Programs need to be recompiled if there are major changes in the architecture - as opposed to superscalar processors that can still run binaries that were compiled for much older designs.

3 If there are less instructions available than the processor’s issue width, the compiler must still fill the bundle. It will therefore issue an instruction that does nothing: NOP.
as a “family” of processors; every parameter set represents a different instance. The
compiler supports these parameters as well; it can be configured to generate code for
different parameters. Using the simulator, these parameter configurations can be evalu-
ated. This way, by testing a program using different architectural parameters, an optimal
configuration can be selected easily without having to modify and recompile the compiler
or simulator. Note that the behavior of a program can vary greatly between different
phases of its execution. A program can therefore have different optimal configuration
parameters for every phase. This will be discussed in Section 2.2.4.

The VEX ISA supports the parameters given in Table 1.1. The Computer Engineer-
ing group of Delft University of Technology has implemented the VEX architecture using
reconfigurable hardware. The next section introduces reconfigurable computing and the
reconfigurable VEX implementation is presented after that.

### Table 1.1: VEX Design parameters (from [6])

<table>
<thead>
<tr>
<th>Processor resource</th>
<th>Design parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional units</td>
<td>Number of Functional units, type supported instructions, degree of pipelining</td>
</tr>
<tr>
<td>Register file</td>
<td>Register size, register file size, number of read ports, number of write ports</td>
</tr>
<tr>
<td>Load/Store unit</td>
<td>Number of memory ports, memory latency, cache size, line unit</td>
</tr>
<tr>
<td>Interconnection network</td>
<td>Number and width of buses, forwarding connections between units</td>
</tr>
</tbody>
</table>

1.1.3 Reconfigurable Computing

This section briefly introduces the concept of Reconfigurable Computing. For more
information, refer to a textbook on the subject, such as [7]. Reconfigurable Computing
essentially aims to combine flexibility with performance. Traditionally when choosing
an implementation for a certain application, the choice was to either create a software
program for some type of general-purpose processor, or to design a hardware circuit that
could be made into an Application-Specific Integrated Circuit (ASIC). The first solution
is flexible; the processor used for the implementation can execute any program, and the
program can easily be modified. The second solution offers much higher performance
and lower power consumption, but an ASIC can never be changed as it has been designed
to perform a single function and a chip, once manufactured, can never be changed.

By using flexible computing fabric technologies such as Field Programmable Gate
Arrays (FPGAs), it becomes possible to change the behavior of a hardware circuit. This
technology sacrifices some performance and power efficiency compared to ASICS, but
FPGA implementations are still orders of magnitude faster than software implementa-
tions (running on general-purpose processors) for certain types of applications. This is because of the different nature of FPGA circuits; a complex operation that could take hundreds of CPU cycles to complete could be designed into a hardware circuit that performs the operation in a single cycle. FPGA circuits can be pipelined so that even complex multi-cycle circuits can still produce a result every cycle.

![Diagram of performance-flexibility tradeoff for different types of implementations.](image)

Figure 1.1: The performance - flexibility tradeoff for different types of implementations (from [8]).

An FPGA is an integrated circuit that can be configured after it has been manufactured. The hardware itself cannot be changed, but an FPGA contains large amounts of programmable logic blocks and interconnections. Every logic block contains a small memory where a truth table can be stored; a lookup table (LUT). By writing values into the truth table, these blocks can be programmed to perform any logical function. By programming the blocks and connecting them using the interconnections, any hardware circuit design can be programmed into the FPGA.

Reconfiguring an FPGA can be done very quickly (in the order of 100 ms, depending on the size and model). FPGA configurations can be created using a high-level Hardware Description Language (HDL) like VHDL. Using these tools, a hardware designer can have his design implemented in hardware in a matter of minutes. This is why FPGAs have been used extensively for prototyping purposes for some time.

More recently, system designers have been using FPGAs to speed up applications. By designing special circuits for parts of the program (that are very slow in software) and running those them on an FPGA, some applications can achieve speedups in the order of 100s of times faster [9]. This is why FPGAs are currently being employed by many high performance computer systems like the Convey Computer and IBM’s hybrid POWER7 systems. When running parts of the program on a general-purpose processor and parts on FPGA circuits, the implementation consists of software and so-called “configware” [10] (because an application-specific circuit has been programmed into the FPGA - an FPGA configuration). It takes considerable effort to convert a normal software implementation

---

4The number of inputs and outputs for these blocks are fixed, so a LUT can perform any logical function for that number of in/outputs.
into an FPGA-accelerated implementation. Although advanced compilers are available that can compile most code directly into an optimized hardware design that can run on an FPGA, this approach still needs skilled programmers that have profound understanding of both software and hardware.

Modern FPGAs contain so many resources\(^5\) that they can instantiate multiple processor cores. In this case, these processors are called “softcores” (because their design is not implemented in an ASIC, but programmed into an FPGA). Using FPGAs, the architectural flexibility of VEX can be implemented; the softcore can be changed according to the VEX design parameters. This project uses a softcore called $\rho$-VEX, which is introduced in Section 1.1.4. The goal of this softcore is to leverage the possibilities of reconfigurable computing to increase performance and efficiency without requiring complex hardware-software co-compilation, thereby “bridging the gap between general-purpose and application-specific processing” [6, pg. 1].

\(^5\)Modern FPGAs contain not only a large number of LUTs, but also other types of resources such as memory blocks and optimized multiplication circuits.
1.1.4 \( \rho \)-VEX: The Delft reconfigurable VLIW softcore processor

In [6], a parametrizable, runtime reconfigurable VLIW softcore called \( \rho \)-VEX is introduced. It is an implementation of the VEX architecture written in VHDL, implementing the VEX architecture’s flexibility using reconfigurable computing (FPGAs). The architectural parameters as discussed in Section 1.1.2 are supported by the \( \rho \)-VEX core (hence “parametrizable”). Using partial reconfiguration [11], the parameters can even be changed while the core is running (hence “runtime/dynamically reconfigurable”).

The core (that will be discussed in more detail in Section 3.4.2) together with the toolchain and other software and hardware components together constitute the \( \rho \)-VEX research Platform (See also publications on the ERA project, such as [12]).

The \( \rho \)-VEX Platform is used for a variety of research topics in the TU Delft Computer Engineering Laboratory. However, its usage is limited by the lack of Operating System (OS) support. Therefore the core is mostly used as a co-processor [12] or in a standalone system that can only run programs bare-metal\(^7\). OS support is very desirable, because this will allow us to run real-world applications on the core and to explore more research possibilities.

An interesting direction for research on the \( \rho \)-VEX is to modify an OS so it can coordinate reconfigurations of the core. The \( \rho \)-VEX as presented in [13] is able to change between configurations during runtime but there is currently no general way to control this. That means that a control program (that must be running on another processor core in the system) must monitor and reconfigure the core ([14] shows how this is done in the \( \rho \)-VEX-based ERA system). Using an OS running on the \( \rho \)-VEX itself, this can be done without requiring a separate core. The execution of programs can be evaluated during task switches so that a reconfiguration can be scheduled for when a particular task will resume (if the evaluation determines that a different configuration would be better for that task).

The benefit of continuously modifying the configuration, is that this scheme allows high performance for tasks that can make use of a wide VLIW architecture (because of their high ILP) while reducing power and area usage when running tasks that cannot. In that case, FPGA resources can be used for other computations. For example, a program can run on a 8-issue \( \rho \)-VEX core, but if it does not use most of the available issue slots, the system can be reconfigured into 2 2-issue cores and a 4-issue core so that other tasks can run on those 2 other cores.

1.2 Motivation

From the previous discussion, we can see that operating system support would be beneficial for the \( \rho \)-VEX Platform. The reconfigurable characteristics could be applied to a fully operational environment that includes an OS. This way, all tasks running on

\( ^{6}\)The word “platform” can also be used in a hardware context, where it refers to all the hardware components of a system. Throughout this thesis, “Platform” will be capitalized only when we are referring to the research Platform.

\( ^{7}\)“Bare-metal” means directly on the core, without any operating system or control software. This means that the program must be manually adapted to run on the core as there are no system calls or other abstraction mechanisms.
the OS can benefit from the advantages offered by the reconfigurable architecture of
the $\rho$-VEX directly, instead of running a $\rho$-VEX as a co-processor with another core
running the OS. This is used in the ERA system (see [14]); it uses a MicroBlaze softcore
that runs Linux combined with supervisor software that controls one or more $\rho$-VEX
cores. In such a system, the OS must offload parts of a task to the $\rho$-VEX, which means
that these tasks need to be extracted from the original program (and both parts need
to be compiled and debugged separately)\(^8\). Additionally, the task offloading introduces
large performance penalties in transferring the instruction and data memories to the
co-processor\(^9\). These drawbacks are avoided by running the OS on the $\rho$-VEX core
directly.

The final goal of the overall project is to implement a system that constantly op-
timizes the reconfigurable hardware parameters for its running tasks.

To accomplish this, the OS needs to be able to control the core’s configuration
and it needs a means to determine the optimal configurations for its tasks\(^10\). In
Section 2.2.4, the concept of using compile-time determined optimal configurations is
discussed. In Section 2.2.5, the notion of adding performance monitoring hardware is
introduced. Using the performance counters, the OS could measure runtime character-
istics of the running task with respect to the current hardware configuration. Using
an evaluation algorithm (discussed in Section 2.2.6), it could evaluate if a different
configuration would be beneficial for a certain task.

Achieving the final goal is a very large undertaking. This thesis focuses purely on
the first step: adding Operating System support to the $\rho$-VEX Platform. As we will see
later, this is already very comprehensive process that is itself comprised of several steps.

### 1.3 Problem statement and Project Goals

The problem statement is:

How to implement Operating System support for the $\rho$-VEX Platform?

Implementing OS support for a Platform requires considerations in a number of
areas; First, there are a number of different options and variants to choose from
when selecting an OS. Each of these options will have different hardware requirements
and levels of functionality. An OS must be selected that offers the desired level of
functionality, while posing realistic demands on the hardware. This will lead to a set
of hardware requirements that must be met by the hardware platform that will be

---

\(^8\)It is also possible to run full programs on the $\rho$-VEX co-processor in such a system, but in that case
the program cannot make use of the OS’s facilities such as system calls so this still limits this scheme to
trivial programs (i.e. no I/O operations).

\(^9\)Alternatively, the co-processor could be designed to be able to access main memory directly, but this
introduces other difficulties such as coherency and synchronization.

\(^10\)While taking into account that this optimal configuration might vary between execution phases of a
program.
used. Besides the hardware, an important factor is the software toolchain. Backends for the \texttt{rvex} target have been developed for GCC and binutils so a complete toolchain is available to generate binaries for the \(\rho\)-VEX. Thus far, the toolchain has only been used to compile relatively simple programs (compared to a full OS). Therefore, it is likely that issues will be encountered during the process. It is important to find and fix problems concerning the toolchain during the debugging. To be able to use the system for future research, it is important to create a well-functioning development environment. This environment should be able to support the development and debugging of real-life applications running on the OS. To summarize, the main goals of this project are:

1. Defining and implementing the hardware requirements for the system\textsuperscript{11}
   (a) Defining hardware requirements to support the OS selected in subgoal 2a.
   (b) Choosing an existing \(\rho\)-VEX-based hardware platform, and evaluate it with respect to the requirements.
   (c) Implementing the missing hardware requirements.
   (d) Evaluating the modified hardware platform.

2. Defining and implementing the software requirements for the system
   (a) Choosing a suitable OS that meets the desired functionality, has realistic hardware requirements and is portable in the limited time frame of this project\textsuperscript{12}
   (b) Porting the selected OS to the hardware platform.
   (c) Evaluating the port.

3. Establishing an effective development environment to support the system.
   (a) Modifying the toolchain so it can be used to generate binaries for the system.
   (b) Identifying and solving issues in the \(\rho\)-VEX toolchain.
   (c) Creating a debugging environment for the system.

1.4 Overview

\textbf{Chapter 2} presents related work and provides additional background regarding the feasibility of the final goal as defined in \textbf{Section 1.2}.
\textbf{Chapter 3} formulates the hardware requirements and describes the set up and evaluation of an initial hardware platform used for the project.
\textbf{Chapter 4} presents the design of the hardware additions that are necessary according to the evaluation.

\textsuperscript{11}In this context, the term “system” denotes the complete set of hardware components, software and supporting tools.
\textsuperscript{12}As is clear from this enumeration, there is a strong connection between the hardware requirements and the selection of an OS. The selection should be made based on the current state of the \(\rho\)-VEX design and should maximize OS functionality while minimizing the necessary hardware modifications. This trade-off seems complicated, but in practice it amounts to using a very popular version of the Linux kernel that is designed to run on limited hardware. This will be discussed further in \textbf{Section 7.1.2}.
In Chapter 5, the implementation of a debugging environment is presented. Chapter 7 describes Linux in detail, starting with a short introduction and argumentation of the choices that were made, followed by an overview of the porting process. Chapter 6 presents the software tools used in the project, the modifications that were made to the toolchain and the issues found during the porting and debugging process. Evaluations of the hardware designs, some characteristics of the kernel and preliminary performance measurements are presented in Chapter 8. Finally, Chapter 9 summarizes all previous chapters, lists the main contributions of this project and gives suggestions for future research.
This chapter provides an overview of related work, and introduces some concepts necessary to understand the project. The first section presents related work on reconfigurable softcores. Subsequently, Section 2.2 discusses various concepts that are needed to create the envisioned system as defined in Section 1.2. It will be argued that the steps performed in this project contribute to that system directly.

### 2.1 Reconfigurable softcores

Well-known softcores include the MicroBlaze from Xilinx and the NIOS from Altera. These softcores are not reconfigurable. There are multiple reconfigurable softcores described in literature. The first reconfigurable VLIW softcore was the Spyder, introduced in [15]. It was designed from scratch instead of being derived from an existing architecture as is the case with the $\rho$-VEX. As such, the $\rho$-VEX benefits from having a industrial-strength compiler available from the start. Other customizable VLIW softcores have been designed [16], but they do not have the architectural freedom of changing the issue width or number of functional units or they can only be customized during design-time.

Other systems using a reconfigurable core, such as [17] and [18], use it as a co-processor. This will, to some extent, have the same drawbacks as the ERA approach as discussed in Section 1.2.

### 2.2 Combining parametrized reconfigurability with an OS

Running an OS on the $\rho$-VEX is an interesting research possibility because it allows us to evaluate the advantage of using a reconfigurable system in an OS environment. However, certain conditions must be met to ensure that the final system will still be able to leverage the $\rho$-VEX’s parametrized reconfigurability characteristics. This section will discuss these conditions.

#### 2.2.1 Optimizing configuration for OS code

The final goal as introduced in Section 1.2 is to optimize the hardware configuration for the tasks running on the OS, not for the OS itself. In other words, optimizing the configuration for OS code is not a goal. In real systems, only a minority of the system’s total execution time will be spent in kernel (OS) code. Operating system code is inherently very control-bound and will not benefit from a wide VLIW architecture (see Section 8.2.2 for an analysis of the binary of our Linux port). Control-bound code also introduces a performance penalty when compiling for dynamic reconfiguration, as will be discussed in Section 2.2.2. Furthermore, adding support for different issue-widths
to the OS code is complicated and difficult to test because it contains highly optimized assembly code and because it must handle very delicate tasks such as synchronization and interrupts. Because of these reasons, the most feasible approach would be to reset the core to a standard configuration when executing OS code. This will allow us to make the simplification of using a standard configuration throughout this project, without impeding the usability of our OS port for the envisioned system. In other words, even though it only supports a single hardware configuration, our OS port is an important first step towards implementing the envisioned system that supports optimization through dynamic hardware reconfiguration.

2.2.2 Supporting dynamic issue-width using generic binaries

Normally, when changing the issue-width of a VLIW processor, recompiling the program is necessary (as discussed in Section 1.1.1). Every bundle in the binary needs to contain one syllable per instruction lane. In our envisioned system, this would be detrimental because it means that:

- A separate binary is needed for every issue-width that is to be supported by the program.
- Every time the core is configuration is changed, the corresponding binary would need to be loaded.

The first point is regrettable as it would nullify the advantage of only needing to distribute a single binary and a set of configuration parameters, but the second point is insurmountable. It is not possible to restore the context of a program after replacing its binary by one that was compiled for a different issue-width as most branch offsets and label addresses will be different (among other problems).

This is why for ρ-VEX, the concept of generic binaries has been developed \[19\]. A generic binary can be executed by a ρ-VEX of any issue-width. This avoids the need to switch binaries when changing the core’s configuration. The generic binary works by including a stop bit in every syllable. A set stop bit characterizes the end of a bundle. The assumption on which the concept is based is that all the syllables before the stop bit can be executed in parallel, so they can also be executed one after the other. When compiling for an 8-issue configuration, this means that the 8 syllables in a bundle could also be executed in 4 sequential steps by a 2-issue core. This assumption poses some restrictions\[1\] and a performance penalty is involved, but generic binaries are a key enabler of a dynamically reconfigurable system. Our ρ-VEX binutils version already supports assembling generic binaries.

2.2.3 Controlling hardware configurations

To be able to orchestrate reconfigurations of the core during runtime, the OS must be able to control the partial reconfiguration. This can be done by using software-accessible registers that control the configuration. An example is to introduce a number of register

\[1\]For example, branch operations must reside in the last syllable of the bundle, otherwise the syllables after the branch will be skipped when running on a smaller issue-width core.
pairs, with one register that should contain a configuration and the other the address of the instruction that should trigger the reconfiguration. Using this scheme, the OS can schedule reconfiguration for its tasks beforehand, and it does not need to interrupt the task to order a reconfiguration action. Instead, the registers will be compared to the core’s Program Counter (PC) during every cycle. If the PC corresponds with one of the address registers, the hardware will reconfigure the core using the configuration store in that pair. The addresses where the core configuration should be changed will coincide with transitions between execution phases in a program. The next sections will introduce these phases and discuss different means for the OS to find the different phases of programs (and their transitions).

2.2.4 Execution phases

In most real-world programs, the behavior varies greatly during runtime \cite{20}. For example, some parts of the program are heavily memory-bound while others are computationally intensive. This takes configuration parameters to a whole new level; because our \(\rho\)-VEX implementation is dynamically reconfigurable (in other words, we can change its microarchitecture while it is running), it is technically possible to apply a new set of configuration parameters for every phase of the program\footnote{Similar ideas (adapting hardware to application behavior) have been proposed by \cite{21} \cite{22} \cite{23} among others, but not combined with a parametrized processor core.}. That means that every part of the program will run as efficiently as possible.

Phase information can be extracted during compile-time (also referred to as static or off-line analysis), but detecting in which phase a program is currently running and predicting the next phase is a whole different challenge. Static analysis can find transitions between phases but this information is only partially useful during runtime\footnote{Transitions can sometimes depend purely on the code (e.g. in case of an unconditional branch), but in many cases it will depend on the execution context. Static analysis can find the different transitions that are possible, but the actual transition will be decided at runtime.}. Another solution is to monitor the execution and extracting information on how the program is running \cite{20}. Using this approach, it might be able to optimize the hardware configuration for a program even when no phase information is available for it.

2.2.5 Runtime execution monitoring

By including performance counters at strategic points in the core’s design, we could evaluate certain execution characteristics of the core to see how a program is running on the current hardware configuration. Even when no phase information (and corresponding optimal configuration parameters) is available, it could be possible to deduce if another configuration might be faster or more power/area efficient\footnote{The most straightforward example is to count the number of NOPs. There are more possibilities, like monitoring if the multiplication units are being used at all. They can be deactivated after being idle for a long period of time and the program will generate an “unavailable hardware unit” exception when it tries to execute a multiplication. The OS can handle this exception much like a page fault; it will activate the unit again and resume the program’s execution.}.

The evaluation mechanism can be implemented by the OS scheduler. During every process switch, it can check the performance counters for exceeding certain threshold
values. It can then decide that the program will run faster or more efficient using a different configuration of the core. When the program is selected to be the next task to run on the core, the scheduler will signal the core to reconfigure itself into the new configuration when resuming execution.

### 2.2.6 Evaluation algorithms

The performance counters can only supply information on a program’s previous behavior. This information must be evaluated to do anything useful with it. To know if hardware reconfiguration is beneficial for the task, the OS needs to predict the behavior of the process when it will resume execution. For this, runtime phase detection and prediction algorithms are needed. [24] discusses a prediction scheme that can capture and predict phases during runtime with very little overhead.

Besides per-process evaluation, the OS should track the predicted behavior of all its running tasks in a holistic way to maximize efficiency; taking into account the current hardware configuration, the available area and the preferred configuration of all the tasks that can be selected by the scheduler. In [25], different task scheduling strategies are evaluated for systems with reconfigurable computing nodes.

### 2.3 Conclusion

This chapter first presents related work on softcore processors. The $\rho$-VEX is the only runtime reconfigurable VLIW core with a complete toolchain. Then, concepts related to the feasibility of the final goal as defined in Section 1.2 are discussed. Combining an OS with parametrized reconfigurability poses a number of requirements on the Platform. First, it is argued that for a system to benefit from parametrized reconfigurability, it does not need the OS code itself to be able to run on varying configurations as the final goal states that the configuration must be optimized for the tasks running on the system. We can therefore implement the operating system without having to ensure that it will run on varying hardware configurations in order to make sure that our OS will be usable for the envisioned system. In other words, even though it only supports a single hardware configuration, our OS port is an important first step towards the final goal.

Subsequently, it is argued that for a task to be able to run on varying configurations, it will need a special form of binary as it is impossible to replace a running binary by one that was compiled for a different hardware configuration. Generic binaries can be used for this purpose. The OS needs to be able to control the hardware configuration. This can be implemented by hardware registers that control the reconfiguration circuitry. The OS needs to know what configuration is optimal for its running tasks. Furthermore, every task will have different optimal configurations during different phases of its execution. Two concepts are discussed to handle this; compile-time phase analysis and using performance counters to monitor runtime execution characteristics of a task in order to evaluate the current hardware configuration. This last scheme might be able to optimize the hardware for a task even if no phase information (and corresponding optimal configurations) is available.
The hardware platform used for this project is built around the reconfigurable ρ-VEX softcore processor. In this chapter, the platform is described in more detail. First, we will list the requirements for the platform with respect to OS support and two different options for our initial setup. Subsequently, we will present the entire hardware platform used for the project and assess the changes that are necessary to meet the requirements. Lastly, we will describe our hardware development environment and provide an overview of the modifications that are performed in this project. The implementation of those additions and modifications are presented in Chapter 4 and Chapter 5. To support the hardware platform, modifications to the toolchain are also needed. These are presented in Chapter 6. The process of porting an OS to the platform is presented in Chapter 7. The hardware platform will be evaluated in Chapter 8.

3.1 Requirements

To support a general purpose Operating System, certain requirements must be met by the hardware. The OS that was selected for this project is Linux as will be discussed in Chapter 7. This section provides an overview of the requirements.

Processor

One of the requirements of the standard Linux kernel is that the processor must support paging (as is described in the README file). Our ρ-VEX core does not have a Memory Management Unit (MMU), which means that it does not support paging. However, there is a Linux variant that was modified to be able to run without paging: uCLinux. This variant and the implications on the final system will be discussed in Section 7.1.2.

Main memory

Before this project, the ρ-VEX was typically used with separate data and instruction memories implemented in BRAMs on the FPGA. The ρ-VEX has separate access ports for data and instruction memories, so this represents a pure Harvard-architecture. For the OS to be able to load and run programs and for the system to be able to load the kernel into memory from FLASH (usually done by a boot loader), a single main memory must be used. Otherwise, the core cannot write into its instruction memory making it necessary to side-loaded every binary manually. The core will still have separate access ports but they will connect to the same memory. This is called a Modified Harvard-architecture.

The first requirement for the platform is to have a single-address space main memory of sufficient size. The size of the binary image of a simple Linux system is estimated to be around 1 MiB. The absolute minimum amount of memory needed is then approximately 2 MiB. However, this would pose severe restrictions on the system (care
must be taken that the binary size stays within the limits). Particularly when using
kernel functions such as a RAMdisk\(^1\) more memory must be available. So although
not a requirement in the strict sense, a memory size of 8 or 16 MiB is very desirable.
Another requirement for the memory is that loading binaries into memory should be
easy and fast, because the kernel will have to be loaded and debugged repeatedly. We
will be loading the binaries directly into RAM for testing.

**Console**
The next requirement for running an OS is a means to communicate with the console,
allowing us to see the kernel messages and provide input. Most Linux-enabled embedded
systems use a serial port (UART) for this. Such a port is available on our development
board (see Section 3.6).

**Programmable hardware timer**
Linux is a time-sharing system, that divides CPU time in very small slices allowing
time multiplexing of multiple processes, thereby appearing to be executing them
simultaneously\(^2\). It relies on a timer to generate “ticks” that tell the kernel that a
certain amount of time has lapsed. The kernel will then check its running timers to see
if any of them has timed out (and take appropriate action if it finds one), and will give
CPU time to another process if the current process’ time slice (quantum) has run out.
The timer will provide the system with ticks by signaling interrupt requests. The hard-
ware is required to have a timer which must be programmable to fire at a certain interval.

**Interrupt controller**
The interrupt must be handled by an interrupt controller that can be programmed to
prioritize and mask interrupts\(^3\) and that can interrupt the core so it will jump to an
interrupt handler routine.

**Trap facility**
Refer to Section 4.2.1.2 for our definitions of trap/interrupt terminology. The core must
be able to be interrupted by the interrupt controller. Additionally, it needs to be able
to handle certain internal conditions (such as exceptions) gracefully by means of a trap
facility. This also entails software interrupts as they are generally used to implement
system calls.

**Debugger**
To establish an effective development environment and to facilitate the Linux porting
process, a debugger is very desirable. The \(\rho\)-VEX design does not include debugging

\(^1\)Which could save us from having to implement and/or debug FLASH storage drivers or networking
subsystems to allow us to mount a root filesystem

\(^2\)paraphrased from [26, pg. 258]

\(^3\)Handling interrupts is a fairly complicated kernel task. Priorities are needed to allow nested inter-
rupts, where a running interrupt handler can only be interrupted by something that is more important.
Masking is needed to temporarily disable interrupts in certain parts of the kernel that may not be inter-
rupted (critical sections) because they can not be synchronized using mechanisms like semaphores. See
[26] pg. 215, 218-219 for more information on Kernel synchronization (particularly regarding interrupts).
3.2. INITIAL PLATFORM OPTIONS

functionality at this time. This is a property of the core itself, not the platform, and will be discussed in [Chapter 5].

3.2 Initial platform options

There are 2 platforms readily available to select as our starting point.

- The first option is used in the Computer Engineering (CE) group as the “ERA reference platform” [12], where a $\rho$-VEX core is added to a PLB bus system that hosts a MicroBlaze running Linux (MBLinux). The core has separated instruction and data memories that are connected to the bus. This connection is slave-only: the $\rho$-VEX core itself cannot issue bus commands. Because of this, and because some functionality is already in use by the MBLinux system (e.g. timer, UART), this platform is not ideal and would require substantial modifications to meet the requirements.

- The second option is a platform based on GRLIB, and is being used by other ERA researchers. This platform uses the $\rho$-VEX core attached to separate instruction and data caches, but they are both attached to an AHB bus with a master interface[4]. The core can thereby issue bus requests, and read/write the DDR memory on the FPGA board. This means that this system can address all of the 256 MiB of memory backed by instruction and data caches. The platform also includes an APB with various peripherals from GRLIB, including a timer and interrupt controller. The Linux SPARC port includes code to use the timer and interrupt controller, which also saves development time.

Table 3.1 and Table 3.2 show an overview of the requirements and the modifications that are needed for both platforms to comply.

Usability of the ERA platform is low because the $\rho$-VEX core’s interface to the bus is slave only; it can not read or write to any resources attached to the bus. Instead, it is only possible to use the MBLinux system to write to the core’s memory and issue commands to it through control registers. Feasibility of meeting the memory requirements is medium because the amount of BRAMs on the FPGA is limited ([27]), and can not be used to create memory arrays of sufficient size. This is partly due to the number of access ports needed; the core needs separate memory ports for data and instruction accesses, of which the instruction port needs to be 128 bits wide to be able to read a full instruction bundle every cycle. Additionally, the system needs to be able to write to the core’s memory from the bus to allow the MBLinux to side-load binaries to the $\rho$-VEX.

If the bus interface would be redesigned into a bus Master, the $\rho$-VEX core could use the DDR memory attached to the bus. However, this memory would either have to be shared with the MBLinux system, or the mechanism of side-loading binaries would have

---

[4] A bus interface can be one of two types: a Master or Slave interface. Peripherals with a Slave interface can only receive commands from the bus and respond to them. To be able to issue commands, writing or reading data to and from other components on the bus, a component must have a Master interface.
Table 3.1: Overview of the Requirements and the necessary modifications for the ERA platform

<table>
<thead>
<tr>
<th>Requirement</th>
<th>ERA platform</th>
<th>Necessary modifications</th>
<th>Feasibility, required effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>64 kiB Imem, 32 kiB Dmem</td>
<td>Increase memory using BRAMs or modify the bus interface to busmaster and share DDR between MBLinux and ρ-VEX</td>
<td>Medium, large effort</td>
</tr>
<tr>
<td>UART</td>
<td>USB UART, MDM UART</td>
<td>USB UART is in use by MBLinux. Bus interface must be changed to Master to use MDM UART.</td>
<td>High, medium effort</td>
</tr>
<tr>
<td>Timer</td>
<td>Not available</td>
<td>Design a timer unit or insert an open source timer core</td>
<td>High, medium effort</td>
</tr>
<tr>
<td>Interrupt/Trap facility</td>
<td>Not available</td>
<td>Design an interrupt controller or modify existing ρ-VEX trap design</td>
<td>High, High effort</td>
</tr>
</tbody>
</table>

The UART could be used if the MBLinux system uses the MDM UART. This requires little effort, but attaching the UART peripheral to the ρ-VEX will require some effort (it would include designing an address decoder) if a Master bus interface is not available. Designing a timer unit (or including an open source design) would not require much effort.

The GRLIB-based system is much closer to meeting the requirements. It already has a programmable timer and interrupt controller. The core is connected to an AMBA bus as a Bus Master, so it can issue commands to any component attached to the bus. This includes the DDR memory, so 256 MiB of DDR memory can be used freely.

This platform is selected to be used for the project, as much less modifications are required. The following sections will describe the GRLIB hardware platform in more detail. In Section 3.5, the platform is evaluated, listing the necessary additions and modifications to meet all the requirements.
3.3 GRLIB

GRLIB is an open-source hardware library (mostly written in VHDL) that contains an integrated set of reusable IP cores [28]. It is being used for a course on processor design taught by the Delft TU Computer Engineering group. As such, we already had previous experience with the AMBA-based platform featured in GRLIB. One of the researchers in the ERA team added a cache subsystem to the ρ-VEX core that connects to the AMBA bus as a Bus Master. The core can perform read/write operations on the memory and all the peripherals attached to the bus. Therefore, there is a multitude of peripherals available that can easily be attached to the platform.

Examples of such peripherals are a UART, Timer and Interrupt Controller that are essential for a hardware platform to support an OS. Also, the AMBA system includes a component that allows a software utility (GRMON) running on the host PC to connect to the platform. GRMON can load binaries into the main memory and perform arbitrary bus commands. It can therefore be used to start/reset the core and to access all peripherals.

Another advantage of using GRLIB is that the entire system can be simulated using Modelsim. This includes the bus and peripherals. This is convenient to debug the software, but also very important for integral testing of our modifications and additions to the hardware.
3.4 The GRLIB platform

The full hardware platform is built around an AMBA bus with an AHB that connects the high-performance elements to each other and to a bridge to an APB, that connects to various peripherals. This section provides an overview of all the different components attached to the system.

3.4.1 Components

The platform consists of a GRLIB AMBA system including:

- Debug Support Unit (DSU)
- GPTIMER general purpose timer unit
- IRQMP multi-processor interrupt controller
- UART serial interface
- General purpose registers
- Main memory (DDR)
- \( \rho \)-VEX system including the core and caches

The most important components will be described in the next sections. The components from GRLIB are more appropriately documented in their respective data sheets or user manuals available from Gaisler Research \[28\].

<table>
<thead>
<tr>
<th>Start address</th>
<th>Size</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>256 MiB</td>
<td>Main Memory</td>
</tr>
<tr>
<td>0x80000000</td>
<td>1 MiB</td>
<td>APB bridge</td>
</tr>
<tr>
<td>0x80000100</td>
<td>256 B</td>
<td>UART</td>
</tr>
<tr>
<td>0x80000200</td>
<td>256 B</td>
<td>Timer</td>
</tr>
<tr>
<td>0x80000300</td>
<td>256 B</td>
<td>Interrupt Controller</td>
</tr>
<tr>
<td>0x80000500</td>
<td>256 B</td>
<td>Core and Debugger control registers</td>
</tr>
</tbody>
</table>

3.4.2 The \( \rho \)-VEX core

The \( \rho \)-VEX is a parametrizable, runtime reconfigurable softcore VLIW that implements the VEX ISA introduced by \[5\]. There is a small number of different versions that are being maintained at this time. The version we are using for this project is not runtime-reconfigurable, but only design-time parametrizable.

\footnote{Throughout this thesis, the terms AMBA-platform and GRLIB-platform are used interchangeably. The platform is built around an AMBA bus from GRLIB.}
3.4. THE GRLIB PLATFORM

For more information on the original development of the core see [29], for the non-pipelined version and the dynamically reconfigurable version see [13]. Our version uses the default configuration of 4 issue slots throughout this project. We will not change the configuration because that would complicate both software and hardware development tremendously.

Our $\rho$-VEX core in its default configuration features a 5-stage pipeline that supports forwarding. It contains 4 ALUs, 2 Multiplication units, 1 Branch unit and 1 Load/Store unit, separate ports for instruction and data memory (Harvard architecture), a general register file (GR) consisting of 64 32-bit registers with 4 write and 8 read ports, a branch register file (BR) with 8 1-bit registers. In this platform, the core is synthesized with a clock frequency of 75 MHz. It is implemented in VHDL and available under an academic license.

![Diagram of the $\rho$-VEX VLIW softcore](image)

Figure 3.1: Design overview of the $\rho$-VEX VLIW softcore using its default 4-issue configuration. The colored components are parametrizable; the number of pipelanes can be 2, 4 or 8, and every pipeline can be equipped with ALU, MUL, Branch, and MEM (Load/Store) units.

Parametrizability is implemented using VHDL generics. The configuration must be

\[\text{See Section 2.3 for an argumentation why this assumption can be made in this project.}\]
coordinated with the compiler and assembler as will be discussed in Section 6.2. The VEX compiled simulation system (see Section 1.1.2) can be used to profile an application using different configuration parameters to quickly find the best configuration parameters. The most important parameter is the issue-width; this can be set to 2, 4 or 8. It specifies the number of pipelanes in the core (see Figure 3.1).

Every pipeline can also be configured, as it can be equipped with one or more functional units. In practice, every pipeline will contain at least an ALU and the entire core will in total contain 1 branch unit and 1 MEM unit. The core can only issue 1 instruction to each pipeline every cycle, so only 1 functional unit can be active per lane at the same time. Therefore, it is best to evenly distribute the units over the lanes, so that the compiler can issue an instruction bundle containing for example a load instruction together with a branch and a multiplication. If the units were placed in the same lane these instructions would have to wait for the lane to become available.

Although it is technically possible to insert more MEM units into the core (which could greatly enhance performance for memory-bound applications), this has not been tested yet because it would also require another data memory port (which would require modifications to the core and the memory subsystem). Multiplication (MUL) units are relatively expensive in terms of FPGA resources and are only useful for multiplication-intensive applications.

The above considerations lead to the balanced default configuration. It contains 4 pipelanes that are all equipped with an ALU and either a branch unit, MEM unit or a MUL unit.

3.4.3 rvex_system

The VHDL module that contains the core is named rvex_system. A schematic representation of this module is given in Figure 3.2. It includes the caches, control registers and the AHB interfaces that connect them to the bus. A large part of the changes will be made in this module, with a debug unit being added to the control register hardware (see Chapter 5) and a Trap controller (see Section 4.2) being added as a new component.

3.4.3.1 Cache

The cache subsystem is taken from the CARPE project, available at [30]. It was integrated into the platform by one of its maintainers. It consists of separate instruction and data caches, an arbiter that will handle the memory requests of both caches, and an interface that connects to the AHB bus.

The caches are also reconfigurable. They have a number of design parameters including different replacement policies, set associativity and size. Throughout this project, the parameters described in Table 3.4 are used.

3.4.3.2 General Purpose registers

The registers in the rvex_system module are used as control registers. Bit 0 of the first register is a self-resetting mechanism to reset the core. When 1 is written to it, the
3.5 Evaluation and necessary hardware modifications

This section presents the additions and modifications to the hardware that are needed for the GRLIB-based platform to meet all the requirements as defined in Section 3.1.

- The memory requirement has already been met.
- The UART can be used immediately, and Linux even has a driver for it.
- The timer can be used immediately; it only needs to be connected to the interrupt controller. This is an easy task because these peripherals have been designed to be used together in GRLIB-based platforms.

Figure 3.2: rvex_system module containing the core and supporting hardware.

Table 3.4: Cache characteristics

<table>
<thead>
<tr>
<th>Parameter</th>
<th>I-cache</th>
<th>D-cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set associativity</td>
<td>1 (direct-mapped)</td>
<td>4</td>
</tr>
<tr>
<td>Nr. of sets</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>Cacheline width</td>
<td>128 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Replacement policy</td>
<td>Least Recently Used</td>
<td>Least Recently Used</td>
</tr>
</tbody>
</table>

hardware will assert the ρ-VEX core’s reset signal and reset the bit value back to 0 in the following cycle. Other registers are a status register and counters for different cache mechanism (this platform was originally created to evaluate the reconfigurable cache design). Another register can be used to read the current value of the Program Counter. See Table 3.5 for an overview of the different registers and their functions.
Table 3.5: Control registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80000500</td>
<td>Control register</td>
</tr>
<tr>
<td>0x80000504</td>
<td>Status register</td>
</tr>
<tr>
<td>0x80000508</td>
<td>Cache read misses</td>
</tr>
<tr>
<td>0x8000050C</td>
<td>Cache read accesses</td>
</tr>
<tr>
<td>0x80000510</td>
<td>Cache write misses</td>
</tr>
<tr>
<td>0x80000514</td>
<td>Cache write accesses</td>
</tr>
<tr>
<td>0x8000051C</td>
<td>Clock cycle count</td>
</tr>
<tr>
<td>0x80000520</td>
<td>Program Counter</td>
</tr>
</tbody>
</table>

- The GRLIB interrupt controller has been designed to connect to the Leon3 SPARC processor core. This means that a module must be added to the ρ-VEX core that can interface with this interrupt controller and handle the IRQs. Section 4.2 describes the design and implementation of this trap/interrupt facility and will elaborate more on the subject.

- An addition to the core is necessary to allow the new trap/interrupt module to be configured. This is implemented in the form of control registers, presented in Section 4.3. These registers should be accessed by the core through special instructions. These instructions form an extension to the VEX ISA and are presented in Section 4.4.

3.6 Platform overview

This section presents the hardware used and the tools needed to simulate and synthesize the platform. Furthermore, we will outline the necessary modifications presented in the previous section.

The FPGA board used in this project is the Xilinx ML605 evaluation kit ([27], see Figure 3.3) that features a Virtex6 FPGA. The ML605 board is the physical hardware. It features a Virtex-6 FPGA, that will be used to program our reconfigurable hardware designs (configware). A schematic representation is shown in Figure 3.4. The main memory is located on the board. GRLIB connects it to the AHB bus. The UART component of GRLIB uses a hardware module on the board, which is connected to our host machine using USB. There is a JTAG connection available on the board, also connected to our host machine. GRLIB implements a TAP controller to allow the host machine to communicate with the GRLIB platform (see Section 5.4).

Shown in red is the trap facility that must be added to the core according to Section 3.5. It connects to the core and to the IRQMP interrupt controller from GRLIB. The design of this trap controller is presented in Section 4.2.

An effective development environment (that must be established to meet our goals as defined in Section 1.3) should include a well-functioning debugging facility. Shown
3.6. PLATFORM OVERVIEW

Figure 3.3: The ML605 development board.

Figure 3.4: Schematic representation of the ML605 board running our GRLIB-based platform.

in yellow is the general-purpose register component (see Section 3.4.3.2) that can be expanded to include hardware debugging functionality. The design of this hardware debug functionality, together with the development of the software to support debugging, is presented in Chapter 5.

Simulation is done using Modelsim SE 6.6g ([31]). For synthesizing, a Makefile included in the project directory calls the Xilinx tools to synthesize, map, Place & Route the design (including the parts from GRLIB that must be included from their own location). If desired, it is also possible to use the ISE design suite to synthesize the project. To download the generated bit file to the board, iMPACT is called by the Makefile. The Xilinx software version used is 13.4.

A large part of the VHDL modifications and development is done using Sigasi ([32]). The tools used for software development and debugging are described in Chapter 6.
3.7 Conclusion

An initial hardware platform based on GRLIB is selected to be used in the project. It already meets most of the system requirements. This chapter started with a discussion of the requirements for the hardware platform. It introduces two existing designs that could be used as an initial platform for our system. The amount of work that is needed for both platforms to meet the requirements is assessed, and one of the platforms is selected accordingly. Subsequently, the key properties of this platform are described in more detail. The chapter finishes with a review of necessary additions and modifications to meet the requirements.
This chapter presents the development of all the hardware additions and modifications that were needed for the GRLIB-based platform (that is described in Section 3.4) to meet the requirements as evaluated in Section 3.5. In Section 4.1, high-level modifications to the platform are presented. Section 4.2 introduces the rationale and design of the trap facility. Section 4.3 and Section 4.4 present additions to the core to control the trap facility. Chapter 5 presents the design of a hardware debug unit separately. The hardware platform, including the modifications, is evaluated in Chapter 8.

4.1 Initial platform modifications

Initially, a number of components did not fully behave as desired and they needed to be modified. First, the caches’ reset signals were connected to the AMBA reset signal instead of the specific ρ-VEX reset signal (that only resets the ρ-VEX core, not all the hardware on the bus). As we wanted to be able to restart the core reliably using the core control registers (see Table 3.5), the caches needed to be flushed as well. Therefore, we changed the caches’ reset signals to connect to the ρ-VEX reset signal.

When we started to test the UART connection, it turned out that the data cache would intercept every read operation, including reads to memory addresses outside of the main memory region. This caused read operations to the peripherals connected to the APB (Advanced Peripheral Bus) to always return the values first read from the peripheral. It was therefore not possible to poll the UART status flags or to read input. Thus far, the UART has only been used by simple programs to output results.

A small modification to the data cache has been developed (by one of its maintainers that also created the initial platform), where an extra input signal called bypass was added to the data cache that would circumvent the cache when asserted high. By connecting this signal to the first bit of the address lines, the cache is circumvented whenever the address is higher than value 0x80000000. All the peripherals on the APB are mapped above this address. Main memory is mapped to region 0x00000000 - 0x20000000 as shown in Table 3.3. Therefore, the data cache only caches requests made to main memory, not requests made to peripherals.

Another addition has been made to the status registers; it was initially not possible to see the current value of the Program Counter (PC). This is very desirable during debugging. We added a new register that returns the value of the PC signal (which connects the core’s address lines to the instruction cache).
4.2 Trap controller

Support for Interrupts and Exceptions is essential for being able to run a general purpose OS like Linux. A simple trap controller needed to be designed that could be connected to the IRQMP interrupt controller from GRLIB.

4.2.1 Introduction

This section introduces some of the concepts needed to understand the design. An Interrupt ReQuest (IRQ) is a signal to the CPU that an event has occurred that needs immediate attention [33]. Essentially, the event wants to Interrupt the processor at what it was doing to handle the event. This could mean that a timer has run out, or a peripheral device finished an operation. These events are usually time-critical: For example when a network adapter is receiving data, it will signal an IRQ when its buffer is full. If the processor does not read the device’s buffer (i.e. store it in main memory), consequent transmissions might be lost. The OS must install handler routines that acknowledge the request, save the execution context, take necessary actions and return to the program that was interrupted. Interrupt handling is an essential and sensitive task of the kernel [26, pg. 132].

An Exception is an unexpected situation encountered by the processor during execution. Examples are an arithmetic overflow, a page fault, or division by zero. Exceptions must also be handled by the OS, which will try to resolve the issue (e.g. by loading the missing page into memory to handle a page fault) and resume execution or terminate the program (e.g. when the program is not allowed to access the requested page).

The area of Interrupts, Exceptions, Traps and Faults has been subject to varying (and sometimes unclear) terminology. There are different definitions of these terms. The next two sections will introduce the most common terminology and the one that will be used in this project.

4.2.1.1 Intel Terminology

To create a general understanding of different types and variations of Interrupts and Exceptions, a commonly used classification is presented here. It stems from the Intel documentation. Note that this is not the terminology used in this thesis; it serves only to clarify the different types of abnormalities that a CPU can encounter during execution [33].

- **Interrupts**
  - **Maskable Interrupts**
    An IRQ from peripherals results in a Maskable interrupt. Maskable means that the interrupt can be masked by the processor; it will be ignored until it is unmasked.
  - **Non-Maskable Interrupts (NMIs)**
    Some events, such as hardware failure, may never be ignored by the processor. They will issue a non-maskable interrupt.

---

1 This table is paraphrased from [26]
• Exceptions
  
  – *Programmed Exceptions*
  The programmer can cause an exception using a special CPU instruction. They are effectively equivalent to Traps, and often called *Software Interrupts.*
  
  – *Processor-detected Exceptions*
    
    * Faults
      The CPU encountered a condition that can generally be corrected by the OS. The OS can then decide to resume execution at the instruction that caused the fault.
    
    * Traps
      In case of a trap, the execution should be resumed at the instruction after the instruction that caused the trap. Mostly used to transfer control to a debugger, to signal that the program has encountered a condition in the program where the debugger should be notified (e.g. a breakpoint).
    
    * Aborts
      A severe internal error has occurred, and the CPU must terminate the program.

This terminology is used commonly (with variations, for instance [33] classifies Exceptions as software interrupts).

4.2.1.2 SPARC Terminology

In this thesis we will use the terminology as defined in the SPARC V8 architecture manual ([34], this section mostly paraphrases the Trap chapter). Our platform is based on GRLIB that contains the LEON3 softcore which is an implementation of the SPARC V8 architecture. The peripherals in GRLIB are designed to work with the LEON3. That means that the IRQMP interrupt controller also complies with the SPARC specifications, and our trap/interrupt facility should be designed to interface with it.

The main concept in the SPARC terminology concerning exceptions and interrupts is the *Trap*, which is a “vectored transfer of control to a trap handler through a Trap Table” (cited from [34, Chapter 7])². A Trap can be caused by an external event (*Interrupt*), by an event in the processor caused by an instruction (*Exception*)³ by a hardware failure or an other error (e.g. bus error or hardware malfunction).

4.2.2 The Trap Table

The trap table contains an entry for every type of Trap. Every Trap type has a number (essentially equal to an index), that will indicate which entry (or “Vector”) in the table the processor will jump to (hence the “vectored transfer of control”). Half of the trap table is reserved for hardware traps (of which 16 entries are used for Interrupt Requests),

²Note that, although not strictly required to handle traps/interrupts, this defines the Trap Table a key element of the Trap handling system.

³This includes a special Trap instruction that can be used to signal 128 different types of software traps.
and the other half is available for software Traps. Every Vector is a single bundle that should contain a jump to the handler associated with the trap type and possibly other operations such as moving the Trap number into a register and/or storing a register on the stack. See Figure 7.5 for different types of trap vectors in our Linux port (discussed in Chapter 7).

The size of the table is 4 kiB; it contains 256 entries of 16 bytes (assuming 1 bundle contains 4 instructions; if the issue width changes, the trap table will change accordingly). Normally, the table is followed by the programs entry point so the first program instructions will start at address 0x1000. As such, the first vector (also called the “reset vector”) at address 0x0 will be an unconditional branch to address 0x1000.

Theoretically, the trap table can be placed anywhere in memory. SPARC defines a special-purpose register that must be loaded with the base address of the table. Our Trap facility does not implement such a register and instead assumes that the table is placed at address 0. This address could be changed at design time and must be coordinated with the linker script.

4.2.3 Interrupts

The hardware trap vectors in the trap table include 16 vectors for handling IRQs. Every one of these vectors corresponds to an IRQ, every IRQ has its own priority. To implement this, the IRQ line from the interrupt controller to the processor actually consists of 4 lines encoding a number from 0 to 15. These IRQ lines are used to calculate the branch target address (they represent the index into the IRQ section of the trap table).

The Processor Interrupt Level (PIL), represented by the 4 least significant bits of the VEX Control Register, is the priority level of the interrupt currently being handled by the processor. Only interrupts with a higher priority than the current PIL can interrupt the processor. When an IRQ is signaled, its priority is compared to the current PIL and the transfer of control (jump) will take place only if the value is higher.

The programmer can also manually set the PIL by writing the VCR (that will be introduced in Section 4.3), thereby masking lower-priority interrupts. Setting the PIL to the highest level (15) will mask all IRQs except for the NMI. This is used in the Kernel to temporarily disable interrupts (see Figure 7.3).

4.2.4 The existing trap design

In [13, Section 3.3], an interrupt controller specifically designed for the ρ-VEX is already presented. However, it had the following drawbacks:

- The version of the ρ-VEX core for which it was designed is much older than the one that is being used in this project. Adapting it to our version would require considerable effort

- This controller was not designed to be used with a general-purpose OS. It was designed for the highest possible interrupt performance (minimizing interrupt latency) instead of flexibility and programmability. It performs many tasks in

4\footnote{Note that on most systems, this level will generally be 0 as the processor will be executing normal code most of the time.}
hardware and is even capable of switching between multiple register windows or hardware-generating the context saving routine. The interrupted programs resume address is stored in a hardware register that is not software accessible, which prevents the OS from performing a process switch triggered by an interrupt. A preemptive system (like Linux) is therefore not possible using this interrupt controller.

- We wanted to use the interrupt controller from GRLIB because the Linux kernel already contains code for that controller. This would save development time, and also make the design of the Trap facility simpler as prioritizing and masking interrupts is already taken care of.

For these reasons, we decided to design a new simple trap controller that can interface with the IRQMP unit from GRLIB.

4.2.5 Requirements

The design of this unit is relatively simple because a preemption signal is already present in the core. This signal is used for running programs on the core as a co-processor. It causes the core to jump to a specified address. The task of our trap design is to drive those signals based on:

- The input from the interrupt controller.
- Instructions that should cause a software trap, including the special Trap instruction.
- Hardware events that should cause a hardware trap (like division by zero or misaligned memory accesses)
- The state of the core with respect to traps and interrupts; it must be possible to disable traps entirely, and to temporarily mask interrupts.

This last point can be implemented in different ways. Most GPPs have one or more special purpose register(s) that control the core. One of the functions of these registers (often called “Processor State Register” or similar) is to enable or disable traps. This functionality must also be implemented for \( \rho \)-VEX. These registers should be accessed using special instructions\(^6\) so these will also need to be added to the core design and to the toolchain. The implementation of the Trap instruction, as well as the instructions to read/write the special registers is described in \( \text{Section 4.4} \)

---

\(^5\) Linux will not always restore the context of the interrupted process and resume its execution. A timer interrupt can cause the scheduler to select a different process and restore its context. This is not possible when the resume address cannot be be changed by software.

\(^6\) These instructions should only be allowed to be executed when running in system mode to prevent the special purpose registers from being accessed by normal programs. That could be very dangerous; for example a malicious program could completely disable traps which would cripple the entire system.
4.2.6 Design

This section presents the design of our addition to the core to support Traps and interrupts. It is implemented as a VHDL module and added to rvex_system. We will first describe its design, and then describe the signals that were added to the core.

4.2.6.1 The control unit

The core has 2 input signals that control the transfer of control: the \( \text{ita} \) (interrupt target address) signal must be driven with the address where the core must jump, and the \( \text{trap} \) signal that indicates that the core must perform the jump. The trap control unit will drive the \( \text{ita} \) depending on the trap or interrupt type.

The control register that controls traps has a trap enabled flag, and a 4-bit Processor Interrupt Level value. These signals are monitored by the trap controller to see if traps can be signaled, and what the current PIL is.

The IRQMP interrupt controller has a 4-bit IRQ signal denoting whether an interrupt is requested and its priority. The trap controller compares that priority to the current PIL. If traps are enabled and the IRQ has a higher priority than the current PIL, it will write the 4 bits of the IRQ priority into bits 5 - 8 of the \( \text{ita} \) and signal the core to jump there by asserting the \( \text{trap} \) signal. At the same time, the controller will automatically assert the \( \text{ack} \) signal to the IRQMP unit for 1 clock cycle. The unit will then clear the interrupt knowing it is being handled.

The instruction decoder can signal a user trap to the controller. When a Trap instruction is being executed, its 7-bit value is routed to the controller. These bits are written into bits 4 - 10 of the \( \text{ita} \) and a '1' is written into bit 11 as the software trap vectors are stored in the upper half of the table.

4.2.6.2 Added signals

The signals that were added to the core for the trap controller are:

- \( \text{traps_en} \), connected to the control register. Signals if traps are currently enabled.
- \( \text{pil} \), connected to the control register. Represents the current Processor Interrupt Level.
- \( \text{trap_ins} \), connected to the branch unit [7] signals that a Trap instruction is being executed.
- \( \text{trap_arg} \), connected to the decoder of the pipelane that contains the branch unit. Signals the value of the Trap instruction.
- \( \text{write_vcr} \), connected to the fetch unit. Signals that an instruction has been fetched that writes to the control register. Traps will be temporarily disabled as long as this instruction is in the pipeline as it might disable traps (see Section 4.4). [7]

---

[7]: As will be explained in Chapter 5, the core must branch to the address of the Trap instruction for a breakpoint to work.
4.3. VEX CONTROL REGISTERS

- **debug_trap**, connected to the debug unit. Signals the debugger that a breakpoint has been encountered.

4.2.7 The Debug Trap

A special trap has been implemented to create support for software breakpoints. This unit can halt the core, and should do so when a debug trap is executed. The last vector in the trap table has been selected for this purpose; Software trap number 127. When the trap controller detects this trap, a signal is sent to the debug unit that will halt the core. At the same time, the branch unit will perform a branch to the same address where this instruction has been encountered. This is because the debugging software will replace the trap instruction with the original instruction before it resumes the core. That way, the original program is not changed when a breakpoint is being inserted. The debug unit and more details about how breakpoints work will be presented in [Chapter 5](#).

4.3 VEX Control Registers

To configure Traps and Interrupts, a set of control registers has been implemented. These registers can be read and written to by means of 2 special instructions; **movtc** and **movfc**. These registers are not to be confused with the GRLIB general-purpose registers (e.g. the core and debug control registers) that are memory-mapped and can be accessed using the **ld** and **st** family of instructions.

<table>
<thead>
<tr>
<th>Register number</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Control register</td>
</tr>
<tr>
<td>1</td>
<td>Trap resume address</td>
</tr>
</tbody>
</table>

The first VEX Control Register (VCR0) is the most important control register. It is comparable in functionality to the Processor State Register (PSR) that is defined in the SPARC architecture. The least significant bits 0 - 3 are the PIL field, and represent the current interrupt level the core is running in. Bit 4 is the Trap Enable flag.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 3</td>
<td>Processor Interrupt Level</td>
</tr>
<tr>
<td>4</td>
<td>Traps Enabled flag</td>
</tr>
</tbody>
</table>

When a trap occurs, the hardware automatically writes the current PC into the VCR1 register; the Trap resume address. The OS can store the contents of this register...
so it can resume this task at a later time. This is necessary when preemption tasks (see Section 4.2.4).

Table 4.3: The VCR1 register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 31</td>
<td>Trap resume address</td>
</tr>
</tbody>
</table>

4.4 Instruction Set extension

A number of registers were added to the core that allow the newly designed Trap controller to be controlled. Most notably: the trap instruction sends a signal and an argument directly to the controller. It will cause the core to jump to the specified trap vector in the upper half of the trap table (provided that the Traps Enabled flag in the Control Register is set).

To read and write the control registers, the movfc and movtc have been added. They can move data between General Registers and Control Registers. During each cycle, the fetch unit (which is the first pipeline stage) compares the instruction bits that contain the opcode to the instructions that write to the VCR. If a write operation is detected, it will signal to the trap unit that it must temporarily disable traps. This is because software will assume that traps are disabled after issuing this instruction. If a trap happens while this instruction is still in the pipeline, it will have no effect because the core will take the trap and re-SET the Traps Enabled bit in the handler routine. In other words; the trap disable operation will be lost.

Note that these instructions are privileged in most architectures. The VEX ISA does not specify any CPU privilege levels. For ρ-VEX, memory protection and privilege levels are not implemented yet. This might be done in a future release. Until that time, accessing the control registers can be done by any program.

The trap, movfc and movtc instructions have been added to the ρ-VEX core and the toolchain. Figure 7.3 shows an assembly code routine from our Linux port (see Chapter 7) that set the IPL to a given level.

4.5 Conclusion

This chapter presents the designs of the additions and modifications that were made to the initial platform to enable it to run a general-purpose Operating System. First, a number of small but crucial fixes are presented. Then, the design of the Trap controller module for the core is introduced, which can connect to the GRLIB interrupt controller as discussed in Section 3.5. To leverage the added functionality of the new module, additions to the core and an instruction set extension were necessary. Control registers were added to the core design, together with special instructions to access them. The instructions were also added to the toolchain so that they can be used by software.
In this project, a mechanism has been added to the core that allows the execution of the core to be halted by means of a set of control registers. In essence, these registers implement a hardware debug unit that supports halting, resuming, single- or n-instruction-stepping and hardware breakpoints.

The unit can be controlled via the bus, which means that rudimentary debugging is possible by using the GRMON tool with the `mem` and `wmem` commands (read/write memory). However, this manual mode of operation is not very convenient, so we would like to extend the functionality and create a more useful debugging environment. This chapter shows the tools that were ported or implemented to leverage the hardware debug unit and provide more useful debugging functionality.

The first section will outline the goals for our environment, the available options and the choices that were made. The sections after that will explain the implementation of the different parts and the chapter concludes with a number of issues that have not been solved at the time of writing.

### 5.1 Creating a debugging environment

The first question before starting the development of any new tool is probably “why do we need it?”. In this case the answer is both simple and complicated: One could argue that a decent debugger environment is vital for any architecture to have any chance of success. Together with a good compiler, it is one of the first requirements any user will have before he will decide to adopt a new system. On the other hand, ρ-VEX is only an academic platform. It is meant for research, not for application development. It is also in active development by multiple researchers with different research interests, which means that is is possible that the core will change in ways that will render it incompatible with the debug unit unless the debugger will be actively maintained as well.

In other words; the debug unit can be a blessing and a curse. It is an additional function that is very useful for application development but it adds more complexity to the core while it might be more desirable to keep it as “lean” as possible (being a research platform that might change radically in the future).

The question can be answered based purely on the perspective of this individual project. Our final goal is to port the Linux kernel to the current version of ρ-VEX. Common experience for such a task says that a very large portion of time will be spent on debugging.

This process without a debugging environment means adding `printf` statements throughout the code to output relevant data structures and register contents via UART at locations in the code where something goes wrong. A debugger would greatly speed up the debugging process, being able to add breakpoints and step through the code while
monitoring the full state of the processor and memory. Moreover, it is possible that some errors would be virtually impossible to track down without a debugger. Most developers will probably argue that porting Linux without a debugger is very difficult and maybe even impossible.

Using Modelsim, we could trace the execution exactly. However, this cannot replace a real debugger functionality. One reason is that there are always differences between the simulation and the real world. Another reason is that simulation can take a long time. At some point during the project the simulation could take several hours to reach the point where something went wrong in the execution. A real debugger also has features such as breakpoints and stepping that are usable from high-level (C-)code. This will speed up the debugging process so tremendously that one could argue it is vital for the success of the project.

The next questions are “What should our debug environment look like and what is needed to support that?” The most obvious debug tool we can use is the GNU debugger program GDB. It supports most commonly used architectures and is open source. New architectures can be added relatively easy by implementing a new target. To read binary files, it uses the BFD system that has already been ported for the ρ-VEX binutils. GDB also has a facility to connect to a remote target to be able to debug embedded systems that do not run an operating system. Eclipse can connect to GDB so we can integrate the debugging process into the development environment.

To be able to use GDB, we should add a new ρ-VEX target to it and implement a connection to our core. This leads to the most challenging part of the debugger: what modifications are needed for the core to support GDB and how can we create the connection? Let us look at the debugging environment of GRLIB to answer these questions. GDB is also supported for the LEON core and GRMON provides command-line debug functions as well as a remote-target for GDB. It does this by connecting to a Debug Support Unit (DSU) that is attached to the AMBA bus. This connection is supported by a JTAG Test Access Port (TAP), a VHDL module that allows your host system to connect to it using a JTAG cable (USB or Parallel). The module allows software running on the host system to issue arbitrary bus commands to the AMBA bus (see Section 5.4). The DSU is controlled by writing to its control registers that are mapped into the bus’s address space. It connects to the LEON core directly to halt/resume the core, read or write its registers and perform other debug functions.

One of the options was to modify this unit (being an open source VHDL module) and connecting it to our ρ-VEX core. However, it appeared that the Leon core (which is an implementation of the SPARC V8 architecture) was not only far more complex, but also fundamentally different. Furthermore, the GRMON tool that implements the GDB remote target is closed source which would prevent us from making necessary modifications to support our GDB target. Hardware debugging support would need to be implemented in our core directly.

Analyzing the debugging hardware of the Leon did give us insight into how we could create a connection between our host system and the debugging hardware using the TAP. It would, however, be needed to create a custom application on the host system.

\footnote{For example, Modelsim initializes all main memory to zeros whenever there is no data defined for it. On the board that memory contains random data.}
to connect to it and serve as a remote target for GDB. Fortunately, a similar (and open source) application was found that could be modified to serve our purpose (see Section 5.4).

One of the challenges of creating the rvex target for GDB is that the ρ-VEX is reconfigurable. This complicates the debugger both in hardware and in software (the tools need to be able to handle all possible configurations). Therefore the choice has been made to initially only implement the debugger for the default 4-issue configuration. The same assumption has also been made for the operating system (see Section 3.4.2) making it valid within the scope of this project.

5.2 Debug unit

We created a simple hardware debug unit for the ρ-VEX. It supports halting the core, setting multiple breakpoints, instruction stepping and readout for all register types (general, branch, link and control registers).

The debug unit can be controlled by writing to its control registers. Initially, this could only be done by using the GRMON tool from Gaisler. This provided a very basic debugging environment; it involved using the command line to issue low-level commands to the debug unit directly and looking up memory addresses of code segments using vex-objdump. Later sections will show in more detail how the unit is used to support a more useful debugging environment including a server program that can serve as a remote target for the GNU debugger. The design of the debugging hardware is presented here.

Figure 5.1: Simplified design schematic of the debug unit.

5.2.1 Overview

The debug hardware is not implemented in an isolated hardware module. The most important part of the design is a set of control registers, implemented in the rvex_system module. Using the values stored in those registers, combinatorial logic and signals are
used to assert the clock enable signal to the core, halting it when appropriate. Additional ports are added to the rvex module to read the register file when the core is in halted state and a signal is connected to the trap controller (see Section 5.2.2) to be able to detect a debug trap instruction. This is used to support software breakpoints as we will show in Section 5.5.2. The debug hardware is controlled by the registers shown in Table 5.1.

Table 5.1: Debug control registers

<table>
<thead>
<tr>
<th>Address</th>
<th>register name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80000540</td>
<td>dbgctrl</td>
<td>VEX Debug Control register</td>
</tr>
<tr>
<td>0x80000544</td>
<td>dbgarg</td>
<td>Debug Argument register</td>
</tr>
<tr>
<td>0x80000548</td>
<td>dbgretval</td>
<td>Debug return value</td>
</tr>
<tr>
<td>0x80000550</td>
<td>dbgbrk1-4</td>
<td>Hardware breakpoint address 1 - 4</td>
</tr>
</tbody>
</table>

5.2.2 VEX Debug Control register

Table 5.2: The Debug Control register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Halt</td>
</tr>
<tr>
<td>1</td>
<td>Stepping mode</td>
</tr>
<tr>
<td>2</td>
<td>Register readout mode</td>
</tr>
<tr>
<td>3</td>
<td>Ignored (reserved for future use)</td>
</tr>
<tr>
<td>4 - 7</td>
<td>Enable breakpoint 1 - 4</td>
</tr>
<tr>
<td>8 - 31</td>
<td>ignored (reserved for future use)</td>
</tr>
</tbody>
</table>

The **dbgctrl** register operates the debugging hardware. The first (least significant) bit of this register is the halt bit; whenever this bit is asserted, the core will halt. The bit can be manipulated by write operations from the bus, but also by the debugging hardware itself. For example, the bit is set high when the program counter is equal to one of the addresses in the **dbgbrk** registers (provided that breakpoint is enabled in the **dbgctrl** register). The second bit enables the stepping mode. When it is enabled, the debug hardware will use the Debug Argument register to count down to zero. When the stepping bit is high and the Debug Argument is zero, the halt bit is set high and stepping is disabled. As this mode of operation is different from the behavior assumed by GDB (it assumes stepping mode must be disabled by the user), this might be changed in the future. Bit 3 enables the register readout mode. This bit is connected to a set of multiplexers and a decoder in the rvex module. When the register readout mode is enabled, these will connect one of the ports of one of the register files (based on the number in the Debug Argument register; see Table 5.3) to the debugger.
Note that the core will not function correctly when running with the readout mode enabled, as 1 pipelane will be disconnected from the CPU registers.

### Table 5.3: GDB register number with corresponding actual CPU registers

<table>
<thead>
<tr>
<th>GDB register number</th>
<th>Actual VEX register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 63</td>
<td>General register 0 - 63</td>
</tr>
<tr>
<td>64 - 71</td>
<td>Branch register 0 - 7</td>
</tr>
<tr>
<td>72</td>
<td>Link register</td>
</tr>
<tr>
<td>73 - 74</td>
<td>Control register 0 - 1</td>
</tr>
</tbody>
</table>

#### 5.2.3 Debug Argument register

For some functions, an argument can be passed to the debug hardware by means of the `dbgarg` register. At this time, there are 2 functions that need an argument, but more could be added in the future. Stepping mode takes the number of cycles to step as an argument. Register readout mode takes the register number to be read as an argument (see Table 5.3).

#### 5.2.4 Debug return value

The `dbgret` register is used to store the result of a register readout operation. It is connected to the output port of one of the register files (General, Branch, Link, Control registers) depending on the value of the readout mode enable bit in the `dbgctrl` register and the contents of the `dbgarg` register.

#### 5.2.5 Hardware breakpoint address registers

There are 4 registers to specify a hardware breakpoint address: `dbgbrk1-4`. If a breakpoint is enabled in the Debug Control register (see Section 5.2.2), the program counter will be compared to this address and the core is halted when they are equal. In the future, these might be accompanied with registers for watchpoint addresses, that can monitor data address lines and halt the core when a certain operation is being performed to a specified data address.

#### 5.2.6 Operation

With the debug hardware operational, basic debugging was possible by writing into the debug registers using GRMON. The procedure is to start GRMON and first load your program into memory. Then the memory address of some function can be found

---

2For example, when implementing watchpoints, an argument could specify the operation type to trigger the debugger. That way it could distinguish read and write operations. Another feature could be to halt the core when a breakpoint has been hit a specified number of times.
using `vex-objdump`. This address can be written into one of the breakpoint address registers. To start the program, reset it using the core control register. If you have enable the breakpoint in the debug control register, the core will halt when encountering the function.

In this situation you can read/write memory contents using GRMON's built-in functions and use the debug registers to read VEX CPU registers. You can enter a new breakpoint address or enable the stepping mode and run a number of cycles.

## 5.3 Porting GDB

To be able to use the GNU debugger, GDB, a new target needed to be implemented for it. This is comparable to a new architecture for the C compiler and the terminology is analogous. After having studied relevant documentation [35][36], we modified an existing architecture from the latest GDB release to work with $\rho$-VEX. The Binary File Descriptor library was taken from the `binutils-rovex` port and also needed to be modified slightly to work with the much newer GDB release.

Because the BFD was already available, the debugger could already handle `rvxelf32` binaries after a very short development time. The most important modifications were:

- A single instruction is 4 syllables long; 128 bits instead of 32 bits. The Program Counter is always a multiple of 16 as instructions are byte-addressable in our design.

- The data structures that store the CPU registers needed to reflect the $\rho$-VEX’s register layout. Special registers needed to be added (link, branch, control registers) and some registers have specific functions such as general register $\$r0.1$ which is the stack pointer according to the VEX ISA.[5]

- GDB usually wants to skip function prologues automatically[4] Our design is a VLIW which means that relevant operations can be scheduled in the same bundles as prologue instructions, so the prologue should not be skipped for our architecture.

- Our debugging hardware does not support writing CPU registers. This is especially a problem for the Program Counter, as we will see in Section 5.5.2.

With the target now being able to read VEX binaries, what needs to be done next is connecting to the debug hardware to be able to monitor the execution of the actual core. The next section will discuss the steps taken to accomplish that.

## 5.4 Communicating with the hardware debug unit

On a normal system that has operating system support (e.g. Linux), the debugger program itself can be started and be instructed to load a binary for debugging. In this

---

3 for more detailed information, please inspect the git repository.

4 The function prologue sets up the stack frame (among other things, depending on the runtime architecture) at the beginning of a function. It is executed every time the function is called but it does not contain instructions that were compiled from the actual high-level code so it is of little interest to the developer.
5.4. COMMUNICATING WITH THE HARDWARE DEBUG UNIT

In the next sections, a short overview is given of how JTAG is used to create a connection between the host machine and our hardware platform via a USB JTAG debug cable.

---

5 Cross-compile and remote debugging are common in embedded systems as they are often not powerful enough to run Operating Systems and compilers.

6 The precise workings of JTAG, Boundary-scan devices and TAP controllers are beyond the scope of this document. Please refer to the relevant IEEE standards or a textbook on the subject such as [37]. A concise description will be given here.
JTAG is a commonly used name for the IEEE 1149.1 standard, that was designed to facilitate testing of increasingly complex circuits. Today, many IC's implement a debug facility using JTAG. The FPGA used in this project features a JTAG USB connection. It is used to program the FPGA using the Xilinx toolchain, but it can also be used to access various hardware internals after the design phase. Gaisler uses the JTAG cable to allow its GRMON tool to connect to the AMBA bus through which it can control IP cores from GRLIB attached to the bus. We will use the same method to connect GDB to our hardware debug unit. The JTAG cable has a number of pins that can be used by the host machine.

Table 5.4: JTAG signals

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCK</td>
<td>Test Clock</td>
<td>Clock signal driving the TAP controller</td>
</tr>
<tr>
<td>TMS</td>
<td>Test Mode Select</td>
<td>Input signal to change TAP controller state</td>
</tr>
<tr>
<td>TDI</td>
<td>Test Data In</td>
<td>Signal to shift data into Boundary-scan registers</td>
</tr>
<tr>
<td>TDO</td>
<td>Test Data Out</td>
<td>Signal to read data shifted out of Boundary-scan registers</td>
</tr>
</tbody>
</table>

5.4.2 Test Access Port Controller

A JTAG Test Access Port (TAP) controller is essentially an on-chip state machine that controls the Boundary-scan registers on the device (in our case, the FPGA). Using the TAP, a number of boundary-scan registers on the device are accessible from the host machine. Some of those registers are built-in and can be used to control or query the hardware (such as manufacturer id, device serial numbers). As we will see later, designers can also create boundary-scan registers in their hardware design. These registers can then be used to perform any function required. For example, they can be used for communication with the hardware. This is how the AMBA bus in our design can be accessed from the host machine using the JTAG USB cable.

![Figure 5.2: Connecting the HOST machine with the debug unit via a USB JTAG cable.](image-url)
The state of the TAP is controlled by 2 of the JTAG pins; the Test Clock TCK and the Test Mode Select (TMS) pins. By manipulating those pins, the host machine can move the TAP into a desired state.

The most important states are shift-DR and shift-IR. In these states, a value can be written to or read from the Instruction Register or a Data register. When in one of these states, the contents of the selected register is shifted into the Test Data Out (TDO) pin. On the other side, the register shift its new value in from the Test Data In (TDI) pin. This way, the host machine can read and/or write a register by moving the TAP into the shift-DR or shift-IR state and reading the TDO pin and writing to the TDI pin at every TCK cycle.

5.4.3 Boundary-scan Instruction/Data Register

By moving the TAP state into the shift-IR state, the Instruction Register (IR) is essentially placed in between the TDI and TDO pins. The IR selects an operation to be performed, in our case it selects a Data register that can subsequently be read or written after moving the TAP into the shift-DR state.

Some instructions are specified in the standard as mandatory; one of the most important is the BYPASS instruction. This instruction is required because the Boundary-Scan chain can contain multiple devices (each of them having their own IR).

In this case the BYPASS instruction must be shifted into the instruction registers of the devices that

---

5Multiple devices are connected serially, so data shifted in the TDI will pass every device until eventually exiting at TDO. For example, if there are 2 devices both with an 8 bits wide IR, 16 bits should be shifted in from TDI to overwrite both IRs completely. This might be an oversimplification as there could be padding bits involved, so it is important to accurately follow the manufacturer’s specifications.
should not perform any operation. When a device is in BYPASS mode, it will relay data from its TDI pin directly to its TDO pin (which may be connected either to the TDI pin of the next device in the chain or to the JTAG cable’s TDO pin) with 1 clock cycle delay. The board used in this project has 2 devices in its Boundary-scan chain, as shown in Figure 5.4. To issue instructions to the FPGA itself, the IR of the xccace device must be set to BYPASS.

Figure 5.4: Xilinx Impact showing the Boundary-scan chain of the ML605.

Many instructions specified in the standard are optional; these include, for example, an IDCODE register to identify attached components and also one or more USER registers. These registers can be defined by the hardware designer. When user instruction is shifted into the IR, its corresponding data register is selected. That register will be placed in the chain when the TAP is moved into the shift-DR state.

5.4.4 GRLIB JTAG communication module

The GRLIB hardware platform has an AHB - JTAG communication link, ahbjtag, that consists of an AHB Master interface and a TAP module. The TAP uses 2 user data registers; one for the address and command type and the other for data. The address register is 35 bits wide. When issuing a command to the bus, the address must be written into the lower 32 bits of this register. Bits 32 - 34 must contain the operand size for the command, as per the AMBA protocol specifications. The last bit determines whether this is a read (if it is 0) or write (if it is 1) command.

The data register is 33 bits wide; the 33rd bit specifying whether this is a burst command. If this bit is 1, the TAP module automatically increments the address register by 4.

The process of writing 32 bits to a certain address would therefore be:

1. Write the USER1 instruction code into the FPGA’s Instruction Register.
2. Shift 35 bits into the data register, the lower 32 being the address, bits 32 - 34 being the transfer size and the last bit 1.

---

8Xilinx’ implementation of JTAG Boundary-scan registers includes 2 USER registers. They can be instantiated using a VHDL template.
3. Write USER2 into the IR.

4. Write the data into the lower 32 bits of the data register.

After all 33 bits have been shifted into the data register, the TAP will send the command to the jtagcomm AHB master interface that will initiate the bus transfer.

Reading a value is done in a similar manner, but a short wait period is required after writing the address to allow the bus read request to finish and its result to be stored in the USER2 data register.

5.4.5 GDBserver alternative - the OpenRISC advanced JTAG bridge

Writing a program that implements a USB JTAG driver that can interface with the jtagcomm module as demonstrated above is a very complex and time consuming task. As mentioned before, gdbserver is a program that implements GDB’s Remote Serial Protocol, so there is no need to implement that. Extending gdbserver with the JTAG interface would still required a considerable amount of effort.

The Opencores-hosted open-sourced “Advanced Debug System“ project\[38\] implements a hardware debug unit for the OpenRISC1000 CPU. This project also includes a TAP module and software that can interface with it and that also implements the RSP protocol. This tool, called adv_jtag_bridge, supports different types of JTAG cables including the USB cable used in this project and also has a facility that reads BSDL files\[9\]. We decided to modify this tool to support the GRLIB TAP controller already present in our platform.

Modifications on low-level library routines for reading/writing the AHB bus were necessary and the implementation of the RSP commands needed to be rewritten to conform to the way the hardware debug unit works. The TAP design of the advanced debug unit worked using only a single USER Boundary-scan DR, so the low-level communication routines needed to be rewritten as well.\[10\] GDB connects to the advanced jtag bridge using the extended-remote target. Performance of the whole environment is acceptable during debugging, but not for transferring large amounts of data (for example, when loading binaries into memory). Transfer speeds are multiple orders of magnitudes lower compared to GRMON (see Section 8.1.4). As GRMON is proprietary and closed-source, it will be practically impossible to find out how it works exactly. However, optimizations to our code in adv_jtag_bridge with respect to the low-level USB communication may result in considerable increases in performance.

5.5 Breakpoints

Breakpoints are arguably the mostly used tool when debugging. GDB offers two different breakpoint types; hardware or software breakpoints.

\[9\] Using Boundary-Scan Description Language, the manufacturer can store the specifications of the Boundary-scan chain of a device in a file. This file can be parsed by a debug program to automatically configure the connection.

\[10\] Please refer to the Git repository for a complete report of all the changes.
5.5.1 Hardware Breakpoints

The workings of the hardware breakpoints have been described in Section 5.2. adv_jtag_bridge writes the breakpoint address into one of the available registers and enables it in the control register.

5.5.2 Software Breakpoints

The default breakpoint type is a software breakpoint. It works by replacing the code at the breakpoint address with an instruction that causes a trap. In this design, a special trap has been implemented that will cause the trap controller to signal the debugger hardware to halt the core (see Section 4.2.7). When GDB is instructed to create a breakpoint, it will not do anything until the user starts the core. At that point, GDB will first place the debug trap and then unhalt the core. After GDB detects that the core has been halted (a dedicated thread in adv_jtag_bridge checks the core’s status periodically), it will place back the original instruction so the user will always see his original code. As the ρ-VEX is a VLIW, replacing an instruction means replacing the entire bundle. The trap instruction is placed in the slot that is executed by the pipeline that contains the branch unit (see Section 3.4.2). The other syllables contain NOPs.

When execution resumes, the Program Counter points to the instruction after the breakpoint. That means that, without intervention by GDB, the original instruction located at the breakpoint address would not be executed at all. GDB solves this by rewinding the PC to the breakpoint address before unhalting the core. However, writing to the Program Counter is not supported by our debug hardware at this time. This issue needs to be solved before software breakpoints can be used reliably. A workaround has been designed that causes the core to branch to the breakpoint address, but for this to work reliably the instruction cache needs to be flushed before resuming. Hardware breakpoints can be used in GDB by using command hbreak instead of breakpoint.

5.6 Issues and recommendations

The debugger has been implemented in a very short time frame and there is ample room for improvements. This is partly due to the debugger being a necessary tool in the porting process (designed to help solving specific problems encountered during debugging), instead of being a main development objective. This section will list unsolved issues and future recommendations.

- Because of an unknown problem in the assembler, wrong line numbers are encoded in the debug symbols. Setting breakpoints can only be done reliably by using debug symbol names directly (e.g. function or variable names).

- Stepping mode does not work consistently. It sometimes runs an additional cycle and it is not clear what causes this. GDB assumes stepping is implemented in a different manner, so the hardware should be changed to work accordingly.

- The system has not been tested enough. Hardware breakpoints are working in simulation and manual tests (using GRMON). However, in actual usage with GDB it
5.7 CONCLUSION

seems that some breakpoints are lost. Testing is difficult because it is not possible to use Modelsim to simulate the JTAG connection and because of the many communication interfaces that can introduce errors.

- Writing the processor registers is not supported by the hardware. Adding it is not very complicated.

- Software breakpoints need cache-coherency or cache flushing. GDB replaces instructions in main memory. The core is connected to an instruction cache that does not know that the memory it is caching has been changed. It will therefore send the old contents to the core in case of a cache hit. GDB needs a way to invalidate (flush) the instruction cache when placing a breakpoint.

- Writing the Program Counter is not possible. As it is not an ISA register, this will be harder to implement. However, supporting this is very desirable, as GDB depends on it when hitting a software breakpoint (it rewinds the PC so that the original instruction will be executed when execution resumes).

- Without solving the previous 2 issues, software breakpoints can not be used. As a workaround, the debug trap instruction is complemented by an unconditional branch to the current instruction. This, however, does not solve the cache problem.

- The breakpoint hardware could be extended to also support matchpoints/watchpoints.\footnote{The Advanced Debug System uses a scheme where multiple matchpoints can be combined. For example, a complex watchpoint can be created that is activated when the PC is at a given address AND the load address is in a certain memory range.}

- The interaction between the trap controller and the debug hardware could be reevaluated.

- Performance should be improved. GRMON shows that there is ample room for improvement in transfer speeds.

5.7 Conclusion

This chapter presents the rationale and design of a hardware debug unit, the implementation of the \texttt{rvex} target for the GNU debugger (GDB) and the implementation of an RSP server program that can be used by GDB to debug programs running on our hardware platform by allowing it to connect to the debugging hardware. A debugging environment is important step in accomplishing our goal of establishing an effective development environment (see Section 1.3). Furthermore, the process of porting an OS will include a lot of debugging which will be sped up tremendously by having proper debugging tools. The debugging hardware and software have been designed in a short time frame and there are many improvements possible on both sides.
In previous chapters, the design of our hardware platform has been presented. See Figure 3.4 for an overview of the platform and the designed modifications. In addition to supporting the new platform, the toolchain must be able to successfully compile Linux kernel images as will be discussed in Chapter 7. In this chapter, we will discuss the toolchain, the modifications needed to support the hardware platform and the Linux kernel (see Section 6.4.1), the issues that were found and the fixes or modifications we made to solve them (see Section 6.4.2).

6.1 Software toolchain and workflow

One of the advantages of the $\rho$-VEX over other reconfigurable softcores is that an existing architecture was used as a starting point. The ISA was already defined by HP, and a toolchain was made available as well. This toolchain included an industrial-strength compiler that can also compile source files to a VEX simulation. STMicro added the $\rho$-VEX core to their xSTsim simulator.

A simple assembler has been created by TU Delft along with the first implementation of the rVEX softcore (presented in [29]). However, one of the goals for the Platform is to be fully open and therefore the TU Delft decided to create VEX ports for the GNU binutils and GCC. IBM has been the main contributor for the latter. As with every new Platform, a new toolchain will not work flawlessly from the start. Section 6.4 presents the modifications and fixes that were made to the toolchain during this project.

For development, the Eclipse IDE is used. The kernel compilation is done using our $\rho$-VEX versions of GCC and binutils. The compiler can only produce assembly output that must thereafter be processed separately by the assembler. This is in contrast with most compilers, that can be configured to produce object files directly. The kernel build system (being comprised out of multiple Makefiles for many kernel subsystems) needed to be modified to reflect this (in other words; it has configured to use our toolchain and to use separate commands for compiling and assembling). See also Section 7.3.

The binary images produced by a build command started out at less than 1 MiB in size. As more architecture-specific code and kernel features were added, this size grew to 6 MiB when including debug symbols (that are needed by the debugger) or 1.7 MiB when excluding them.

Binary images can be examined using the objdump tool from binutils; it has multiple features including a very useful disassemble function. The binary image can be loaded into the memory by using the GRMON tool. Technically, the ported debugger application could also be used, but its transfer speeds are prohibitively low.

When the image has been loaded into memory, the core can be reset by writing to the control register (see Section 3.4.3.2). This can be done using GRMON or GDB. As
the kernel boots, its output is printed to the serial interface that can be monitored using *minicom*.

### 6.2 Generating code for different machine configurations

The VLIW approach leaves the responsibility of resolving all pipeline hazards (see [4]) to the compiler to minimize hardware complexity. To be able to do this, the compiler needs to know all the relevant characteristics of the processor. As the $\rho$-VEX is reconfigurable, our compiler needs to be able to compile code for different core configurations. The HP compiler (discussed in [Section 1.1.2]) uses a machine configuration file. All the configuration parameters are documented in [5, Section A.6]. The compiler needs to know the available resources (e.g. the issue-width, number of MUL units), the delay of all the operations, and the number of registers. For GCC, these values are fixed and can only be changed by modifying the VEX Machine Description (MD) and recompiling the compiler. GCC has been configured to compile correct code for the default configuration we are using throughout this project.

The assembler needs to know for what issue-width the code has been compiled. It must also know the pipeline configuration; in which lanes are the different functional units located. This is because it must place operations for a certain unit in the syllable that is executed by the pipeline that actually contains that unit. For example, if the branch unit is located in the first pipeline, then every branch instruction must be placed in the first syllable of the bundle. Vice versa, if the Load/Store unit is located in the third pipeline, then every memory operation must be placed in the last slot.

Lastly, the compiler needs to take into account the possible presence of “long immediates”. A value stored inside the instruction is called an immediate. It is called a long immediate if it is too large to fit within the instruction encoding. For example, the assembly instruction

```
add $r0.1 = $r0.1, VALUE
```

will fit inside a single syllable only if VALUE is representable with less than 10 bits (this is determined by the $\rho$-VEX instruction encoding). Otherwise, the assembler will insert an extra syllable containing bits 32 - 10 and information on which syllable this long immediate belongs to. The hardware will concatenate the parts during execution. However, the assembler cannot insert the extra syllable if the bundle is already full. Therefore, the compiler needs to take into account if a long immediate is needed an reserve an open slot for it.

### 6.3 Compiling for Linux

For most programs, the HP VEX compiler is the compiler of choice. However, Linux makes use of some very specific extensions [39] that are only available on GCC. It is therefore the only compiler that can reliably compile a Linux kernel, and is stated as a requirement in the Linux README file. Our *rvex* GCC port is based on the 4.6.0 release from 2010 (flagged experimental, see [Figure 8.3]). The binutils port is based on version 2.20 from 2009.
During the project, a number of issues have been encountered in the toolchain that needed to be fixed, and some additional modifications were necessary before the first binary kernel image could be produced. An overview of those modifications and issues is presented in the next section.

6.4 Toolchain Development

As mentioned before, the toolchain had not been used to compile programs whose complexity is comparable to the Linux kernel. This has the consequence that it is very likely that the Linux kernel contains code constructs that have not been seen before and might not be compiled correctly. These issues obviously need to be found and fixed before the toolchain can compile a bootable image.

As the maturity of the toolchain is very important for the ρ-VEX Platform, this project is an excellent opportunity to put it to the test and improve it. A well-functioning toolchain is also required to accomplish our goal to establish an effective development environment (see Section 1.3). At the start of the project, the Git repositories were copied to create a local version of the development tree. All the fixes were performed on this separate tree.

Note that compiler internals are beyond the scope of this document, so it will be assumed that the reader is familiar with the concepts. For GCC, the online documentation [41] and the internals document [42] are valuable references. Binutils documentation is also available online [43].

6.4.1 Modifications

This section lists the modifications that were made to the toolchain. For every item it discusses the old behavior, the desired behavior and the modifications.

6.4.1.1 Linker

Current behavior

1. The linker uses a built-in default linker script that is created to link simple programs for the standalone version of the core. As described in Section 3.1, a requirement for our system is to have a single address space and the standalone version of the core uses 2 separate address spaces.

2. The linker did not correctly link relocatably; wrong branch offsets were inserted into the final binary.

3. The normal linker scripts used to compile ρ-VEX binaries define certain memory locations such as __DATA_START to allow references to be made in the code to those locations. These references will be resolved by the linker. Linux also needs

---

1There is no literature available reporting the development of the ρ-VEX target for binutils and GCC, so it is unknown how well they have been tested.

2For an overview of the Git version control system, refer to the online documentation or [40] (also available online).
these memory definitions to calculate the available memory and to initialize it. However, the variables can only be defined once, otherwise the linker will find multiple definitions in the final link step.

Desired behavior

1. The memory system has been changed from a pure Harvard architecture into a Modified Harvard architecture. This must be reflected by the linker script, that decides where to place the sections into memory. When instruction and data memory are separated, their respective sections are placed into separated overlapping memory spaces (because both memories start at address 0, both sections are placed at address 0). This method is used by the built-in linker script.

2. Linux requires multiple linking steps, linking together libraries, device drivers, filesystems and other kernel components into the final binary image. The intermediate object files need to be relocatable.

3. Only the final linker script (that produces the full kernel image) should define these variables.

Modifications

1. The linker script has been modified to use a single address space.

2. The cause has been found in the way the assembler and linker compute and store relocation records. Binutils (or to be more specific, the Binary File Descriptor library) supports different styles of keeping this information; information can be stored either in the instruction itself or separated in the relocation record table. Also, branches can be stored using absolute values or PC-relative (jump distances relative to the program counter). Our \( \rho \)-VEX port uses a slightly different method, and the linker expects to find relocation records created by the assembler. A patch has been written for \( \text{ld} \) that correctly handles relocatable links.

3. Multiple linker scripts were created for the different linking steps. The final script defines all the needed variables. The scripts were incorporated into the build system (see Section 7.3).

6.4.1.2 Compiler - GCC

Current behavior

- The rvex backend for GCC includes the icall and igoto instructions.

---

3Linking relocatably involves linking the object files without writing final memory addresses into the resulting object file. Instead it will keep symbol records in a table that reflects the locations of all the symbols relative to the base address of the file. This information can be used in the final link, when the object file is placed in its final location thereby knowing its base memory address.

4See the Git commit for more information.
• The GCC function prologue was implemented in such a way that callee-saved registers were stored below the stack pointer before it was updated to its new value (by subtracting the size of the function’s stack frame).

Desired behavior
• The \texttt{icall} and \texttt{igoto} instructions have been removed from the VEX instruction set. HP decided to overload \texttt{call} and \texttt{goto} to support their purpose, so it was also eliminated in the assembler.

• The VEX runtime architecture \cite{5} (more commonly known as the Application Binary Interface, ABI) explicitly states that programs may not write below the stack pointer (GR $0.1$).

Modifications
• The instructions were modified in GCC.

• The function prologue has been modified to comply with the ABI; it updates the stack pointer in the first cycle of a function. Changes to the frame allocation were also needed, because GCC uses GR $0.62$ as the frame pointer and this is a callee-saved register. Therefore, the frame pointer could be updated only after it had been saved.

6.4.2 Fixed issues
In addition to the modifications presented in the previous section, a number of issues were identified while working with the toolchain. These issues and their fixes are listed in this section.

6.4.2.1 GCC
• Another problem was found that was being caused by the function prologue. If GCC enters a non-leaf function (meaning there is another function being called from this function), it must save the contents of the Link Register (LR). Otherwise, the return address of the calling function will be lost when the new function call is executed. The function prologue stored the LR in the first instruction. This caused a problem in our $\rho$-VEX core, because there is a delay involved when performing a call instruction. A call instruction updates the LR with the return address (i.e. the location where the program must continue after the function is finished). It takes 2 cycles for the LR to be updated. This means that the function prologue will save the old value instead of the return value. The prologue was modified to wait a cycle before saving the LR.

• A rare case was found where GCC did not write an immediate value (a value that will be coded directly in the instruction) to the assembly output. It only occurred when the value was equal to 0. After debugging the code, an erroneous \texttt{if/else} statement was found and removed.
• The VEX architecture header file wrongly defined the `JUMP_TABLES_IN_TEXT_SECTION` variable. This variable specifies if GCC can use the `.text` section to store jump tables. These tables are used in some optimized switch statements, and contain addresses of the jump targets. VEX can not read PC addresses directly from memory. It can only read an instruction containing a jump, or it can read an address from memory into the Link Register and use that as the jump target. Another result of inserting the tables in the `.text` section was misalignment of the code following the table. Concluding: jump tables must be stored in the `.data` section so this option was disabled.

• GCC’s scheduler needs to know which instructions it can dispatch at the same time, as we are dealing with a VLIW architecture. There are two factors to consider; code dependencies (which GCC can check internally) and hardware limitations. This last factor is architecture-dependent, so it needs to be handled by architecture-specific code (in other words, this code was written for our ρ-VEX port). One of these limitations is the number of long immediates (see Section 6.2) that fit in this bundle. If a long immediate is detected, the number of instructions that the scheduler can place in this bundle should be decreased by 1 (thereby reserving an empty slot for the assembler to place the long immediate in). The code that detects this did not work correctly in all cases. This is related to the VEX Machine Description, that is not fully consistent in the way it describes all the instruction types. As revising the MD is beyond the scope of this project, a workaround has been created in the code to allow the compiler to check more instruction types for long immediates.

6.4.2.2 binutils

• The assembler writes incorrect values into the relocation table in some cases. The cause was found to be related to alignment. Every instruction (bundle) in the binary must be aligned according to the issue-width of the core. GNU assembly syntax defines a keyword `align` that should align the next code at a specified boundary. However, our assembler always aligns the code and the keyword actually caused misalignment. Removing the keyword from the GCC function header definition fixed the issue.

• An issue was found that caused the linker to compute incorrect relocation values for `.lcomm` entries. This issue is related to the earlier fix of relocation record handling by `ld`. A workaround was created, but a more definitive solution is desirable (for both issues; this might include modifying the behavior of both `as` and `ld` with respect to relocation records).

6.4.3 Unfixed issues

Some issues that were encountered did not directly impair the compilation of a binary image. They will remain unsolved in this project. In this section, these issues are listed and recommendations will be made for possible approaches to solve the problem.
Our GCC port produced various error messages that the GCC version on our development machine (version 4.5.0 i586) did not. It did not accept switches where labels were not followed by code. This construct is used in multiple locations in the Kernel. We did not try to fix this issue, as the syntax defined in the C standard explicitly states that a statement is required after a named label (See the ISO/IEC 9899 standard [44]). The solution was to slightly modify the code itself so it would be accepted by the compiler.

Another unresolved issue was the occurrence of Internal Compiler Errors (ICEs). These were located in code that was not modified by our architecture-specific modification to add the rvex architecture. As such, it would be very difficult to trace these errors and beyond the scope of these project to fix. The solution was to work around the problem. The problem was caused by duplicate function names, so renaming a duplicate function resolved the issue.

In some cases, GCC wants to load a value from memory directly into a Branch Register in an attempt to directly compare some memory location with 0. This is not supported by rvex; a BR can only be manipulated by certain instructions (e.g. compare operations) or by moving a value from a General Register. This behavior is specified by the Machine Description that models the $\rho$-VEX, but for unknown reasons GCC does not comply to these specifications in rare cases. The reason has not been localized, as the MD does not seem to influence these cases (implying the fault lies in another part of the code, again not architecture-specific). Working around this issue involved rewriting a compare operation in the code.

The Assembler does not input correct line number information into debug symbols when assembling with debug symbols enabled. This problem only occurs for SLINE entries, FUN entries (specifying start addresses of functions) are correct. The assembly code input is correct, so the problem must lie in the assembler. No cause was found as it was not localized in area’s that are affected by our port.

GCC does not always respect the instruction delays specified in the machine description specifically when the dependencies cross basic-block boundaries. These Inter-Basic Block scheduling issues are discussed by [45], and a custom-made solution is presented that could serve as an example for our back-end. Implementing such a solution, however, is beyond the scope of this project. The most important manifestation of this problem, the LR update delay, was circumvented with a workaround in the function prologue (see Section 6.4.2.1).

\footnote{In RISC-like architectures, the hardware itself can detect these dependencies, and stall the pipeline when needed. VLIWs are designed for minimal hardware complexity, so they typically cannot perform these checks. It is therefore the responsibility of the compiler to insert NOPs to observe the dependencies.}

\footnote{A basic block is a portion of code that is always executed in a straight line (i.e. there are no jumps to/from anywhere within the code block).}
6.5 Conclusion

This chapter discusses the toolchain of the ρ-VEX Platform in relation to our goal of porting an OS to our hardware platform. First, the toolchain is introduced and Section 6.2 discusses how it can generate binaries for different hardware configurations. Then, the development of the toolchain during the project is presented. A number of modifications were necessary to support our hardware platform and to support compiling for Linux. Furthermore, several issues were encountered and fixed during the project. In the end, the toolchain could compile a bootable Linux kernel image, thereby satisfying our goal. However, it is unlikely that all issues have been found. The toolchain is not yet perfect and more work is needed to improve it. Multiple problems were solved by a workaround in the code, instead of revising underlying problems in the way our architecture was implemented. The VEX Machine Description in GCC should be revised, and an extra pass should be added to GCC to detect inter-basic block hazards. Binutils should be updated to a newer version, and the way that relocations are implemented in the assembler and linker should be revised.
7

Linux

In Chapter 3 the hardware platform for this project is defined. Chapter 4 presents the additions that were needed to support our OS. This chapter presents the process of porting Linux to our GRLIB-based hardware platform, starting with an introduction of Linux, argumentation of the choices that were made in this project, followed by details about the Linux version we are using and the porting process itself. An evaluation is presented in Chapter 8.

Operating Systems are very complex in nature and Linux is no exception. It is beyond the scope of this document to elaborate on every concept that is necessary to understand Linux and the porting process. Therefore, in this chapter it is assumed that the reader is familiar with general Operating System concepts. There are many excellent textbooks available on this subject, such as [46].

7.1 Introduction

Linux is one of the most popular and well-known open source Operating Systems available today. Porting an OS can only be done if the source code is available, so only open source operating systems could be used in this project. There is a large number of reasons to use Linux as the OS of choice. It is very beneficial for any platform to support Linux, because of the following reasons:\(^\text{1}\)

- Hardware and software support. There is an enormous amount of drivers and programs available for Linux.
- Scalability. Linux runs on mobile phones and microcontrollers, but also on the largest supercomputers in service today.\(^\text{2}\)
- Linux is open source and can be used free of cost.
- There are many active developers contributing to Linux.
- More and more hardware and software vendors now support Linux.

Another important characteristic of Linux is its portability\(^\text{3}\), which makes it a very suitable operating system for this project. To keep Linux portable, all architecture-specific code is kept separated from the generic code. In the ideal case, only that code

\(^{1}\text{Paraphrased from [47].}\)

\(^{2}\text{According to the TOP500.org November 2013 list [48], 414 of the 500 fastest supercomputers is running Linux.}\)

\(^{3}\text{Initially, Linux was written for the Intel 386. When support was added for different architectures (starting with the DEC Alpha), parts of the kernel were rewritten to make them portable. Now that Linux is running on so many different architectures, portability has become very important [49].}\)
needs to be rewritten for our new architecture. This is of course not a trivial task, but it keeps the process clear.

7.1.1 Operating System components

Linux itself is actually only an OS kernel. All it does is process management (CPU scheduling), memory management, device management, system calls and generally also networking. Linux itself does not include any application software. For a completely usable system, the following components are needed:

- The kernel (with drivers for our hardware)
- A Filesystem
- A C standard library (like the GNU C library glibc)
- User space programs (most notably core-utils such as ls, cd, etc)

Note that a Linux OS is very complex and there are endless different ways it can be broken down into components. This breakdown has been chosen because all items represent a separate task towards the goal of implementing a full system. These items will be discussed in individual sections after we have introduced the selected distribution in Section 7.2.

7.1.2 GNU/Linux distributions

In common language, the term “Linux” refers to a full system containing all of these components. There are many different versions of such an integrated system. They are usually called distributions or “distros” (e.g. Debian, SuSE). Officially, such a system should be named GNU/Linux because all of the libraries and tools stem from the GNU project [51] [52].

An other important argument to choose Linux as our operating system, is the availability of uCLinux (microcontroller Linux) which is essentially a special distribution developed to run Linux (a modified version) on very limited hardware. This means that we will have a full-fledged OS with relatively minor limitations (which we will see in Section 7.2.5) uCLinux claims to be the “the world’s most portable embedded operating system” [53]. It includes a number of different Linux kernels, standard C libraries and a large collection of programs that are ready to run on uCLinux systems.

uCLinux is meant for embedded systems with small amounts of memory and systems that lack a memory management unit (MMU). While our platform does not suffer from insufficient memory, as of yet the ρ-VEX does not have an MMU and designing one for
this project is not realistic because of two reasons: it is a difficult and time-consuming task and it makes the memory management subsystem of the operating system substantially more difficult. Porting an OS with memory management (memory access protection, paging) to $\rho$-VEX would be difficult and time-consuming for a master thesis project as even an OS without memory management already is a challenge. For these reasons, uCLinux has been selected as the distribution of choice for this project.

7.2 uCLinux

uCLinux started out as a separate project that was not much more than a collection of patches for the Linux 2.0 kernel that changed the kernel in such a manner that it could run without a Memory Management Unit. The MMU normally enables every process that runs on the system to have its own private address space, that is protected from other processes while at the same time preventing non-system processes to access memory that is reserved for the operating system (such as kernel code and data structures and memory-mapped hardware). It also enabled paging that 1. allowed processes to use more memory than actually present on the system and 2. allowed memory in use by other processes that are not being executed to be temporarily removed from memory (swapping).

The uCLinux distribution is an important part of satisfying our project objectives: it will be the core of the development environment. Developing an application for the uCLinux system is easy and by integrating it into the build system (see Section 7.3), binaries are compiled and added to the generated filesystem image automatically.

7.2.1 uCLinux Kernel versions

The uCLinux distribution comes with 3 different releases of the Linux Kernel: the latest and currently supported 3.x and the end-of-line 2.4 and 2.0 versions. Although these later two are officially not supported anymore, we decided to use the 2.0 version because of the following reasons:

- The codebase is much smaller and the kernel code is considerably less complex.
- starting from Linux 2.4, uCLinux’ NO_MMU code was included into the Linux Mainline and the arch/ directories were consolidated by merging the ordinary and *nommu versions of the architectures (using #ifdefs in the code and the kernel configuration script to select which code will be compiled). The separate ports of the 2.0 kernel, however, made the code itself much more clear.
- The newer kernel versions are using more compiler directives (using the __attribute__ directive), some of which are not supported by our GCC port(such as the section attribute).

\(^6\)For example, the 2.0 Linux kernel was not preemptible yet. This means that processes could not be suspended while running kernel code.
In short, to minimize risk and complexity, we decided to use the 2.0 kernel. Once the 2.0 port is operational, the VEX architecture-specific code can be moved to the latest kernel version more easily compared to starting development on the latest version from scratch.

### 7.2.2 Standard C library

The standard C library provides many basic functions that most programs need, to save programmers from having to implement those functions themselves and to provide a well-defined set of interfaces and functionality. The library is specified in the ANSI C standard [44]. Many of the functions in the library rely on the Operating System. They have to use system calls to perform their task, so every OS must have its own implementation of those functions. In other words, the C library is the interface between software programs and the OS. Even the simplest program needs some functionality from the C library; “hello, world” relies on `stdio.h` to use the `printf()` function.

The uCLinux distribution contains 3 different system libraries that the user can choose from. These also contain libraries that are optimized for microcontrollers, like the `uCLibc` library that is the default in uCLinux systems. Normal Linux systems use the `glibc` GNU C library. `uCLibc` was designed to be much more efficient in terms of memory use. Consequently, programs that were compiled using `uCLibc` are significantly smaller compared to using `glibc` [54].

To be able to compile user space programs, we will need to port `uCLibc` to our platform. This task will be done in a future project because of the limited time frame. This means that we will not be able to run any programs on our system yet.

### 7.2.3 Filesystem

Linux, being a UNIX derivative, is designed in a filesystem-centric way [26, pg. 12]. It works with a single unified filesystem, that is the main means of communication between programs [55] and considered a central component of the OS [56]. It is unified because it uses a single rooted tree from which all media can be accessed (denoted as “/”). When a new storage medium is inserted (e.g. a CD or USB drive), its filesystems tree can be “mounted” on a directory in the unified tree, making the new mediums tree available under that directory. This is in contrast to some other systems, that use a separate tree with its own root for every storage medium (e.g. drive letters in MS Windows).

Linux needs a filesystem to mount at the root of its tree. This filesystem usually contains the most important system files and is called the “root filesystem” [17, pg. 132]. Mounting the root filesystem is one of the first steps that the kernel will perform after initializing the hardware and essential data structures. The kernel cannot boot without a root filesystem. In a normal system, the root filesystem will usually be located on the hard drive. In embedded systems, the root filesystem will often be located on Flash storage.

---

7Technically, it is possible to modify Linux to run without a filesystem, but it would be useless as every program that can be executed is stored in a file. Otherwise, you would have to compile your application into the kernel itself but this would be almost equivalent to running the program bare-metal.
7.2. uCLinux

uCLinux provides a number of options to generate a root filesystem. It will copy all needed files and programs into a directory and run a set of tools that will create an image (see Section 7.3). This image is a file that contains a formatted filesystem of a certain type, that can be copied onto the board. Generally, this image will be written into Flash memory a single time, and only updated afterwards when necessary.

In this project, we have chosen to use a ROMFS image that will be loaded directly into main memory (RAM). Linux will address this filesystem as a RAMDISK. The reason is that this does not require us to develop and/or debug a Flash storage driver for our board, and because our filesystem will not contain any programs yet as we will not have any user space programs.

7.2.4 User space programs

The uCLinux distribution comes with a set of user space programs that were modified to run on a uCLinux system (see Section 7.2.5). They have been tested on the NO_MMU Linux kernel and can be easily configured to be added to the build (see Section 7.3). Programs like `ls` (list the contents of a directory) and `cd` (change the current working directory) are basic components needed to make a Linux system usable.

The final step in the boot sequence is a call to a shell (e.g. `/bin/sh`). This is a user space program that can receive input from the user and provide output. In embedded systems (such as our hardware platform), this is usually done through a UART serial connection. The shell is the interface between the system and the user; the user can type commands into the shell and see the result on the screen. As we have seen in Section 7.2.2, we cannot compile any programs for our system yet. This means that the call to `/bin/sh` will fail, leaving the system with only the kernel threads running.

7.2.5 Limitations

This section has been paraphrased from [57], [58], [53]. The absence of a MMU causes the following technical limitations:

- No virtual memory; the system has a single address space. This also implies no copy-on-write.
- No memory protection. Every process can access every memory location.
- No paging. This also implies no swapping. Programs also have to be loaded into memory completely, instead of only loading the pages that are being used (on-demand paging).
- No dynamic stack. A write beyond an application’s allocated stack will go undetected (resulting in undefined behavior).
- `sbrk()` and `brk()` are not supported. Memory allocation (`malloc()`) is implemented using a global shared memory pool.

These technical limitations result in the following practical limitations:
• Every process runs in the same address space. This means that every process has access to the memory of every other process, including kernel code and data.

• Because binaries can not run in their own address space starting with virtual memory address ‘0’, every process must run in a unique memory region and require runtime relocation or Position-independent code (PIC). Normally, a program expects to be loaded at address 0 (zero), because it has its own address space.

• Because allocating memory is handled in a different way and copy-on-write is not supported, forking a process can only be done using the vfork() system call instead of fork() (see Section 7.2.5.1).

• The mmap() system call should only be used on sequential an contiguous files, otherwise it will need to copy the entire file into newly allocated memory.

• No tmpfs. Instead, fixed-sized RAMdisks can be used.

• Stack sizes are fixed, and care must be taken that applications will never exceed the allocated stack memory. The default stack size can be changed at compile-time.

• The memory pool implementation of malloc() can cause problems related to memory fragmentation, and any process could use up all available memory (starving the other processes). Memory fragmentation can only be solved by restarting programs, as allocated memory can not be moved as it is in use by a process.

• Shared libraries work differently, and must be compiled for XIP. Otherwise, a full copy of the library must be loaded into memory for every program that uses it (this would be worse than statically linking the library).

7.2.5.1 Fork-exec

In normal Linux, the fork() system call causes a “copy on write” copy of the process’ address space to be created. This means that for read-accesses, Linux will just reference the memory of the parent process. When modifying the child’s memory, a copy of the page that is being modified is created for the child. uCLinux does not support copy-on-write. The most important difference between fork() and vfork() from the applications point of view, is that a vfork() needs to be followed by a exec() or _exit() call by the child [59]. uCLinux states this as a limitation where the parent is blocked until the child calls exec() or exit() [53]. You can implement forks using vfork() in such a way that the child process has its own variables, but you might need to modify the source code in a non-trivial way. A common use of fork is actually Fork-exec [60], where a new application is started by a parent by calling vfork() and have the child switch to another binary (thereby starting the new application) using exec() immediately. For comparison, MS Windows does not feature system calls like fork(), but it has functions that behave like calling fork() and exec() immediately after each other. One of the

8eXecution In Place. Running a program directly from its storage location (usually Flash memory) without loading it into main memory.
large advantages of uCLinux is that a lot of software is included in the distribution that has already been adapted for this limitation and is ready to run.

### 7.3 The build system

For a more elaborate introduction to build systems, please refer to [47, pg. 445]. uCLinux has a build system that is based on the Kconfig; the Linux Kernel build system. It is used to configure which parts of the distribution are compiled and included in the project. This applies not only to the Linux kernel itself (that can be configured as a subcomponent), but also to user space applications and libraries. These can be selected to be compiled and included in the filesystem image that will be produced by the build system.

Another important task of the build system is the hardware configuration; There is a configuration for every supported board. The first thing that needs to be done when adding support for a new board, platform or architecture, is to create the necessary configurations for the build system. These will make sure that the correct files are being compiled, and that the correct toolchain is used.

In our case, some manual modifications were necessary because we did not have a standard C library and no user space applications ready to compile. These steps needed to be disabled in the Makefile to allow the build system to proceed to the final task of generating the filesystem image.

### 7.4 Porting Linux

The next step is to create a new directory for the architecture-specific code. Usually when porting software, the best thing to start with is an existing port of a similar architecture. In our situation, there were 2 ports that were similar to the ρ-VEX hardware platform we are using, namely the st200 port by STMicro and the sparcnommu. This last one is similar because it is a nommu version, and because we are using a SPARC-like interrupt system (see Section 4.2) and a number of peripherals from GRLIB (the interrupt controller, timer, and UART). The ST200 port is not included in the Linux Kernel Mainline nor in the uCLinux distribution and uses the 2.6.11 kernel version. Therefore we decided to use the sparcnommu port as a basis, and to use the st200 port for reference whenever the code needed adjustments because of architectural differences. For example, a lot of arch-code for SPARC handles the register windows which is not necessary on ρ-VEX. On the other hand, these register windows facilitate some facets of context switching (both for different tasks and interrupts) so new code needed to be written to support the same functions on the ρ-VEX.

#### 7.4.1 Steps

The development process went as follows:

- Create a new target for the build system, including Makefiles and linkerscripts
- Create new directories that will contain the architecture-specific code: arch/VEX and /include/asm-VEX.
• Create stubs for all low-level functions; these will be implemented later.

• Start implementing initialization code; Head.S and start.c.

• start.c contains the main() function that is called by Head.S after the BSS and stack are initialized. It will initialize early console output and call start_kernel(), which is the kernel’s entry point.

• Implement the low-level initialization functions that are called from start_kernel().

• Implement low-level context-switch code, IRQ handling, and configure the timer to start firing interrupts (timer ticks).

• The kernel will now start forking multiple kernel threads, this also requires low-level code.

• After creating the first thread (which has not executed yet at this time), the startup thread will drop into the idle loop. At this time, the scheduler is being called for the first time. It will select a new kernel thread to run next, so the first process switch is being performed. This also needs architecture-specific code.

• The first new thread, init, will mount a root filesystem. At this point a filesystem image is needed.

• Implement low-level code for system calls.

7.4.2 The boot process

This section describes all the steps of the boot process. When the core is reset, it will start executing at address 0. This is the location of the reset vector, and it contains an unconditional branch to address 0x1000 which is located directly after the trap table. The system entry point is stored here; it is the assembly code of the Head.S file (see Figure 7.4). This code initializes the core so it can properly execute C code. It writes a valid location into the stack pointer (so that it points to the end of memory), zero-initializes the BSS section and jumps to main(). This function is located in start.c, and is still architecture-specific startup code. It initializes the serial port so we can view output very early in the process. Then it performs a sanity check to see if it can properly access memory and then calls the start_kernel() function.

In most systems, the number of steps that are needed before start_kernel() can be called is larger (see Appendix A). It usually involves initializing caches and MMU and running a boot loader. This boot loader initializes required hardware to load the kernel from some device into memory. Our hardware platform is simpler, and we must manually load the kernel into memory before booting.

The idle loop repeatedly calls the scheduler. It should never return. It can only be executed by process 0 (the startup process). The idle loop will only run when there are no other processes runnable. Modern systems have special instructions, such as HLT on x86, that will cause the CPU to go into a low-power mode until an interrupt is received.
7.5. CONCLUSION

From `start_kernel()`, various initialization functions are being called. The first step is to initialize the systems memory. For example, the memory is divided into pages and datastructures are initialized that administer the availability of all the pages. In the next step, traps are enabled. Then, before enabling interrupts, the system configures the IRQ handlers (at this time, only the timer IRQ is being configured). The next steps are to initialize the time system, the scheduler, the console, the datastructures for kmalloc (the kernel memory allocator), and to enable interrupts. They are needed for the following step which is calibrating the busy-loop delay (measuring the “BogoMIPS”\(^{10}\)). After this step, the system initializes various datastructures for the filesystem (e.g. inodes, name tables), buffers, and sockets. The last steps of `start_kernel()` is to create the `init` process and then drop into the idle loop. The idle loop will call the scheduler, which will select `init` to run. From `init`, more kernel threads are created (`bflushd`, `kswapd`) and function `sys_setup()` is called. This function initializes all the hardware devices. This includes the RAMdisk that will load the root filesystem image from a specified memory location. After loading the image, the filesystem is mounted.

At this time, the system is ready to proceed into user space and start the first user processes (by running `/etc/init` or by creating a new kernel thread that reads the configuration stored in `/etc/rc`). As there are no programs in the filesystem that can be started, `init` will drop into its default behavior of periodically calling `wait()` (see also Section 8.2.1).

7.4.3 File listings

Figure 7.1 and Figure 7.2 show the architecture-specific files that were implemented during the project. Most of them have been adapted from the corresponding files from the SPARC or ST200 code (see Section 7.4).

7.4.4 Code examples

This section include a number of important code sections (full routines or fragments) from our port. Figure 7.3 shows how the OS can temporarily disable interrupts by writing a value into the IPL section of the VCR. The `mov $c0.x` operations will automatically be converted to `movtc` or `movfc` instructions by the assembler because of the control register operand.

7.5 Conclusion

This chapter discusses the process of porting an Operating System to our hardware platform that was implemented in previous chapters. We decided to use Linux for our OS; the Linux 2.0 `no_mmu` kernel to be more precise. The chapter starts with a short introduction of Linux and lists the arguments for its use in this project. Subsequently, a short description of OS components and Linux distributions is given. The uCLinux distribution and its `no_mmu` kernel allows us to use Linux on our ρ-VEX core with relatively

\(^{10}\)BogoMIPS a unit conceived by Linus Torvalds, and is commonly defined as “The number of million times a second a processor can do absolutely nothing” 61
CHAPTER 7. LINUX

Figure 7.1: Architecture-specific files in the arch/VEX directory

```
./lib/unimplemented.c
./lib/div.c
./lib/swap.c
./lib/delay.S
./lib/divrem.c
./lib/ldiv.c
/mm/fault.c
/mm/init.c
/kernel/irq.c
/kernel/traps.c
/kernel/cpu.c
/kernel/setup.c
/kernel/sys_vex.c
/kernel/syscall.S
/kernel/ptrace.c
/kernel/process.c
/kernel/signal.c
/kernel/rirq.S
/kernel/switch.S
/kernel/heads.S
/kernel/time.c
/platform/RVEX/entry.S
/platform/RVEX/start.c
/platform/RVEX/trap_table.S
/platform/RVEX/sysinit.S
```

Figure 7.2: Architecture-specific files in the include/asm-VEX directory

```
./asmmacro.h
./asm-consts.h
./atomic.h
./bitops.h
./bytworder.h
./checksum.h
./ctirngwdf.h
./delay.h
./head.h
./ioctl.h
./flo.h
./file.h
./page.h
./ptable.h
./processor.h
./ptrace.h
./resource.h
./segment.h
./semaphore.h
./sigcontext.h
./signal.h
./socket.h
./staps.h
./stat.h
./string.h
./syscallparam.h
./system.h
./termios.h
./unistd.h
./user.h
```

minor limitations because of the lack of an MMU. These limitations and more details on uCLinux are presented, followed by the porting process. The chapter concludes with
7.5. CONCLUSION

Figure 7.3: Assembly code to configure the ρ-VEX’s IPL

```assembly
extern inline void setipl(int newipl)
{
    int tmp;
    /* no shifting necessary, IPL are bottom 4 bits of VCR */
    newipl &= VCR_PIL_MASK;
    __asm__ volatile (
        "c6 mov %0 = $c9.0\n"
        "c6 mov %0 = %0, %0\n"
        "c6 or %0 = %0, %2\n"
        "c6 mov %c8.0 = %0 \n"
        "c6 mov %c8.0 = %0 \n"
        "c6 mov %c8.0 = %0 \n"
        "c6 nop\n"
        "c6 nop\n"
        "c6 nop\n"
        "c6 nop\n"
        "c6 nop\n"
        "c6 nop\n"
        "c6 nop\n"
        "c6 nop\n"
        "c6 nop\n"
        "c6 nop\n"
        "c6 nop\n"
    "=r" (tmp)
    "=r" (tmp), "=r" (newipl), "=r" (-VCR_PIL_MASK)
    
    ;
}
```

Figure 7.4: Entry code - initialing the Stack Pointer, BSS section, calling main()

file listings of our added architecture, an overview of the steps in the boot process and a number of routines and fragments from important assembly code in our implemented architecture-specific code.
#define TRAP_SIMPLE(vec) \
   cb goto vec

/* Generic trap entry. */
#define TRAP_ENTRY(type, label) \
   cb stw -(TRACEREG_S2-PT_R3)[$r0.1] = $r0.3 \
   cb mov $r0.2 = type \
   cb goto label

/* This is for traps we should NEVER get. */
#define BAD_TRAP(num) \
   cb stw -(TRACEREG_S2-PT_R3)[$r0.1] = $r0.3 \
   cb mov $r0.3 = num \
   cb goto bad_trap_handler

/* Software trap for Linux system calls. */
#define LINUX_SYSCALL_TRAP \
   cb goto linux_syscall

/* software breakpoint to halt the core (gdb) */
#define DEBUGGER_TRAP \
   cb goto trap_into_debugger

/* H save some reg on stack and put int_level there. jump to real_irq_entry */
/* Another option is to create a read/writeable trap_entry, ctrl reg ofen.  */
/* so we can read the int_level in the real_irq_function */
/* using movc instruction */
#define TRAP_ENTRY_INTERRUPT(int_level) \
   cb stw -(TRACEREG_S2-PT_R3)[$r0.1] = $r0.3 \
   cb mov $r0.3 = int_level \
   cb goto real_irq_entry

Figure 7.5: Macros used to define different types of trap vectors
This chapter will discuss the results obtained from the implementation steps described in previous chapters. According to Section 1.3, we will evaluate the hardware and software and present an overview of the development environment that has been established in this project. First, some performance measurements are presented of the hardware platform. Secondly, we will describe the level of functionality we managed to achieve for our Linux system. Then, we will present some quantitative figures and preliminary performance measurements of our port. These measurements are preliminary because there are no real applications running on the system yet (see Section 7.2.2).

8.1 Hardware platform evaluation

This section presents performance measurements of the hardware platform.

8.1.1 Core

This section shows performance measurements of the core in a standalone configuration compared to performance of the core in our hardware platform. In the standalone configuration, the program binaries were converted to VHDL and synthesized on the FPGA as memory arrays that can be accessed in a single cycle (ideal memory). This way, we can measure the impact of the memory hierarchy of the GRLIB-based hardware platform on the performance. We used a subset of the Powerstone benchmarks for evaluation. The results are presented in Table 8.1.

Table 8.1: Execution time of Powerstone benchmark programs measured in cycles. The performance without caches is purely theoretical; it is calculated by multiplying the number of memory accesses (see Table 8.2 and Table 8.3) by the bus transfer latency (see Figure 8.1).

<table>
<thead>
<tr>
<th>Program</th>
<th>Cycles (Ideal memory)</th>
<th>Cycles (without caches)</th>
<th>Cycles (GRLIB-platform)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADPCM</td>
<td>18.370</td>
<td>872.760</td>
<td>821.242</td>
</tr>
<tr>
<td>blit</td>
<td>10.332</td>
<td>454.440</td>
<td>710942</td>
</tr>
<tr>
<td>CRC</td>
<td>9.971</td>
<td>413.020</td>
<td>131.331</td>
</tr>
<tr>
<td>DES</td>
<td>45.229</td>
<td>1.985.440</td>
<td>2.145.703</td>
</tr>
</tbody>
</table>

These programs were chosen because their runtime is relatively short; simulation of the GRLIB platform running some of the benchmarks would take several hours or days to complete.
8.1.2 AMBA bus and main memory

Simulation waveforms are presented in Figure 8.1. In Simulation, an initial single bus transfer takes 15 cycles to complete, and subsequent transfers take 10 cycles. As the bus is running at 75 MHz and the transfer width is 32 bits, this leverages a sustained memory bandwidth of 240 Mbit/s (assuming continuous bus transfers). Both the DDR memory and the AHB are capable of providing more bandwidth, but the cache’s interface to the AHB does not support burst transfers (which would speed up the interface considerably).

Figure 8.1: The AMBA bus transfer delay for memory accesses is 10 cycles

8.1.3 Caches

The instruction cache reads instructions from the bus in blocks of 4 bundles when it encounters a miss. This means that the system will spend 160 cycles reading new instructions from the bus at every instruction cache miss. This also means that the cache miss figure is misleading because a single instruction cache miss will usually result in 4 bundles being loaded into the cache. Also note that this scheme of prefetching bundles might actually degrade performance, because it is possible that the other bundles in the block are not executed at all (because of a branch, or because the execution has jumped to a bundle that is not the first of the block). This can be seen in Table 8.1, where the results from the core with a perfect memory system can be compared to the theoretical performance of a system without caches. Table 8.2 and Table 8.3 show the cache miss rates of the Powerstone benchmarks presented in Table 8.1.

Table 8.2: Instruction cache miss rates

<table>
<thead>
<tr>
<th>Program</th>
<th>Icache read accesses</th>
<th>Icache read misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADPCM</td>
<td>23.296</td>
<td>4.688</td>
</tr>
<tr>
<td>blit</td>
<td>14.793</td>
<td>4.066</td>
</tr>
<tr>
<td>CRC</td>
<td>11.349</td>
<td>707</td>
</tr>
<tr>
<td>DES</td>
<td>57.277</td>
<td>12.098</td>
</tr>
</tbody>
</table>

8.1.4 Debugger Unit

When using the GRMON tool to load binaries into the platforms memory, the transfer speed is 1.53 Mbit/s as can be seen in Figure 8.2. Our modified `adv_jtag_bridge` (see Section 5.4.5) is able to transfer binaries at speeds of 1.752 bytes/s on average when using
8.2. LINUX EVALUATION

Table 8.3: Data cache miss rates

<table>
<thead>
<tr>
<th>Program</th>
<th>Dcache read accesses</th>
<th>Dcache read misses</th>
<th>Dcache write accesses</th>
<th>Dcache write misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADPCM</td>
<td>9.550</td>
<td>324</td>
<td>4.246</td>
<td>660</td>
</tr>
<tr>
<td>blit</td>
<td>2.098</td>
<td>2.016</td>
<td>2.018</td>
<td>2.014</td>
</tr>
<tr>
<td>CRC</td>
<td>868</td>
<td>86</td>
<td>550</td>
<td>548</td>
</tr>
<tr>
<td>DES</td>
<td>15.560</td>
<td>9.992</td>
<td>2.068</td>
<td>760</td>
</tr>
</tbody>
</table>

Figure 8.2: Loading a binary image to the platform using GRMON

burst transfers of approximately 16 kiB in size. This large discrepancy is probably due to the unoptimized communication with the USB driver. The speed might be improved by changing the way burst transfer are implemented. Transfer speeds are sufficient to control the core and readout CPU registers and memory sections. GRMON can still be used to load the binary into the memory, as both programs can use the same TAP controller. Therefore, no effort has been made to increase the performance yet.

8.2 Linux evaluation

This section presents figures of our port. As we have seen in Chapter 7, we have only ported the Linux kernel. For a complete, usable OS, more components will need to be ported.

8.2.1 Level of functionality

The kernel now boots up to the point where the init process (its pid is 1; it is the first new process created by the startup process) is trying to call /linuxrc, start the /etc/init program and open a shell like /bin/sh. These attempts all fail because none of these files exist yet (see Section 7.2.2). Instead of panicking [62], the init thread will fall through into its default code which is an endless loop of wait() calls.

\(^2\)The init process is the parent of all processes (directly or indirectly). It is responsible for adopting all orphaned processes and cleaning up after they terminate (otherwise these processes would become
Because Linux is unable to start a shell (see Figure 8.3), there is no possibility for the user to interact with the system. Kernel messages can be viewed on the system console, so it is possible insert debug messages to see the kernel switch between the processes that were started by the init process. Simple programs can be compiled into the kernel, and started as a kernel thread (see Figure 8.5).

8.2.2 Kernel image

This section gives an overview of the binary kernel image. Linux is very modular, so many components can be configured to be included or excluded from compilation. For example, in this project only 2 filesystem types were compiled into the kernel: ext2 and ROMfs. Figure 8.4 show the relative sizes of kernels components. The total size of the binary image is 1.704 kiB. The .text section is 1654 kiB, the .data section 44 kiB. The size of the .bss section is 102 kiB.

The image contains 423.504 syllables or 105.876 bundles. Of these syllables, 302.296 are NOPs (71,4%). 11.974 syllables contain a long immediate value (2,8%). The number of control-operations is 18.885, so 17,8% of the bundles contain a control operation as there can only be 1 control operation in a bundle. This is why the number of NOPs is so large; the code is largely control-bound (branchy). Additionally, our core needs 1 "zombies"). That is what the wait() call does.

3The BSS section contains variables and arrays that have no pre-set value, such as buffers. Therefore it does not need to be stored in the binary. One of the first steps in the boot sequence is initializing all memory in the BSS section to zeros (see Section 7.4.2).
Figure 8.4: Components of the binary Kernel image
cycle delay before most branch operations. This means that many branch operations are preceded by a syllable that only contains NOPs. The scheduler contains 16,020 syllables (4,005 bundles) of which 10,596 are NOPs (66%) and 560 bundles contain a control operation (14%).

8.2.3 Task switches

Without any user applications, the only way to test task switches is to compile routines into the kernel and use them to create kernel threads. Figure 8.5 show a running task that is interrupted by a timer tick. Before returning to the task, the timer interrupt routine checks a variable need_resched to see if it should invoke the scheduler. Normally, when need_resched is set, this will mean that its quantum has run out and another task should now receive CPU time. In this case, the scheduler selects the same task. This gives us the possibility to evaluate the performance of a task switch.

The evaluation works by having the compiled-in kernel thread read out the cycle counter register (see Section 3.4.3.2) repeatedly in an endless loop. As is clear from the figure, the tasks loop normally takes approximately 0x32700 clock cycles (approximately 200,000 cycles or 2.7 ms). After the scheduler invocation, the cycle count was 0x1FAAA3 (approximately 2,000,000 cycles or 27 ms) which means that the interrupt and scheduling code has run for approximately 24.3 ms. Note that the debug messages will add a substantial amount of cycles to these numbers. However, they must be included to demonstrate that an interrupt is triggered, the scheduler is invoked which first runs a timer Bottom Half before selecting and resuming a task.

8.2.4 Interrupt latency

In Figure 8.6 a waveform is presented of a simulation where a trap is activated that causes the core to jump to the trap_setup routine. This routine saves the context and should be called by every interrupt or trap entry point. The instruction that activates the trap signal runs after 1,254 cycles. The routine finishes at 14,096 cycles, so it takes 12,842 cycles to complete (171 us). This routine runs slowly, as the instruction cache is constantly fetching new instructions from main memory. Comparing these results to [63], where the interrupt latency of a uClinux kernel running on a MicroBlaze softcore is measured to be 5,630 cycles, it seems that the performance is reasonably well as the number of registers of the ρ-VEX is much larger than that of the Microblaze (72 vs 37). Interrupt latency could be improved considerably by limiting the registers that are saved, coordinating those registers with the interrupt handler routines (they should not use any registers that have not been saved). However, this is outside the scope of this project as interrupt performance is not a development goal.

---

4 This delay must be introduced by the compiler between writing a branch register and referencing it for a conditional branch.

5 A Bottom Half (BH), also known as a second level/slow interrupt handler, is the part of an interrupt handler that does not have to be executed immediately. Instead the upper half or fast interrupt handler flags that the BH must run, thereby deferring it to the next time the scheduler runs. The purpose of splitting the handlers is to keep interrupt handling time as short as possible.

6 This includes the 1-bit branch registers, the link register and the control registers for the VEX and the 5 special purpose registers for the Microblaze.
8.3 Development Environment

A schematic representation of the full system that has been implemented in this project is given in Figure 8.7. It also shows the development environment, that features a software development and software debugging flow. The debugging workflow has been added during this project, and the development workflow has been improved and modified to work with the hardware. Of all the components in this figure, the C standard library and Application components have not been implemented yet.

8.4 Conclusion

This chapter evaluates the platform and provides preliminary performance figures of the Linux kernel running on the platform. Only the kernel has been ported to the platform, so the OS is unable to start a shell or other userspace programs that can interface with a user. The interrupt latency of the kernel running on our platform is comparable to similar systems. The task switch latency measurements can not be used for comparisons directly because the system outputs various debug messages and can thus far only switch between kernel threads (no user processes). When ports have been created for a C standard library and a core set of user applications (see also Section 7.2.2), more tests and evaluations of the system are necessary.
Figure 8.6: Waveform of the activation of a software trap, causing the core to jump to the interrupt vector located at address 0xFE0 (the second last software trap)

Figure 8.7: Schematic representation of the full system that has been implemented in this project.
Conclusions

In Section 1.3, the problem statement and goals for this project are defined. This chapter will discuss the results and evaluate to what extent these goals have been reached. Section 9.1 Summarizes the chapters of this thesis. Subsequently, Section 9.2 presents the main contributions of this project. The chapter concludes with recommendations for future work on this subject in Section 9.3.

9.1 Summary

Chapter 2 first presents related work on softcore processors. The $\rho$-VEX is the only runtime reconfigurable VLIW core with a complete toolchain. Then, concepts related to the feasibility of the final goal as defined in Section 1.2 are discussed. Combining an OS with parametrized reconfigurability poses a number of requirements on the Platform. First, it is argued that for a system to benefit from parametrized reconfigurability, it does not need the OS code itself to be able to run on varying configurations as the final goal states that the configuration must be optimized for the tasks running on the system. We can therefore implement the operating system without having to ensure that it will run on varying hardware configurations in order to make sure that our OS will be usable for the envisioned system. In other words, even though it only supports a single hardware configuration, our OS port is an important first step towards the final goal.

Subsequently, it is argued that for a task to be able to run on varying configurations, it will need a special form of binary as it is impossible to replace a running binary by one that was compiled for a different hardware configuration. Generic binaries can be used for this purpose. The OS needs to be able to control the hardware configuration. This can be implemented by hardware registers that control the reconfiguration circuitry. The OS needs to know what configuration is optimal for its running tasks. Furthermore, every task will have different optimal configurations during different phases of its execution. Two concepts are discussed to handle this; compile-time phase analysis and using performance counters to monitor runtime execution characteristics of a task in order to evaluate the current hardware configuration. This last scheme could optimize the hardware for a task even if no phase information (and corresponding optimal configurations) is available.

In Chapter 3, an initial hardware platform based on GRLIB is selected to be used in the project. It already meets most of the system requirements. The chapter starts with a discussion of the requirements for the hardware platform. It introduces two existing designs that could be used as an initial platform for our system. The amount of work that is needed for both platforms to meet the requirements is assessed, and one of the platforms is selected accordingly. Subsequently, the key properties of this platform are described in more detail. The chapter finishes with a review of necessary additions and
modifications to meet the requirements.

Chapter 4 presents the designs of the additions and modifications that were made to the initial platform to enable it to run a general-purpose Operating System. First, a number of small but crucial fixes are presented. Then, the design of the Trap controller module for the core is introduced, which can connect to the GRLIB interrupt controller as discussed in Section 3.5. To leverage the added functionality of the new module, additions to the core and an instruction set extension were necessary. Control registers were added to the core design, together with special instructions to access them. The instructions were also added to the toolchain so that they can be used by software.

Chapter 5 presents the rationale and design of a hardware debug unit, the implementation of the \texttt{rvex} target for the GNU debugger (GDB) and the implementation of a RSP server program that can be used by GDB to debug programs running on our hardware platform by allowing it to connect to the debugging hardware. A debugging environment is important step in accomplishing our goal of establishing an effective development environment (see Section 1.3). Furthermore, the process of porting an OS will include a lot of debugging which will be sped up tremendously by having proper debugging tools. The debugging hardware and software have been designed in a short time frame and there are many improvements possible on both sides.

Chapter 6 discusses the toolchain of the \texttt{\rho-VEX} Platform in relation to our goal of porting an OS to our hardware platform. First, the toolchain is introduced and Section 6.2 discusses how it can generate binaries for different hardware configurations. Then, the development of the toolchain during the project is presented. A number of modifications were necessary to support our hardware platform and to support compiling for Linux. Furthermore, several issues were encountered and fixed during the project. In the end, the toolchain could compile a bootable Linux kernel image, thereby satisfying our goal. However, it is unlikely that all issues have been found. The toolchain is not yet perfect and more work is needed to improve it. Multiple problems were solved by a workaround in the code, instead of revising underlying problems in the way our architecture was implemented. The VEX Machine Description in GCC should be revised, and an extra pass should be added to GCC to detect inter-basic block hazards. Binutils should be updated to a newer version, and the way that relocations are implemented in the assembler and linker should be revised.

Chapter 7 discusses the process of porting an Operating System to our hardware platform that was implemented in previous chapters. We decided to use Linux for our OS; the Linux 2.0 \texttt{no_mmu} kernel to be more precise. The chapter starts with a short introduction of Linux and lists the arguments for its use in this project. Subsequently, a short description of OS components and Linux distributions is given. The uCLinux distribution and its \texttt{no_mmu} kernel allows us to use Linux on our \texttt{\rho-VEX} core with relatively minor limitations because of the lack of an MMU. These limitations and more details on uCLinux are presented, followed by the porting process. The chapter concludes with file listings of our added architecture, an overview of the steps in the boot process and a number of routines and fragments from important assembly code in our implemented architecture-specific code.

Finally, Chapter 8 evaluates the platform and provides preliminary performance figures of the Linux kernel running on the platform. Only the kernel has been ported to
the platform, so the OS is unable to start a shell or other userspace programs that can interface with a user. The interrupt latency of the kernel running on our platform is comparable to similar systems. The task switch latency measurements can not be used for comparisons directly because the system outputs various debug messages and can thus far only switch between kernel threads (no user processes). When ports have been created for a C standard library and a core set of user applications (see also Section 7.2.2), more tests and evaluations of the system are necessary.

9.2 Main contributions

In this thesis, we present the implementation of a hardware platform based on the ρ-VEX processor and the porting of the Linux 2.0 no_mmu kernel to that platform. This section presents the main contributions of this project.

In Figure 8.7, the full system including the hardware and software components as well as the toolchain is presented. Of all the components in the diagram, the C standard library and Applications have not been implemented. They will be included as future work. Implementing them during this project was impossible due to the limited time frame. It must be noted that the absence of these elements implies that the problem statement has not been resolved to its full extent as they are necessary parts of an OS (as discussed in Section 7.1.1). One project that aims to port the uClibc library to the ρ-VEX has already been started at the time of writing.

The goals (defined in Section 1.3) that were obtained from the problem statement were achieved during this project in the following ways:

- The hardware requirements are defined in Section 3.1 and implemented in Chapter 4.
- The software requirements and development process with respect to the OS including the choice to use Linux (and which variant) is presented in Chapter 7. The software requirements and development with respect to the toolchain has been presented in Chapter 6.
- By implementing the OS, considerably improving the toolchain and creating a debugging environment (presented in Chapter 5), an effective development environment has been established.

The main contributions of this project are:

- Created Vectorized Trap/Interrupt controller with 128 user (software) traps and 128 system traps or exceptions.
- Added Trap instruction to the ρ-VEX core and to the toolchain.
- Added control registers to the core and created new instructions for the core to read/write them (also adding them to the toolchain), essentially extending the Instruction Set Architecture.
- Created a hardware debug unit.
• Modified the OpenRISC1000 advanced jtag bridge program to support GRLIB’s Test Access Port.

• Added $\rho$-VEX support to GDB

• Solved numerous issues in the toolchain, writing fixes for GCC, the GNU linker and assembler.

• Performed modifications to the toolchain to support the new development environment.

• Ported the Linux 2.0 kernel to the $\rho$-VEX using a uCLinux NO-MMU distribution.

9.3 Future work

In the first chapters, many ideas have been presented about how to integrated a runtime reconfigurable core with an OS. Still, there is an enormous amount of work that needs to be done to implement the final goal that was defined in Section 1.2. This section will list a number of these tasks related to the final goal of the overall project, after presenting future work that is more directly related to the Linux port.

Future work related to our Linux port:

• Porting the C standard library to the platform.

• Integrate the C library with low-level code for handling system calls, Signals and Exceptions.

• Porting userspace applications to the platform.

• After that, thoroughly testing and further debugging of the system using real applications.

• Adding device drivers to allow the system to use more peripherals (FLASH storage, Ethernet).

• Adding the newly created VEX architecture to the latest Linux kernel version.

• If a MMU is designed for $\rho$-VEX, a normal kernel could be used that supports virtual memory.

Future work related to the envisioned system as described in Section 1.2:

• Using the runtime-reconfigurable version of the core, modifying it to allow OS-controlled reconfigurations.

• Adding Performance monitoring hardware to the $\rho$-VEX design.

• Modifying the OS scheduler to read performance metrics for the running tasks.

• Implementing algorithms that optimize the hardware configuration for the running tasks.


List of Acronyms

**ABI** Application Binary Interface.

**AHB** Advanced High-performance Bus. Bus protocol specified in AMBA.

**ALU** Arithmetic and Logical Unit.

**AMBA** Advanced Microcontroller Bus Architecture.

**APB** Advanced Peripheral Bus. Bus protocol specified in AMBA.

**ASIC** Application-Specific Integrated Circuit.

**Bare-Metal** Directly on the chip, with no OS or other supervising software.

**BH** Bottom Half. Deferred portion of an interrupt handler routine.

**BRAM** Block RAM. Special FPGA resource that can be used to instantiate memory arrays.

**Bus** Just a bunch of wires. Connects all the components to each other. Because it is a shared communications channel, some protocol is needed along with control circuitry to allow all attached devices to access it.

**CPU** Central Processor Unit.

**DDR** Dual Data Rate. A type of RAM.

**DSU** Debug Support Unit. A hardware debug unit for the Leon processor from GRLIB.

**ERA** Embedded Reconfigurable Architectures. EU-funded project based on the ρ-VEX processor core.

**FPGA** Field Programmable Gate Array. A type of reconfigurable hardware.

**GCC** GNU Compiler Collection.

**GDB** GNU Debugger.

**GNU** GNU’s Not UNIX.

**GPP** General Purpose Processor

**GR, BR, LR, CR** General Register, Branch Register, Link Register, Control Register.

**GRLIB** Gaisler Research Library. An open-sourced VHDL library.

**I/O** Input / Output. Usually a transfer of some data between main memory and a peripheral device (like a hard disk or network adapter).

**IC** Integrated Circuit.
IDE Integrated Development Environment.

ILP Instruction-level parallelism.

IRQ Interrupt ReQuest.

ISA Instruction Set Architecture.

JTAG Joint Test Action Group.

LUT LookUp Table.

MMU Memory Management Unit.

NMI Non-Maskable Interrupt.

NOP No-op, no-operation. CPU instruction that does nothing.

OS Operating System

PC Program Counter. Register containing the address of the instruction that the processor will execute next.

PIL Processor Interrupt Level.

RAM Random Access Memory.

RC Reconfigurable Computing.

RSP Remote Serial Protocol. Protocol used by GDB to debug remote systems.

SPARC Scalable Processor ARChitecture. ISA developed by SUN microsystems.

TAP Test Access Port.

USB Universal Serial Bus.

UART Universal Asynchronous Receiver/Transmitter. Peripheral device used for serial communication.

UNIX Now both a family of Operating systems and a design philosophy, the original UNIX Operating System was developed by Ken Thompson and Dennis Ritchie (who has also created the C Programming Language).

VEX VLIW Example.

VHDL VHSIC (Very High-Speed Integrated Circuit) Hardware Description Language.

VLIW Very Long Instruction Word. Type of processor architecture designed to exploit ILP without increasing hardware complexity by having the compiler resolve the dependencies.

XIP eXecution In Place. Running a program directly from its storage location (usually Flash memory) without loading it into main memory.