Interfacing the $\rho$-VEX with the $\mu$Blaze processor

M. de Zeeuw

Abstract

Developments in reconfigurable platforms result in constantly increasing available area and improving technology. These improvements allow embedded systems to implement increasingly complicated systems. As a result the performance gap of processors build on FPGA technology compared to Semi-custom ASIC technology is decreasing. The down side of these developments is the increasing complexity in designing an embedded system, and the longer development tracks that result from this complexity. In this thesis, we design and implement a flexible platform aimed for ease of use whilst offering significant speed-up compared to a General Purpose Processor based system. As a basis for our platform we have taken the MOLEN Polymorphic Processor. By combining a General Purpose Processor with custom accelerators only parts of an application have to be implemented in custom hardware to gain considerable speed-up. We aim to decrease the implementation effort further by using a VLIW processor as accelerator in the MOLEN system. Specifically the Xilinx MicroBlaze processor is used as host processor and the r-VEX VLIW processor is used as the accelerator. The r-VEX processor offers parametrization of many of its aspects, such as the issue width and logic organization. As a result of the parametrization a r-VEX processor can be implemented that meets the applications specific requirements.
Interfacing the ρ-VEX with the μBlaze processor

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

M. de Zeeuw
born in Leiden, The Netherlands
Interfacing the $\rho$-VEX with the $\mu$Blaze processor

by M. de Zeeuw

Abstract

Developments in reconfigurable platforms result in constantly increasing available area and improving technology. These improvements allow embedded systems to implement increasingly complicated systems. As a result the performance gap of processors build on FPGA technology compared to Semi-custom ASIC technology is decreasing. The down side of these developments is the increasing complexity in designing an embedded system, and the longer development tracks that result from this complexity. In this thesis, we design and implement a flexible platform aimed for ease of use whilst offering significant speed-up compared to a General Purpose Processor based system. As a basis for our platform we have taken the MOLEN Polymorphic Processor. By combining a General Purpose Processor with custom accelerators only parts of an application have to be implemented in custom hardware to gain considerable speed-up. We aim to decrease the implementation effort further by using a VLIW processor as accelerator in the MOLEN system. Specifically the Xilinx MicroBlaze processor is used as host processor and the $\rho$-VEX VLIW processor is used as the accelerator. The $\rho$-VEX processor offers parametrization of many of its aspects, such as the issue width and logic organization. As a result of the parametrization a $\rho$-VEX processor can be implemented that meets the applications specific requirements.

Laboratory : Computer Engineering
Codenummer : CE-MS-2011-01

Committee Members :

Advisor: Dr. ir. J.S.S.M Wong, CE, TU Delft
Chairperson: Dr. ir. K. Bertels, CE, TU Delft
Member: Dr. ir. A. van Genderen, CE, TU Delft
Member: Dr. phil. H.-G. Gross, ST, TU Delft
I dedicate this thesis to my mother, father and girlfriend, for giving me unconditional love and support.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>xi</td>
</tr>
</tbody>
</table>

1 Introduction 1

1.1 Motivation 1
1.2 Project Description 2
1.3 Thesis organization 2

2 Background 3

2.1 MOLEN Polymorphic Processor 3
2.1.1 MOLEN instructions 4
2.1.2 MOLEN implementation 5
2.2 \( \rho \)-VEX 6
2.3 \( \mu \)Blaze processor 7
2.3.1 Local Memory Bus 7
2.3.2 Processor Local Bus 9
2.3.3 On-Chip Peripheral Bus 9
2.3.4 Fast Simplex Link 9
2.3.5 AXI4 9
2.4 hArtes Toolchain 11
2.4.1 hArtes toolboxes 12
2.4.2 GCC Compiler 14

3 Compiler Design 15

3.1 MOLEN implementations 15
3.2 hArtes GCC compiler 16
3.2.1 Compiler front-end design 18
3.2.2 Middle-end design 18
3.3 hArtes configuration file 19
3.4 MOLEN function calls 21
3.4.1 MOLEN Set function 21
3.4.2 MOLEN MovTX function 22
3.4.3 MOLEN MovFX function 22
3.4.4 MOLEN Execute function 22
3.4.5 MOLEN Break function 23
3.5 Conclusion 23
List of Figures

2.1 The organization of the MOLEN polymorphic processor [6] ............... 3
2.2 The organization of a 4-issue ρ-VEX processor .......................... 6
2.3 μBlaze processor overview [22] ............................................. 8
2.4 Typical usage of the LMB in a μBlaze system [19] ....................... 8
2.5 IBM CoreConnect bus overview [8] ........................................ 9
2.6 A PLB bus write transaction [11] ......................................... 10
2.7 An OPB read transaction [9] ............................................... 11
2.8 Block diagram of the Fast Simplex Link Bus [21] ....................... 11
2.9 Overview of the hArtes toolboxes [7] ..................................... 13
3.1 hArtes GCC compiler data flow ............................................. 17
4.1 Overview of the hardware ..................................................... 25
4.2 The PLB bus interface ......................................................... 28
4.3 The basic wrapper design ..................................................... 29
4.4 Control register organization ............................................... 29
4.5 Status register organization ................................................. 29
4.6 The control and status signals defined in the ρ-VEX interface ........ 31
4.7 A state machine, used to create the control signals for the ρ-VEX processor 31
4.8 The states used in the original implementation of the ρ-VEX core .... 32
4.9 Our new state machine, creating the correct run signals .............. 33
4.10 Overview of the interrupt system ......................................... 34
4.11 Memory mapping options, a) Memory is accessed through special memory registers inside the wrapper. b) The whole memory is directly mapped on the bus .................................................. 35
4.12 Memory interface options, a) Memory control logic inside the wrapper. b) Memory control logic inside the ρ-VEX core ................. 36
4.13 Memory signals connecting the PLB and the wrapper ................. 37
4.14 Memory signals connecting the wrapper and the ρ-VEX core ........ 38
4.15 Overview of the system organization ..................................... 39
4.16 Data memory overview ....................................................... 40
4.17 Instruction memory overview .............................................. 41
4.18 Instruction address decoding in a 4-issue ρ-VEX core ................. 41
4.19 Overview of the status memory .......................................... 42
5.1 Data flow in the Front-End of the compiler .............................. 45
5.2 Middle-End data flow FSM .................................................. 47
5.3 Data Memory interface ....................................................... 52
5.4 Instruction Memory interface ............................................... 53
6.1 Cycles required for the different actions .................................. 66
6.2 Cycles required for the different scenarios ................................ 68
# List of Tables

2.1 Channels used in the AXI4 Bus architecture .......................... 12  
2.2 MOLEN functions .................................................... 14  
4.1 Truth table for the idle signal ...................................... 33  
5.1 Build dependencies .................................................. 44  
5.2 Build scripts .......................................................... 44  
5.3 PLB transaction macros .............................................. 54  
6.1 System configuration .................................................. 63  
6.2 Resource utilization for our platform ............................... 64  
6.3 Transfer delays to read and write from a $\rho$-VEX memory ..... 65  
6.4 Cycle times obtained with the MindTCT fingerprint application 69  
A.1 $\mu$Blaze configuration ............................................... 79  
A.2 $\mu$Blaze configuration ............................................... 80
First of all, I want to thank my advisor, Stephan Wong, for the excellent guidance I received during my thesis. I want to express my gratitude for his patience, confidence and continuous support which have greatly impacted my thesis work.

Secondly, I want to thank Fakhar Anjam for advising me while I performed my project. Our discussions were very beneficial and he was always ready to assist me.

I want to thank Roel Seedorf for his continues guidance and help. There was never a moment I could not contact you when I ran into a problem, even during your holiday. We had some big discussions and brainstorm sessions which were a big impact on my thesis.

Furthermore, a big thanks to all my family, friends and my girlfriend Ciska Hupkes, for all the patience, support and love.

M. de Zeeuw
Delft, The Netherlands
August 16, 2011
Introduction

The current trends towards multimedia, 3D visualization and connectivity place increasingly higher demands on computing systems. In embedded systems multi-core architectures are widespread in attempt to address the memory, frequency and power bottlenecks created by these demands. With the increasing capabilities of embedded systems the complexity of mapping an application to such a system continues to grow. This thesis present the integration of embedded soft-core processors into a hardware-software co-design solution. In Section 1.1 discusses the motivation for this thesis. Subsequently, Section 1.2 describes the project and gives an overview of the conducted research. Section 1.3 concludes this chapter by presenting the framework of this thesis.

1.1 Motivation

Following Moore’s Law every 18 months the number of transistors on a chip doubles. This rapid increase of available transistors has resulted in ever more capable Field Programmable Gate Array’s (FPGA), which have diminished most of the classical drawbacks. With the combination of many benefits from both the software and Application-Specific Integrated Circuit (ASIC) implementations, reconfigurable computing is able to accelerate many software systems significantly [13]. Where in the past an ASIC implementation had to be designed, now a reconfigurable platform based design could be up to the task.

As the complexity and available logic on FPGA’s rises so does the development complexity of applications target for these platforms. Recently progress has been made to decrease application development complexity whilst fully utilizing the available hardware. These systems often focus on hardware-software co-design. With the availability of a General Purpose Processor (GPP) only subsets of the application have to be identified and mapped to processing elements in order to gain considerable speed-up. The MOLEN Polymorphic processor provides such a system [6, 15, 14]. It combines a GPP with an application specific custom hardware on the reconfigurable fabric of a FPGA. The processor takes care of the general-purpose (control) calculations, whilst the custom hardware executes application specific logic. These two processes can run concurrently and combined with the available speed-up in the custom hardware, a speed-up larger then 3 has been achieved. In addition to the speed-up the development time is decreased, since only a subsection of the application needs to be implemented in custom hardware.

Very Long Instruction Word (VLIW) processors can efficiently execute applications containing enough Instruction Level Parallelism (ILP). Recent work done by Thijs van As and Roël Seedorf has resulted in a flexible VLIW processor targeted at FPGA fabrics [17, 16]. This VEX processor offers a reconfigurable and flexible platform for applications. The drawback of the VLIW processor is its low efficiency when running low ILP
CHAPTER 1. INTRODUCTION

applications. This is caused by the cache usage and inactive logic inside the processor. The parametric architecture of the $\rho$-VEX processor is able to deal with this drawback for many applications. Furthermore, by using the VEX processor as an accelerator inside the MOLEN platform, this disadvantage of VLIW processors can be circumvented completely. Only parts of an application containing enough ILP will be executed on the VLIW processor, the rest will be run on the GPP.

1.2 Project Description

The main goal of this project is to design and implement a platform that interfaces the $\rho$-VEX processor with the $\mu$Blaze processor. By utilizing the MOLEN paradigm this platform offers ease of use and flexibility. The platform combines the $\mu$Blaze GPP processor and the $\rho$-VEX VLIW processor to benefit from the processors specific benefits. Specifically, only utilizing the $\rho$-VEX processor during sections of the application containing enough ILP. This requires selecting and implementing the communication interface between the two processors. Furthermore, to create a workable system we develop a customized compiler that will integrate the MOLEN functionality into the $\mu$Blaze processor. The project has been split in multiple stages.

1. Create the software toolchain. The first step we take is to implement the required toolchain. This involves adopting the $\mu$Blaze GCC compiler to incorporate the MOLEN features.

2. Design and implement the $\rho$-VEX bus wrapper. Using this wrapper a $\rho$-VEX core can be connected to a communication bus which would then be able to control the $\rho$-VEX processor.

3. Combine the toolchain and the wrapper to get a integrated system that allows using a $\rho$-VEX processor as a accelerator within a MOLEN polymorphic processor. This involves implementing the MOLEN functions on the $\mu$Blaze processor.

1.3 Thesis organization

The remainder of this thesis is organized as follows. We start in Chapter 2 of by giving background information of the different components used in the thesis. Chapter 3 present the design of our compiler. This compiler targets the $\mu$Blaze processor and will have integrated support for the MOLEN polymorphic processor. In Chapter 4 the bus wrapper is designed. This wrapper will connect the $\rho$-VEX core to the bus, thus allowing the $\mu$Blaze and $\rho$-VEX to communicate. Following these design chapters the implementation of the system is detailed in Chapter 5. Subsequently, Chapter 6 discusses the experimental setup and results obtained with our system. Chapter 7 a summary of the thesis, our conclusions, and an elaboration on the future work, related work and recommendations is presented.
In this section we discuss the different concepts and systems used throughout this thesis. Particular attention is given to the MOLEN Polymorphic processor and the ρ-VEX processor as these are core elements of our system. First, in Section 2.1 we present the MOLEN Polymorphic Processor. Subsequently, Section 2.2 describes the features of the ρ-VEX parametrizable VLIW processor. Section 2.3 discusses the μBlaze processor. Finally, in Section 2.4 the background on the hArtes toolchain is presented.

2.1 MOLEN Polymorphic Processor

A major concern of general-purpose processors (GPP) is performance. A good candidate to address the performance concern is reconfigurable hardware that coexists with the GPP. The MOLEN Polymorphic Processor makes this combination to create a hybrid processor [6].

By a one-time extension of the GPP Instruction Set Architecture (ISA) an almost arbitrary number of reconfigurable functions can be provided. The newly introduced instructions are used to control and communicate with the Reconfigurable Processor (RP) implemented on reconfigurable hardware. During execution all instructions are processed
by an Arbiter and forwarded to either the GPP or the RP. The arbiter will send all regular instructions to the GPP, but will filter out the MOLEN instructions. These MOLEN instructions are then decoded and forwarded to either the RP or the GPP, depending on the instruction. An overview of the MOLEN organization is shown in figure [2.1]. The reconfigurable processor is subdivided into a $\mu$-code unit and a Custom Computing Unit (CCU). The $\mu$-code unit controls the CCUs present on the Field Programmable Gate Array (FPGA) and ensures the correct configuration is applied. The CCU contains the application specific custom hardware designed to speed-up selected aspects of the application. These CCUs are the only hardware that needs to be customized in a project. All remainder code of an application will be executed unaltered on the GPP. Communication between the GPP and CCUs is done using exchange registers (XREGS). The main memory is also available for both the GPP and RP, so large data sets do not need to be transferred using the exchange registers.

2.1.1 MOLEN instructions

With the inclusion of the new MOLEN instructions a new ISA is created called $\pi$ISA. The MOLEN instructions contained in the $\pi$ISA are required for controlling the reconfigurable hardware.

- **Set** instruction is used to initiate the configuration of a CCU on the reconfigurable hardware. It is divided into two instructions *partial set* and *complete set*. The partial set (p-set) instruction handles configuration of the common parts of functions and frequently used functions. The remaining blocks are configured with the complete set (c-set) instruction. If no partial reconfiguration is used then the c-set instruction alone can be utilized to perform all configurations.

- **Execute** is used to trigger the start of execution of a operation on the configured CCU.

- **Set prefetch** this instruction triggers the $\mu$-code to pre-fetch microcode responsible for CCU reconfiguration into local on-chip storage. This is an optimization instruction, which allows hiding the memory delay when loading the CCU configuration from memory.

- **Execute prefetch** has the same functionality as the *set prefetch* instruction, only now it pre-fetches execution related microcode from memory.

- **Break**. This instruction facilitates parallel execution of both the reconfigurable processor and the general purpose processor. It halts the GPP until the RP has finished execution thus offering a synchronization mechanism.

- **MovTX** is the write interface for the GPP. It moves content from a general-purpose register to an exchange register.

- **MovFX** is the counterpart of the movtx instruction. This instruction reads content from an exchange register and moves it into a general-purpose register.
2.1. MOLEN POLYMORPHIC PROCESSOR

The instructions in $\pi$ISA can be divided into three extension sets, depending on the required performance and available technology.

- **The minimal $\pi$ISA** is the smallest set of MOLEN instructions needed for a working scenario. The four instructions in this set are execute, c-set, movtx and movfx. The set and execute instructions allow any suitable CCU to be loaded and executed. The movtx and movfx instructions are required to provide an input/output interface between the GPP and RP.

- **The preferred $\pi$ISA** extends the minimal $\pi$ISA with instructions addressing reconfiguration latencies. Both $p$-set and $c$-set instructions are utilized to allow partial reconfiguration. In addition to the set instructions also the prefetch instructions set prefetch and execute prefetch are used. These provide a way to diminish the microcode loading time.

- **The complete $\pi$ISA** contains all $\pi$ISA instructions. The addition of the break instruction allow synchronization of the GPP and RP. Thus the GPP and RP can work in parallel. The other two $\pi$ISA sets do not offer this synchronization, they halt the next GPP instruction until the execution of the CCUs is done.

2.1.2 MOLEN implementation

Currently the MOLEN polymorphic processor is implemented on the Xilinx Virtex4 and VirtexII pro platforms. A on-chip PowerPC processor is used as the Core Processor and the exchange registers are connected using the device control register bus. A disadvantage of this platform is the limited area available for the CCUs. This has limited the possible optimizations in applications. To implement an application as a MOLEN machine the following process is used.

1. Determine sections in the application source code that can be isolated and optimized on CCU hardware. This selection can be done using profiling tools as offered in the Delft Workbench.

2. Each of the code sections found in the first step has to be evaluated for suitability of implementing in hardware. The gained speed-up and the effort it takes to map the code on hardware are important factors in this evaluation.

3. All selected code sections are implemented in a Hardware Description Language (HDL). This can be done automatically or manually.

4. Calls in the source code to the hardware mapped functions should be replaced with calls to the hardware units. The data exchange between the GPP and the hardware units have to be convert to using the exchange registers.

5. The hardware units have to be configured on the reconfigurable fabric during execution of the application.
2.2 $\rho$-VEX

The $\rho$-VEX is a Very Long Instruction Word (VLIW) processor originally created by Thijs van As [17]. The processor is based on the VLIW Example (VEX) ISA introduced in [12]. The VEX ISA offers a scalable architecture for embedded VLIW processors. Many aspects of the architecture can be customized, such as the instruction issue width and the organization of the Functional Units. It is a Load/Store, four stage Harvard architecture and the basic instructions are equivalent to the Reduced Instruction Set Computer (RISC) instructions. After the initial design Roel Seedorf greatly improved the processor core and toolchain [16]. The multi-cycle design was changed into a new pipelined $\rho$-VEX core. Additionally a new toolchain containing a GCC based C compiler and linker were created.

The $\rho$-VEX is split into four stages. Fetch, decode, execute and writeback. These four stages each occupy a pipeline stage. With exception of the execute stage which is split into two pipeline stages, an overview of the $\rho$-VEX organization is shown in figure 2.2. The five pipeline stages are configured as follows.

- The first stage fetch reads an instruction from the instruction memory. The instruction is send to the next stage for decoding. Additionally it also selects the address for the next instruction. Usually the address will simply be the next instruction, except when a branch occurs. If the Branch Unit in the decode stage finds a branch this is presented to the fetch stage as a potential branch target.
- Next comes the decode stage. This stage splits the instruction into the syllables
for each issue of the processor. These syllables are then decoded in parallel. The values from the Register File are loaded and the correct immediate values are generated. If a conditional branch is detected the branch flag is read and passed through to the Branch Unit. Finally all operands and control signals are send to the next stage.

- The third and forth pipeline stages are the two execute stages. The execute stage is split into two pipeline stages, mainly to split the 32 x 32 bit multiplication into two phases.

- The final stage of the pipeline is the writeback stage. Results from the execute stage are received by the WriteBack and committed to the destination general-purpose registers.

With the introduction of the pipeline stages the ρ-VEX is able to run at a clock frequency of 100 MHz on a Xilinx Virtex6 board. This has resulted in a speed-up of more then 3 compared to the non-pipelined ρ-VEX core.

Although originally designed to be used as a MOLEN accelerator, the ρ-VEX currently works as a stand-alone processor. The only external communications are an incoming reset signal and an outgoing UART signal. This UART is used to transmit the contents of the data memory after execution over a serial connection to a connect computer. This data can then be used to validate the correct execution of the ρ-VEX program.

### 2.3 μBlaze processor

The μBlaze processor embedded processor is a soft core processor developed by Xilinx [22]. It uses a reduced instruction set computer (RISC) and is available for the Xilinx FGPAs. The μBlaze core is organized as a Harvard architecture with separate bus interfaces for accessing data and instruction. An overview of the μBlaze core is shown in figure 2.3. Many aspects of the μBlaze can be user configured such as: cache size, embedded peripherals, memory management unit, and bus-interfaces. This customization lets developers make tradeoffs appropriate to the specific hardware and software requirements. μBlaze offers support for hardware-based paging and protection of the memory, with use of the memory management unit. This allows the μBlaze to host operating systems such as the Linux Kernel.

The Memory Management Unit of the μBlaze is capable of using on-chip block RAM (BRAM), DDR memory and caches for the instruction and data memories. On the μBlaze I/O addressing is done using the memory-mapped scheme. Multiple types of interfaces are available for use on the μBlaze.

#### 2.3.1 Local Memory Bus

The Local Memory Bus (LMB) [19] is a synchronous bus used primarily to access on-chip Block RAM (BRAM). It offers single clock cycle access to the local BRAM. This has been obtained by using a simple protocol and a minimum number of control signals. Only
one master is supported and no arbiter is used. By default two LMB busses are used on the µBlaze as shown in Figure 2.4. One each for the data and instruction BRAM.

Figure 2.3: µBlaze processor overview [22]

Figure 2.4: Typical usage of the LMB in a µBlaze system [19]
2.3. µBLAZE PROCESSOR

2.3.2 Processor Local Bus

Part of the IBM CoreConnect architecture \[8\] the Processor Local Bus (PLB) is developed as an on-chip bus-communications link \[11\]. An overview of the CoreConnect architecture is shown in Figure 2.5. The PLB is used to connect fast components to the CPU. The bus can be configured with widths of 32, 64 or 128 bits and supports burst transactions. A maximum of 16 masters and any number of slaves can be connected to a single bus. When multiple masters are used an arbiter is used to grant the masters access to the bus. Each slave checks if its address is currently on the bus.

![Figure 2.5: IBM CoreConnect bus overview \[8\]](image)

2.3.3 On-Chip Peripheral Bus

A second bus defined by the CoreConnect architecture is the On-Chip Peripheral Bus (OPB). As the name indicates this bus is focused to interface slow peripherals. It is non-split and does not have burst support. A maximum of 16 masters and 16 slaves can be connected. The bus transaction is done in a single synchronous handshake.

2.3.4 Fast Simplex Link

The Fast Simplex Link (FSL) \[21\] is a point-to-point uni-directional FIFO based bus. The bus has a single master and slave, the master pushes data onto the bus and the slave reads this data from the bus, a block diagram is shown in Figure 2.6. The µBlaze processor can contain a maximum of 16 FSL masters and 16 slaves. A simple handshake protocol is used where each transfer takes a single clock cycle.

2.3.5 AXI4

The latest version of the µBlaze processor, version 7, has support for the Advanced eXtensible Interface (AXI) protocol. This is a burst-based bus protocol where the address
and control information is issued separately from the actual data transfer \[23\]. The AXI bus uses five different channels, shown in Table \[2.3.5\]. It has support for multiple outstanding transactions which can be completed out-of-order.
2.4. **HARTE TOOLCHAIN**

The Holistic Approach for Real Time Embedded Systems (hArtes) project main objective was the development of a single common toolchain for the development of products consisting of heterogeneous reconfigurable embedded platforms [7]. This has resulted in a
toolchain and methodology that provides a fast development trajectory from application development to the design of such a complex system. At the core of this project lies the MOLEN polymorphic processor. Extra tools have been added to the MOLEN platform and changes have been made to increase adaptability of the system.

2.4.1 hArtes toolboxes

Three toolboxes have been created for the hArtes toolchain. Each containing one or more software tools that aid in a specific phase of the toolchain. A total of three toolboxes are defined.

- **Algorithm Exploration and Translation** (AET). This is the first phase of the toolchain and contains tools for developing and refining algorithms at a higher abstraction level. This phase is optional as developers can write C code and use it as an entry point for the toolchain.

  Two high level abstraction tools have been used in the AET. The first is Leaff’s NU-Tech graphical software development tool. The NU-Tech software offers a graphical interface allowing application design by connecting logic blocks. In addition to the design the tool can also simulate the applications. NU-Tech will generate C code which will be used by the next level in the toolchain.

  Additionally the Scilab tool can be used. This open-source MatLab variation can convert matlab code into corresponding C code. These tools will provide translation into an unified internal description, based on the C language and a XML annotations file.

- **Design Space Exploration** (DSE). Once the internal description files are generated the applications tasks are automatically or manually partitioned and mapped to the most appropriate hardware components (GPP, DSP or FPGA) of the heterogeneous system. Two tools are used one for the partitioning and one for the mapping. Zebu is the partitioning tool which is part of the PandA framework by Politecnico di Milano. It decomposes the application into tasks to improve performance. The mapping is done by the hArmonic tool developed by Imperial College London. It selects parts of the C application that will be executed on specialized processors.

- **System Synthesis** (SysSyn). The final level in the toolbox contains the back-end C compilers for the different targets (GPP, DSP and FPGA). The mapping soft-
ware has generated separate C code for each compiler in the toolbox. Each source is compiled separately and then linked together into a single hArtes runtime. The GPP compiler is based on GCC 4.3 and generates MOLEN instructions for annotated functions. The MOLEN instructions are implemented as runtime library function calls. It also does basic scheduling of the MOLEN execute and break calls. The Delft Workbench Automated Reconfigurable VHDL Generator (Dwarv) generates VHDL code from the C code. It exploits available parallelism of algorithms and generates designs suitable for the MOLEN paradigm. Finally the hArtes linker combines the results of the compilers and generates a single executable. This result contains executable and bitstreams for the different components used on the platform. It is based on the GNU Linker.
2.4.2 GCC Compiler

As part of the SysSyn toolbox the GNU Compiler Collection (GCC) compiler [2] has been extended to implement the MOLEN instructions. The compiler uses a system description input file formatted using the XML format. This file contains a full description of the available hardware components and the applications that can be run on these components. In addition to the system description input file a pragma statement is implemented to specify which parts of the source program needs to be run on a specific part of the system. Using the xml file and the pragmas the compiler will replace functions into MOLEN function calls, listed in table 2.2. The minimal MOLEN instruction set has been implemented extended with the break instruction. This offers a non-reconfigurable set-up with synchronization capabilities.

The biggest change from the original MOLEN implementation is that the instruction set architecture of the GPP no longer needs to be extended with the MOLEN instructions. Instead corresponding functions are used which implement the required functionality. This introduces a penalty since function calls are relatively slow, but makes the system much easier to implement on different processors. The benefits are gained by the removal of the Arbiter and \( \rho\mu \)-code hardware units. Both these are implemented in software and integrated into the MOLEN functions. With the removal of the Arbiter which resides in between the memory and the GPP possible latencies and restrictions related to caching are removed.
In order to interface the µBlaze processor with the ρ-VEX processor two main tasks are defined. These tasks focus on the two domains used in our system, software and hardware. The software side of the implementation involves incorporating the MOLEN functionality into the µBlaze GCC compiler. Secondly the hardware related domain focuses on the interface between the µBlaze and ρ-VEX processors. This section focuses on the software toolchain of our system. The software design consists of MOLEN integration into the µBlaze GCC compiler and implementation of the MOLEN function calls. We do not need to make changes to the ρ-VEX compiler since all MOLEN logic is handled by the host processor.

This chapter is structured as follows. First we will discuss which MOLEN implementation we will use in our toolchain in Section 3.1. This is followed in Section 3.2 with the design of hArtes functionality into our compiler. An architecture description file is used by the hArtes compiler, this is discussed in section 3.3. In Section 3.4, we discuss the implementation of the MOLEN functions into the compiler. Finally, the chapter is concluded in Section 3.5.

3.1 MOLEN implementations

Currently there are two generations of the MOLEN implementation available. The original platform is based around the PowerPC processor [6]. An arbiter filters all instructions before they are sent to the PowerPC. If a MOLEN instruction is issued the Arbiter will halt the PowerPC and apply the required action associated with the MOLEN instruction. The instructions can be executed on the PowerPC in case of data transfer instructions. Other actions will be executed by a Reconfigurable Processor. This processor has a ρµ-code block that will configure the correct accelerator logic and is able to start and synchronize the accelerators.

Recent work done for the hArtes project has resulted in a second implementation of the MOLEN polymorphic processor [7]. The hArtes project offers a complete toolchain which is build on the MOLEN paradigm. The platform is build around an ARM processor and uses the GCC compiler. In order to make the system more adaptable they have developed an alternative route to apply the MOLEN functionality in the processor. They choose to remove dedicated MOLEN control hardware and instead implement the logic in software. As a result no Arbiter and ρµ-code unit have to be designed for the target platform. This has been done by implementing the MOLEN instructions as function calls. The hArtes compiler generates different function calls each corresponding to the original MOLEN instructions.

By implementing the MOLEN instructions as function calls instead of actual instructions no changes to the Instruction Set Architecture (ISA) have to be made. A
disadvantage of this approach is that execution of a function call is slower than executing an instruction, this will introduce a small performance penalty. On the other hand the hArtes implementation offers many advantages. Many aspects of the system can easily be changed with only a small impact on the implementation as a whole. Moving the implementation to a different hardware platform would only require modifying the compiler and changing the communication method can be done completely in the MOLEN functions.

For our project the advantages of the hArtes MOLEN platform are very beneficial. The ρ-VEX processor is still under heavy development and the flexibility of hArtes would allow future ρ-VEX processors to be easily incorporated. This has resulted in our decision to use the hArtes implementation as the starting point in our project.

## 3.2 hArtes GCC compiler

The first step in implementing the hArtes functions is integrating the required functionality into the existing host compiler. Our host, the µBlaze processor, uses the GCC compiler so we have integrated the hArtes functionality in this compiler. Tools or software designers will annotate the source code of the application with MOLEN specific pragma statements. These pragma’s define which functions are executed on the Reconfigurable Processor. It is the task of the compiler to detect these annotations and replace the code with the correct MOLEN functions required to execute the code on an accelerator.

```c
/* Define a function that is always executed on a rVEX core */
#pragma map call_hw RVEX 1
void test_1(a, b) {
    ...
}

/* Define a normal function, executed on the MicroBlaze */
int test_2(a, b) {
    ...
}

/* Main body of the application */
void main() {
    /* Executed on rVEX core 1 */
    test_1( 1, 2);

    /* Executed on the MicroBlaze */
    r = test_2( 3, 4);

    /* Executed on rVEX core 1 */
#pragma map call_hw RVEX 1
    r = test_2( 5, 6);

    /* Overrule the default configuration. */
    /* Executed on rVEX core 2 */
```
3.2. HARTES GCC COMPILER

Listing 3.1: Source C code that will cause a function call to be executed on a ρ-VEX core

 GCC compilers are divided into three stages [18]. The front-end reads the source code and parses this into an Abstract Syntax Tree (AST). This is then passed to the middle-end of the compiler. In this second stage the AST is gradually transformed towards its final form. Optimizations are applied to the code and the code is analysed. Finally architecture specific assembly code is produced, this is called the back-end.

To apply the hArtes functions changes have to be made mainly to the front-end and back-end of the compiler. The code flow of the compiler is shown in figure 3.1. First the back-end has to implement the correct parsing of the source code annotations. This is performed by registering the MOLEN pragma and parsing the source code when the pragma is invoked. The front-end replaces the original source code following the MOLEN annotation with a new tree representation of the MOLEN function. In the middle-end these MOLEN trees are parsed into the required MOLEN calls.

Figure 3.1: hArtes GCC compiler data flow
3.2.1 Compiler front-end design

Once the compiler reads the source code and the MOLEN pragma ‘map call_hw’ is detected the hArtes function that parses the pragma is invoked. This function will check the pragma statement and the code it precedes. Requirements are the correct arguments for the pragma statement and that the code it precedes is either a function declaration or a function call. The pragmas should be written in the following format:

```
#pragma map call_hw TARGET TARGET_ID
```

Listing 3.2: hArtes pragma format

The map is the identifier for the MOLEN pragmas. Call_hw indicates to the MOLEN parser the following call should be executed on an accelerator. The target specifies which type of accelerator is used. In our system this can only be VIRTEX4 or RVEX, both indicate the FPGA fabric. The target id specifies which target should be used of the specified type. This allows multiple accelerators to be initiated on the same fabric, or multiple instances of the same fabric to be used. In our implementation this is used to indicate which $\rho$-VEX core should execute the code.

There are multiple ways to invoke an accelerator in the source code, as shown in Listing 3.1. If a function declaration is preceded with the MOLEN pragma this will redirect all calls to this function to the corresponding accelerator. If a function call is preceded by a MOLEN pragma only this function call will be redirected to the corresponding MOLEN accelerator. Additionally a combination can be used, where the general function declaration annotation is overruled by an annotated function call.

Parsing of the pragmas is finalized by replacing the original call or declaration with a new dummy call. This dummy will have the original tree for the call or declaration as an argument. These dummy trees will be converted to the different MOLEN function calls in a later pass executed in the middle-end of the compiler.

3.2.2 Middle-end design

In the middle-end of the compiler an extra pass is introduced that converts the dummy MOLEN trees into the MOLEN function calls. Once a call to the MOLEN dummy function is found this call is removed from the output. The original call will then be parsed so the correct accelerator will be set-up and executed. It will load the hArtes configuration file and read the correct settings for the specified accelerator. Using the configuration data the implementation is validated and some settings are retrieved. These settings are the control and instruction memory addresses, as well as the input and output addresses used in the data memory.

Once the data is validated and the addresses of the $\rho$-VEX core are loaded the MOLEN function calls are inserted. First the call to set the accelerator is called. This call will take care of initializing the $\rho$-VEX core with the correct instruction memory. Following the set are the move instructions. Each argument for the accelerator is send to the data memory using a movtx function call. Once all arguments are moved to the $\rho$-VEX core the execute call is created. This will make the $\rho$-VEX start executing its
3.3. HARTES CONFIGURATION FILE

program. The final MOLEN function that is inserted is the break instruction. This function will halt the host processor until the ρ-VEX has finished execution.

After all MOLEN functions have been created and inserted into the source tree no more actions are required. The result of the hArtes Middle-end is a regular tree that the back-end can parse to the language specific assembly code.

```
addik r5,r0,1    # rvex id
addik r6,r0,test_2__fpga_text__ # program id
brlid r15,molen_SET

addik r5,r0,1    # rvex id
addik r6,r0,167772160 # register address
addik r7,r0,3    # value
brlid r15,molen_MOVTXval

addik r5,r0,1    # rvex id
addik r6,r0,167772161 # register address
addk r7,r22,r0    # source address
addk r8,r0,r0    # size
addk r9,r5,r0    # copy
brlid r15,molen_MOVTXaddr

addik r5,r0,1    # rvex id
brlid r15,molen_EXECUTE

addik r5,r0,1    # rvex id
brlid r15,molen_BREAK

addik r5,r0,1    # rvex id
addik r6,r0,167772161 # register address
addk r7,r22,r0    # target address
addk r8,r0,r0    # size
brlid r15,molen_MOVFXaddr

addik r5,r0,1    # rvex id
addik r6,r0,167772176 # register address
addik r7,r1,36    # target address
brlid r15,molen_MOVFX
```

Listing 3.3: Example assembly code created by the hArtes compiler

Shown in Listing 3.3 is example assembly code calling the various MOLEN functions on a µBlaze target. The add(i)k instructions place the arguments for the functions into the correct registers. By invoking the brlid instruction the processor branches to the specified function. A full description of the MOLEN functions and their properties is given in Section 3.4.

3.3 hArtes configuration file

The hArtes compiler requires an input configuration file to configure the available accelerators in the system. This architecture description file is organized using the XML
standard. Three basic elements are used in the configuration functional components, storage components, bus components and operations.

Functional components represent the hardware components available in the system. Originally these components could be the GPP and FPGA. Basic information of these components is given, like the size and speed. The storage components are available memory elements. Memory size and type are given for each of the memory blocks. A bus component is used to represent the links between the functional components and the storage components. Information about the bandwidth, address width and read cycles is configured here. Finally the operations represent the operations available to be executed on each of the functional components. These are the accelerators that can called from the host processor.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE ORGANIZATION SYSTEM "architecture.dtd">
<ORGANIZATION>
  <HARDWARE>
    <FUNCTIONAL_COMPONENT>
      <NAME>RVEX4</NAME>
      <IDGEN>4000</IDGEN>
      <TYPE>FPGA</TYPE>
      <MASTER>NO</MASTER>
      <FREQUENCY>100</FREQUENCY>
    </FUNCTIONAL_COMPONENT>
    <STORAGE_COMPONENT>
      <NAME>FPGA_MEM</NAME>
      <TYPE>SDRAM</TYPE>
      <SIZE>64</SIZE>
      <START_ADDRESS>0x0</START_ADDRESS>
      <END_ADDRESS>0xffff</END_ADDRESS>
    </STORAGE_COMPONENT>
    <BUS_COMPONENT>
      <NAME>RVEXBUS</NAME>
      <TYPE>INTERNAL</TYPE>
      <BANDWIDTH>102400</BANDWIDTH>
    </BUS_COMPONENT>
  </HARDWARE>
  <OPERATIONS>
    <OPERATION>
      <NAME>rvex_dummy</NAME>
      <COMPONENT>
        <NAME>RVEX4</NAME>
        <IMPLEMENTATION>
```
3.4 MOLEN function calls

Instead of utilizing an Arbiter and \( \mu \)-code unit to control the MOLEN functionality we have used the hArtes implementation using MOLEN functions. The hArtes compiler will create calls to these functions, so a platform specific implementation of the functions has to be created. Each MOLEN instruction is implemented using a corresponding function with a small extension to allow the usage of pointers.

3.4.1 MOLEN Set function

Changing what accelerator is available and initializing this accelerator is done using the *Molen Set* instruction. In our system we do not change the hardware with this set instruction, but instead change the instruction memory of a specific \( \rho \)-VEX processor. This will change the functionality of that \( \rho \)-VEX core and will allow running different applications on a single \( \rho \)-VEX core. The set function has two arguments, first the id of the \( \rho \)-VEX core is given. This id indicates which \( \rho \)-VEX core should be configured and by using this id multiple \( \rho \)-VEX cores can be used in a single system. The second argument is the id of the program that needs to be executed on the \( \rho \)-VEX core. Each \( \rho \)-VEX program has a unique id so this can be used to transfer the correct instruction memory data. To ensure the accelerator is in a known state and is capable of receiving the instructions, a reset signal is issued before the transfer commences.

```c
molen_SET( int rvex_id, int * program_id );
```

Listing 3.5: The MOLEN set instruction
3.4.2 MOLEN MovTX function

All arguments for the accelerator are transferred to the ρ-VEX core using this move function. We have not implemented separate exchange registers as used in the MOLEN paradigm, but instead opted to write directly to the data memory available on the ρ-VEX core. Instead of the single movtx instruction to move data from the GPP to the Reconfigurable Processor (RP), two variants are created in the hArtes compiler. The different types allow different types of variables to be exchanged with the RP. The basic move is one of these variants, this simply copies the local value to the correct exchange register. In order to be able to use pointers a second variant is used. With pointer arguments the size of the data can be multiple words, so the size of the data is passed. Also a boolean flag is passed to the function indicating if the variable will be copied back after execution has finished.

```c
// Move a regular variable or value to the r-VEX core
molen_MOVTXval( int rvex_id, int register_index, int value );

// Move a pointer variable to the r-VEX core
molen_MOVTXaddr( int rvex_id, int register_index, u32 * source_address, int size, int copy );
```

Listing 3.6: The MOLEN move to exchange register instructions

3.4.3 MOLEN MovFX function

After execution has finished the return values of the original function have to be read and stored in local registers. This is a simple read action on the ρ-VEX data memory. Just like the movtx two variants are defined of this function. Regular variables and literals are read from the ρ-VEX, pointer arguments are read from the location they were stored on by the movtx function.

```c
// Read a regular variable or value from the r-VEX core
molen_MOVFX( int rvex_id, int register_index, u32 * store_address );

// Read a pointer variable from the r-VEX core
molen_MOVFXaddr( int rvex_id, int register_index, u32 * store_address, int size );
```

Listing 3.7: The MOLEN move from exchange register instructions

3.4.4 MOLEN Execute function

The start of execution is a single control signal send to the appropriate ρ-VEX core. This simply writes the START signal to the control register of the ρ-VEX wrapper.

```c
molen_EXECUTE( int rvex_id );
```

Listing 3.8: The MOLEN execute instruction
3.5. CONCLUSION

3.4.5 MOLEN Break function

In order to be able to synchronize execution of the \( \rho \)-VEX core and the \( \mu \)Blaze processor the break function can halt the \( \mu \)Blaze until execution on the \( \rho \)-VEX has finished. This requires the \( \mu \)Blaze to know when a \( \rho \)-VEX finishes its execution and this could easily be done using polling. The \( \mu \)Blaze will simply request the run state in a loop and exit this loop when the \( \rho \)-VEX core replies that it has finished. This is a very simple implementation but has its disadvantages. The constant polling will create a lot of data on the PLB bus. This would impact other devices that want to communicate at the same instant.

An alternative solution is to let the \( \rho \)-VEX core send a signal to the GPP to indicate it has finished. Instead of polling the \( \rho \)-VEX to check if execution has finished, the \( \mu \)Blaze only needs to check if the done signal has been received from the core. By using interrupts this behaviour can be created, without the need to implement the \( \rho \)-VEX cores as a Master on the bus. This interrupt could in the future also be used to signal other events on a \( \rho \)-VEX core, such as an exception or internal interrupt. When the \( \rho \)-VEX finishes execution it generates an interrupt signal for the \( \mu \)Blaze. When the interrupt is received the \( \mu \)Blaze will determine who send the interrupt and why. When the MOLEN break function is executed it will wait until the correct interrupt is received before continuing execution.

\[
\text{molen\/BREAK( int rvex\_id )};
\]

Listing 3.9: The MOLEN break instruction

3.5 Conclusion

This chapter discussed the hArtes software design and implementation for the \( \mu \)Blaze processor. A choice was made on the type of MOLEN implementation and the implications this has on the resulting system. Following the decision to use the hArtes compiler we discussed the changes we had to make in the existing \( \mu \)Blaze GCC compiler. This involved changes to the front and middle-end of the compiler to parse the MOLEN directives. Furthermore the hArtes Architecture Description file was explained and the modifications used for our \( \rho \)-VEX components were explained. Our modified compiler will output MOLEN function calls which have to implement the correct functionality for these calls. Details about MOLEN functions were given and the functionality was explained.
In this section the design of our ρ-VEX wrapper will be detailed. Due to requirements set by our wrapper changes are also made to the ρ-VEX core to make it compatible with our wrapper. The aim of the wrapper is to connect the ρ-VEX core with the μBlaze processor, an overview of the system is shown in figure 4.1. The resulting system will then be able to operate following the Molen paradigm. In this Molen polymorphic processor we will use the μBlaze processor as General Purpose Processor (GPP) and the ρ-VEX core will function as Reconfigurable Processor. Design decisions are chosen that the aim to create a flexible system which can easily be adapted to meet the future requirements.

![Figure 4.1: Overview of the hardware](image)

This chapter will start off with Section 4.1 discussing the requirements for our wrapper design. Next we discuss the selection of an interconnection scheme in Section 4.2. In Section 4.3, the communication scheme over the bus is discussed. Section 4.4 details the required status and control logic used between the wrapper and the ρ-VEX processor. An interrupt system is used to signal the μBlaze when a ρ-VEX processor finishes execution, this is discussed in Section 4.5. The ρ-VEX memory access is designed in Section 4.6. Finally, the conclusion for this chapter is presented in Section 4.7.

## 4.1 Requirements

Our platform is aimed to interconnect the μBlaze processor and the ρ-VEX processor. As discussed in the previous chapter this is achieved by using the Molen paradigm. By adopting the hArtes compiler we designed an easy to use system which separates the
software toolchain from the hardware implementation. With this compiler the main requirements of flexibility and ease of use are met. The requirements on the hardware implementation build on the results of the software implementation and have to offer a flexible system. Due to continued work that is being performed on both the $\mu$Blaze processor and the $\rho$-VEX processor, the hardware has to be easily adoptable to the changes in these two hardware components. Additionally, a future intent within the ERA project is to replace the bus to a Network on Chip. This is out of the scope of this thesis, but it stresses the requirement for a flexible system.

4.2 Bus selection

The General Purpose Processor (GPP) and the Reconfigurable Processor (RP) in the Molen polymorphic processor are connect via a bus interface. This interface is used to transfer control, status and data information between the two processors. In the original Molen implementation only a single RP was used, so a simple interconnect was able to fulfill this task. With the newer $\text{hArtes}$ implementation this has changed somewhat, because now it is possible to use multiple RPs in a single system. This changes the demands for the used bus interface. In the Molen system the Device Control Register (DCR) bus is used. This is a simple bus interface aimed to transfer data to and from exchange registers at peripherals \[10\]. It is a simple daisy-chained bus and has a large fan-out when multiple slaves are used. For our system this bus does not meet our requirements, and since the bus is not available on our $\mu$Blaze GPP processor we have to select another bus implementation.

The $\mu$Blaze processor has built-in support for several bus variants. A selection of one of these buses is preferred, because using a custom bus would require changes to the $\mu$Blaze hardware or use of a bridge. Changing the $\mu$Blaze processor is not a viable option as this would create a lock-in to a specific $\mu$Blaze processor version and bus interface. Additionally, one of the available buses on the $\mu$Blaze can be used combined with a bridge to connect the custom bus. But this would reduce or remove the advantages of a custom bus. As a result one of the built-in bus interfaces is the preferred option, as long as our requirements can be met.

The following bus interfaces are available on the $\mu$Blaze processor. More information on these buses can be found in Section 2.3.

- **Local Memory Bus** (LMB), a single-cycle bus to access on-chip dual-port block RAM. Local memory in BRAM is connected to the $\mu$Blaze processor using this bus.
- **Processor Local Bus** (PLB), supports split transactions and also burst transfers. Aimed to connect fast components.
- **On-Chip Peripheral Bus** (OPB), single synchronous handshake handshake protocol. This bus is intended to connect slower peripherals.
- **Advanced eXtensible Interface** (AXI), a new interface introduced in version 7 of the $\mu$Blaze processor. Allows burst transfers and quality of service.
4.3 Wrapper communication

As a starting point for the ρ-VEX wrapper a basic bus slave is designed. The objective for this basic wrapper is to implement the communication from the μBlaze to the wrapper, without any logic to control and monitor a ρ-VEX core. Transfer of control information has to be send to the wrapper and status information needs to be read. As part of the Xilinx XPS toolchain a PLB bus handler is used to handle the PLB bus communication. This handler will translate the bus request for the wrapper into ready to use signals. The interface between the PLB handler and the wrapper is detailed in figure 4.2.

We require two control signals that will be used to set the state of the ρ-VEX processor.

- **Reset**, as the name indicates this signal forces the wrapper and ρ-VEX core to reset. This reset needs to be send before the ρ-VEX core is setup, to ensure the ρ-VEX is in the correct state. Additionally no mechanism is available for the wrapper to stop a ρ-VEX processor once it has started execution. The wrapper can only reset the ρ-VEX if the processor fails to finish its execution.

- **Start** is the signal for the ρ-VEX processor to begin executing its program. All setup of the instruction and data memory should be finished before this signal is given. The ρ-VEX will continue running until it executes a STOP instruction or if the processor is reset.

The μBlaze processor requires some status information from the wrapper in order to know its state and to apply the correct control signals. The available status signals are.

**Fast Simplex Link** (FSL), this is a uni-directional point-to-point link.

The LMB is reserved for the memory interface so this is not of interest for us. The Fast Simplex Link is a point-to-point link, since we want to be able to easily use multiple ρ-VEX instances, each with its own memory, this is also not applicable. Of the remaining interfaces the OPB can be eliminated because of its speed. The OPB is used to interface slow peripherals, since we want a fast interface this is not interesting. This leaves two bus interfaces, the PLB and Amba AXI4. The AXI and PLB bus alike offer high speed interfaces for multiple slaves and masters. Two disadvantages of the AXI bus are that the PowerPC found on our older FPGA boards, the Virtex-II pro and Virtex4, do not implement this bus. Secondly the AXI bus is only implemented in the most recent version of the μBlaze, this version cannot be used on some older boards thus breaking compatibility. Therefore, the PLB bus remains as our bus interface of choice. Just like the Molen move instructions, the PLB is memory mapped, which will make implementing the bus in software rather straight forward. The high speed transfer rates and possibilities like burst transfers will limit the latency of executing a task on a RP. Due to the availability on both the μBlaze and PowerPC processors and the high performance the PLB bus is used to interconnect the μBlaze with the ρ-VEX. To implementing the interface both software and hardware have to be development to facilitate the connection. Additionally, the interface of the ρ-VEX processor needs to be adopted to allow it to be used as an accelerator.
Idle, this signal indicates the $\rho$-VEX processor is currently in the idle state. Only when the $\rho$-VEX processor during executing an application will this signal equal zero.

Done, after the $\rho$-VEX processor encounters a STOP instruction during executions it will go into a done state and assert this signal. The wrapper will use this signal to notify the MicroBlaze processor that the $\rho$-VEX core has finished.
4.3. WRAPPER COMMUNICATION

An overview of the resulting control and status interface is shown in Figure 4.3.

![Figure 4.3: The basic wrapper design](image)

4.3.1 \( \rho \)-VEX Wrapper control and status registers

Both the control and status data is stored in registers located in the wrapper. Two registers are used, one control register and one status register. Although only two control bits are currently implemented the control register is eight bits wide. The additional unused bits can be used for future features. Figure 4.4 shows the organization of the control register.

![Figure 4.4: Control register organization](image)

A second register, the status register, is used to hold the status information. Just like the control register this status register is eight bits wide, but only two bits are currently used. The organization of the status register is shown in Figure 4.5.

![Figure 4.5: Status register organization](image)

4.3.2 \( \mu \)Blaze instructions

The PLB bus is implemented as a memory mapped device on the \( \mu \)Blaze. To write a word to the bus two things need to be known. The first piece of information required is the address of the device on the bus. This address is configured during the design of the system and is available as a constant in the auto-generated parameters information file. The second piece of data is the offset of the register that needs to be written or read. This depends on the implementation of the device. We have only need for two registers,
the control and status registers. When the device address and register offset are added the final memory address is obtained. By writing to this address the μBlaze will send the write command and the data over the PLB bus to the wrapper. If the memory address is read then the read command is send and the response is returned. The following code is an example of sending the reset signal and reading the status afterwards.

```c
// Reset the r-vex wrapper and core ,
rvex_reset ( RVEXPLB_BASEADDR );

// Start the r-vex core
rvex_start ( RVEXPLB_BASEADDR );

// Read the r-vex status information
status = rvex_status ( RVEXPLB_BASEADDR );

// Check if the r-vex has finished its execution
if ( status & RVEX_DONE ) {
    // The r-VEX has finished execution
}
```

Listing 4.1: C code example of controlling a ρ-VEX over the PLB

The example shown in Listing 4.1 shows usage of C Macros to send the commands and read the status for a ρ-VEX core located at the address RVEXPLB_BASEADDR. Using three macros, rvex_start, rvex_reset, rvex_status all control and status transfers are handled. Checking the status of the ρ-VEX core can then be done by comparing the returned status with the defined status values. Using these basic instructions the wrapper is controlled and can be tested by validating the response in the status information.

4.4 ρ-VEX processor control and status design

With a ρ-VEX wrapper which can receive control data and return status information the next step is taken. A ρ-VEX core is adopted to be controlled by the wrapper and then added to the wrapper. Once finished a system will be available where the μBlaze processor is able to control and read the status of a ρ-VEX processor. A piece of preconfigured software can be executed on the ρ-VEX core and once the processor finishes the execution this can be detected by the μBlaze.

Development on the ρ-VEX is an ongoing process and multiple instances of this processor are currently in use. Differences between the ρ-VEX implementations are the number of issues in the core and different optimizations to improve the cycle time and throughput. As a result the requirement for the control and status interface is a generic interface which can be used by the different ρ-VEX instances. To meet this requirement the interface as shown in figure 4.6 is defined. The memory interface will be discussed in a later section of this chapter, and is not detailed here.

In order to keep the changes made to the ρ-VEX core as small as possible the wrapper gets full control of the state of the processor. To start execution the run signal has to be raised and must remain high until the ρ-VEX core indicates it has finished execution. The control signal to run the ρ-VEX core has to be generated by the wrapper, because
the Molen control signal only indicates the start of execution and can be reset afterwards. Remembering the processing state of the ρ-VEX core is implemented by a state machine, which is shown in Figure 4.7.

When a reset is received the state machine will always return to the ready state. This state does not generate the reset signal for the ρ-VEX core, that signal is simply passed through from the local reset signal. When the start signal is received the control state will move to the second state, running, where the run signal for the ρ-VEX core is raised. The running state is left when the ρ-VEX raises the done status bit. Now the FSM will enter the done state. This done state raises the interrupt signal for the μBlaze and will exit in the next cycle, in order to enter the final state. finished is the last state which halts the pipeline. The ρ-VEX can only be restarted by a reset signal from the μBlaze.
This guarantees the state of the \( \rho \)-VEX does not change before the status and data can be read. Each of the states will exit and move the FSM to the initial state when a reset is received.

With the correct control signals from the wrapper, the functionality is introduced to the \( \rho \)-VEX processor. The original \( \rho \)-VEX processor does not support an external run control signal. Instead it will start the execution immediately after the reset signal is cleared. This behaviour is controlled by the fetch unit. Inside the fetch unit a flag is available stop, once the decoder unit decodes a stop instruction the fetch will be notified. It will then set the stop flag. This flag is used to determine if the program counter should be enabled. On normal operation the program counter increments the instruction address each clock cycle, except during a flush. Once the done flag is raised, the program counter will be disabled which will stop the execution on the \( \rho \)-VEX. This can be represented by the following states.

![State Diagram](image)

Figure 4.8: The states used in the original implementation of the \( \rho \)-VEX core

To apply the new run control signal a new state has to be added. This halt state is inserted in between the reset and run state and will halt execution when the run signal is not set. Figure 4.9 shows the new state machine. As is clear from this state machine the \( \rho \)-VEX can be halted at any time, even after execution has started. This has been done to make the required adaptations to the \( \rho \)-VEX core as small as possible. Additionally this will allow the wrapper or \( \mu \)Blaze to halt the \( \rho \)-VEX core at any execution point. In our system this is not implemented, but if for instance a remote interrupt system would be designed this becomes a welcome feature.

With the designed behaviour the adaptations to the fetch unit are very small. The run signal has to be connected to the fetch unit and the generation of the disable signal for the program counter is extended. In figure 4.9 the new state machine is displayed.

Now the \( \rho \)-VEX processor will wait until the run signal is set before it will start execution. The only thing that remains is to create the two status signals that are send back to the wrapper. First the idle signal is simply the inverse of the incoming run signal combined with the done signal, Table 4.4 displays the truth table for the idle signal. Next the done signal has to be created. Here two sources are available, the first would be the done flag inside the fetch unit. The second source would be the stop signal generated by the decode unit and passed through the pipeline. The main difference is that the done signal is set before the entire pipeline is cleared. Whereas the stop signal
4.5. INTERRUPT SIGNAL

Is only available when all pipeline stages have processed the stop instruction. As a result we have utilized the stop signal. This ensure the ρ-VEX core has finished completely.

<table>
<thead>
<tr>
<th>run</th>
<th>done</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: Truth table for the idle signal

4.5 Interrupt signal

To notify the host processor the ρ-VEX core has finished executing an application the wrapper has to send an interrupt signal to the host. A standard interrupt handler and controller is implemented, as shown in Figure 4.10. These are functional units made available by Xilinx and can simply be enabled in the XPS software. The interrupt handler is located in the bus interface and takes interrupt signals from the wrapper. This signal can represent multiple interrupts. A single interrupt signal is send from the interrupt handler to the interrupt controller. This interrupt controller is a PLB peripheral that receives all the interrupts from the interrupt handlers. All these interrupts are combined and prioritized. When a interrupt is received by the interrupt controller a single interrupt signal is generated and is send to the μBlaze processor interrupt port. Upon receiving the interrupt the μBlaze will request interrupt information from the interrupt controller over the PLB bus. This system allows multiple peripherals to generate interrupts over the single interrupt port on the μBlaze.

4.6 Memory access

Now we have a system that is able to start a ρ-VEX processor connected to the PLB and read the status of the ρ-VEX. The next phase in the development is the implementation of the memory interface. Following the Harvard architecture the instructions and data
are stored in two separated memories. Our μBlaze host processor has to be able to access these memories inside the ρ-VEX cores. This access is required in order to be able to change the algorithm executed on a core by changing the instruction and data memory. Secondly the correct arguments and data have to be supplied to the core. This is done by placing these in the Data memory of the ρ-VEX core.

### 4.6.1 Memory addressing mode

To expose the memories of a ρ-VEX processor to the bus, two implementations have been designed. The first implementation uses registers inside the wrapper to address the memory. The memory location configured in the memory address register can be read and written by a memory data register. This configuration is displayed in figure 4.11 a). Benefits of this system are the small address space required on the bus, and the relaxation of the timing constraints. The full memories can be accessed through two registers, therefore only two registers have to be accessible through the bus. Because the memory address is always known before the actual read or write access is received timing constraints for the memory access are very loose. On the other hand, the split addressing and data requests require two bus transactions, which is not optimal. Furthermore bulk transfers are not possible using this configuration.

A second implementation of exposing the memory to the bus would be to map the memory directly to the bus, as shown in figure 4.11 b). Instead of using dedicated memory registers, the memory requests are directly translated to the memory location and that memory address is then read or written. This requires more address space on the bus, because each memory address has to be mapped. However, due to the mapping, no separate address and data transfers are required any longer. As for the memory space on the bus, 32 bit bus addressing is used and with a 16 bit memory address, which is larger then 11 bits currently used, still 16 bits are available for addressing different devices. The 16 bits available for addressing could address \(2^{16} = 65536\) different memories, and thus mapping the memories is a valid option. Added with the benefit of the single transaction for random reads and writes the mapped memory is our preferred implementation.
4.6. MEMORY ACCESS

Figure 4.11: Memory mapping options, a) Memory is accessed through special memory registers inside the wrapper. b) The whole memory is directly mapped on the bus.

4.6.2 Memory interface

In order to be able to interact with the ρ-VEX memories a memory interface has to be designed and some memory control logic has to be implemented. This control logic will translate the read and write requests from the bus to the correct memory location and control signals for the local memories. The memory control logic can be placed in the wrapper or in the ρ-VEX core itself, as shown in figure 4.12. If the wrapper holds the control logic the ρ-VEX core would only require minimal adaptations to allow the memory access. This would make adopting a given ρ-VEX core easy, but this also has its cons. Due to the different configurations of the ρ-VEX cores related to issue width, core size, and other optimizations, different memory implementations are used. These different memory implementations require different memory control logic and with this logic residing in the wrapper, multiple wrappers would have to be implemented. By placing the memory control logic inside the ρ-VEX core and defining a standard memory interface between the wrapper and the core, only a single wrapper could be implemented for all cores. Each core would be responsible for the control logic and would be able to optimize this logic for its specific needs. We want to create a flexible system and the abstraction level offered by placing the memory logic in the ρ-VEX core will benefit this goal.

With the decision to use a single wrapper implementation independent of the actual memory implementation an abstraction level can be introduced with a standard memory
interface. We did the same for the control and status signals, and as a result a wrapper is created that will be able to interface many different ρ-VEX processor implementations. The memory interface should be easily mapped to the bus requests entering the wrapper and has to be able to be efficiently translated into the final memory signals in the ρ-VEX core. We perform this by investigating the bus request signals and the memory control signals.

4.6.2.1 Bus request signals

When the PLB interface detects a request for the current device the request is send to the wrapper using the signals displayed in figure 4.13.

- The memory address is send through the Bus2IP.Addr signal. By default the memory address is placed in the least significant bits, and the bus device address in the higher bits. As a result the correct memory address can easily be extracted by dropping the most significant bits from the address.

- The Bus2IP_CS signal is generated by the bus interface and decodes the device
4.6. MEMORY ACCESS

Figure 4.13: Memory signals connecting the PLB and the wrapper

address to determine which memory is currently addressed. The width of the Chip Select signal is equal to the number of configured memories, with each memory having its dedicated select signal.

- Using the active low read-not-write bit $Bus2IP_{RNW}$ signal, it is determined if the current transaction is a read or a write request.

- For write transfers the data that has to be written in available in $Bus2IP_{Data}$. Depending on the bus width this signal can range from 32 bits to 128 bits wide. The $\mu$Blaze is only able to send 32 bit of data each transaction, so we have no benefit when using a bus width higher then 32 bits.

- A Byte Enable signal is available in $Bus2IP_{BE}$. Since we use a 32 bit wide bus, and our memories are only 32 bit addressable we do not use this signal.

- Burst transaction allow fast transfer of larger chunks of data. During such a burst transaction the $Bus2IP_{Burst}$ bit will be high to indicate such a transfer.

- With the $Bus2IP_{BurstLength}$ signal the size of a burst is known. Each burst can have a maximum length of 128 bytes [20] and the actual length is specified in this
When a valid read transaction is available, the Bus2IP.RdReq signal is driven high to indicate this. This signal can be used to generate the acknowledgement for an incoming event.

The write counterpart of the previous signal is the Bus2IP.WrReq, which signals an incoming write request.

The first output of the wrapper is the IP2Bus.AddrAck signal. This acknowledgement signal has to be generated when a read or write transaction is received.

On read requests the corresponding data is sent to the bus via the IP2Bus.Data signal. This data has the same width as the bus, so in our case this is 32 bits wide.

Apart from the address acknowledgement signal, also a read acknowledgement has to be generated when the read request is handled by the wrapper. This ack is sent over the IP2Bus.RdAck, because the memory requires one cycle to output the data this acknowledgement is delayed by one cycle.

The final acknowledgement signal indicates the correct reception of write requests, and is passed through the IP2Bus.WrAck signal.

For the most part the memory signals generated by the bus handler can be directly passed through to the ρ-VEX core. The signals are all ready formatted in a generic and practical way to be interpreted by a ρ-VEX core. By filtering out some of the signals a more compact interface can be implemented. To remove any dependencies between the ρ-VEX core and the bus interface all bus related logic should be handled by the wrapper. Hence all acknowledge signals are handled by the wrapper. The Byte Enable signal is not used in our implementation, so we can simply drop that signal. The resulting memory interface then becomes:

![Memory signals connecting the wrapper and the ρ-VEX core](image)

- The incoming Bus2IP.CS signal is simply passed on as the mem.select signal.
4.6. MEMORY ACCESS

• The `mem_address` are the bits of the Bus2IP.Addr signal that represent the actual memory address. As the higher bits contain the device address, only the least significant bits are forwarded to the ρ-VEX core. Presently the memories inside the ρ-VEX core require a maximum of 11 bits, so the lower 11 bits are placed in the `mem_address` signal.

• By default all incoming request are assumed to be read request. The `mem_write_enable` signal indicates the current transaction is a write request, and the valid data is present on the data signal.

• With valid write transactions, the corresponding data is placed on the `mem_write_data` signal.

• Finally the last signal in the memory interface is the data coming from the memories on read requests, which is placed on the `mem_read_data` signal.

With the definition of these signals the design of the memory interface is completed. The wrapper has the task to generate the required acknowledge signals and pass through the memory signals in the correct format. It is up to the ρ-VEX core to execute the memory transactions on the selected memory and ensure that correct data is read or written.

4.6.3 Memory control logic

It is up to the ρ-VEX system to ensure that incoming memory transactions are mapped to the correct memory block. Each ρ-VEX system has two memory blocks, instruction and data memory. Additionally a dummy memory is used that holds status information of the ρ-VEX core. Because each of the memories has its own specific organization the control logic for the memories differs. Figure 4.15 shows an overview of the system organization.

![Figure 4.15: Overview of the system organization](image-url)
4.6.3.1 Data Memory interface

The data memory is organized as a dual-write and dual-read port BRAM module. One read port and a write port are used by the $\rho$-VEX core, the second read port is utilized by a UART interface. This UART interface will send the contents of the data memory over a serial link after execution is finished. This is utilized to verify the correct execution and is the only outgoing interface on the stand-alone implementation of the $\rho$-VEX core. As we implement reading this data memory through the bus the UART has become obsolete and can be removed. By removing the UART unit one of the read-ports is freed, and can be used for the bus interconnect. The second write port was unused, so this write port can directly be used by the wrapper. An overview of the data memory logic is given in figure 4.16.

![Data Memory (BRAM)](image)

Figure 4.16: Data memory overview

4.6.3.2 Instruction Memory interface

Originally the $\rho$-VEX core only reads the instruction memory. As a result this memory is implemented as a simple ROM with a single read port. In our implementation we need to be able to write to the instruction memory, so we have to convert this memory. Just like the data memory the instruction memory is mapped to BRAM and has two read ports. One read port is used by the $\rho$-VEX core and the remaining read port is directly available for the wrapper. Unlike the data memory only a single write port is utilized for the instruction memory. The $\rho$-VEX core has no write access to the instruction memory, so the single write port is utilized by the wrapper. An overview of the instruction memory is shown in Figure 4.17. Strictly speaking the wrapper does not need to be able to read instruction memory, but this is preferred in order to be able to verify the instruction memory contents.

Each instruction for the $\rho$-VEX has a width of 32 bits $\times$ number of issues. So a 4-issue $\rho$-VEX core will need to read 128 bit wide instructions. This introduces a problem when the memory is read or written by the wrapper. The wrapper and PLB can only handle 32 bits of data for each transfer, so for a 4-issue configuration this requires extra addressing logic to be able to read and write the full 128 bit instruction data. The solution to this is to use a word selectable memory. Just like the data memory the instruction memory is exposed to the wrapper as a 32 bit word addressable memory.
4.6. MEMORY ACCESS

Depending on the number of issues of the ρ-VEX core a few of the least significant bits of the address are then extracted from the address and used to select a single word from the full instruction [4.18].

4.6.3.3 Status Memory interface

A third memory is introduced inside the ρ-VEX core. This memory is used to store status data which can be read over the bus. This status memory is not organized as a real memory, but instead offers an interface to read internal data for the GPP. Information like the cycle count and current program counter are available through the status memory, and the function is mainly for debugging purposes. In future work this memory is also designed to hold interrupt and exception information, thus removing the need to expand the interface with the wrapper. The status data could also have been mapped on either the data or instruction memories, but this would complicate the logic for that memory. Additionally this would result in a less flexible implementation, as simply adding some information to the status memory would require extensive changes.
to the control logic.

![Status Memory Diagram](image)

Figure 4.19: Overview of the status memory

Because all data available in the status memory is directly available inside the ρ-VEX core, the memory does not require any new registers to store the data. Instead the existing status registers can be forwarded directly to the output, see figure 4.19.

### 4.7 Conclusion

In this chapter we have presented the design of our bus wrapper for the ρ-VEX processor. We started by defining the requirements for the wrapper. A suitable bus interface has been selected that will be used to connect the ρ-VEX cores to the µBlaze processor. Important features for the selected bus were the throughput and scalability. With the bus selected the communication is detailed. This includes the different commands used by the control software on the µBlaze to control a given ρ-VEX core. Next the control and status interface between the wrapper and ρ-VEX processor was designed. Aimed at flexibility this interface is able to control the connected processor and read-back the status information of the processor. The last part of this chapter focussed on the memory interface. Again a flexible interface is designed that allows setting and reading of the memories residing inside the ρ-VEX processor core.
In the previous chapters we have discussed the design of the compiler and wrapper. In this chapter, we present our implementation of the complete system. First we discuss the implementation methodology that was used in Section 5.1. Following this methodology we start with the implementation of the $\mu$Blaze compiler in Section 5.2. Subsequently we discuss the wrapper and $\rho$-VEX implementation in Section 5.3. Changes made to the $\rho$-VEX system are presented in Section 5.4. A summary of this chapter is given in the final section, Section 5.6.

5.1 Implementation methodology

Implementation of the software and hardware aspects for our system is split in a few steps. Designing the system in one go would be a daunting task and error prone. By using a bottom-up strategy during the implementation phase, each step is verified before the next step is taken. This gave us the ability to test the different components individually and find the bugs early in the implementation phase. During implementation of the compiler applying a bottom-up strategy proved to be difficult. The structure of the compiler depends on a lot of the different components and splitting these often breaks the compiler. By extensively keeping track of the applied changes and by monitoring the build process, we have tried to make the implementation of the compiler as smooth as possible.

Luckily with the implementation of the wrapper, the bottom-up approach can be extensively applied. We start with a basic bus implementation. With a stable bus interface, the different components in the wrapper can be individually implemented and tested.

5.2 Compiler implementation

As a starting point the two existing compilers were taken. The $\mu$Blaze compiler, as part of the Xilinx toolchain, targets our host processor. Where as the hArtes compiler implements the MOLEN functionality. After investigating the two compilers it became clear that the best option was to use the $\mu$Blaze compiler as the starting point and apply the MOLEN functionality to that compiler. This was caused by the difference in integration both compilers had of their specific functionality. With the $\mu$Blaze compiler the changes are made throughout different compiler components. Additionally, the changes are is tightly interwoven into existing GCC source code. On the other hand, the hArtes compiler has a more modular approach on its integration into the GCC compiler. Most of the changes required to integrate the hArtes functionality are located in new files and custom functions.
CHAPTER 5. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMP</td>
<td>Library for arbitrary precision arithmetic [4]</td>
</tr>
<tr>
<td>MPFR</td>
<td>Library for multiple-precision floating-point computations [3]</td>
</tr>
<tr>
<td>Expat</td>
<td>XML parser, used to read the hArtes configuration file [1]</td>
</tr>
</tbody>
</table>

Table 5.1: Build dependencies

<table>
<thead>
<tr>
<th>Script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>common.sh</td>
<td>Common file for path and filesystem detection</td>
</tr>
<tr>
<td>build_binutils.sh</td>
<td>Executes the first step in the build process, create the binutils</td>
</tr>
<tr>
<td>build_gcc.sh</td>
<td>The main script, this configures and builds the compiler</td>
</tr>
</tbody>
</table>

Table 5.2: Build scripts

5.2.1 Build dependencies

With the starting point chosen the dependencies for both compilers have to be met. The GCC compiler has a lot of dependencies, but most of the common dependencies for GCC can are all ready present on most systems, or they can be installed in a single development package. There are however a few dependencies that are less common, and probably require manual installation. These dependencies are shown in Table 5.2.1.

5.2.2 Build scripts

Supplied with the μBlaze GCC compiler are some build scripts for the different phases of the build process, see Table 5.2.2. Since the configuration required to build the μBlaze GCC compiler is rather extensive these scripts are an useful tool. The build scripts apply the correct naming convention used by the μBlaze toolchain, required libraries are configured, and the build paths are defined. By using the build scripts and extending them with the MOLEN specific configuration, compilation of the hArtes μBlaze GCC compiler has become an easy task.

5.2.3 Compiler data flow

As discussed in Section 3.2 the compiler is split in tree phases: front-end, middle-end and back-end. The changes required for the MOLEN functions are located in the front-end and middle-end of the compiler. There is no direct communication between the two phases, so the code trees are extended with MOLEN dummy trees in order to pass data from the front-end to the middle-end phases.

We cannot apply the entire MOLEN functionality in a single pass, because there are multiple ways a function can be annotated. The function definitions and the function calls can be annotated, but these require different parsing to handle. With function definitions all calls to that function have to be replaced, so before any call to this function is parsed the function has to be parsed first. With function calls the actual accelerator target can be changed from its default definition. To be able to handle all situations in a clear solution, the intermediate dummy MOLEN tree is used. As a result, all calls to an annotated function can easily be detected in the middle-end, irrespectively of the used
arguments and the origin of the annotation. As a result two passes are used to parse the MOLEN annotations, these passes are discussed in the next sections.

5.2.3.1 Front-end data flow

In the first phase of the compiler, the input C code is read by and parsed into a tree representation. The MOLEN extension is applied by defining a new *pragma* annotation. This is done by registering the pragma with the following command.

```c
#define REGISTER_TARGET_PRAGMAS() do {
    /* Support for pragma call_hw */
    c_register pragma (0 , "map" , microblaze pragma call hw);
} while (0)
```

Listing 5.1: Register the Pragma annotation

When the `map` pragma is detected by the C-paser the `microblaze pragma call hw` is invoked. Within this function the annotated code is verified and a new dummy MOLEN tree is created. All following C code is then parsed normally, until the entire function body has been parsed. When the entire annotated code is parsed, the new tree is inserted into the main tree as the MOLEN dummy tree, instead of a regular function tree. This data flow is shown in Figure 5.1.

---

![Figure 5.1: Data flow in the Front-End of the compiler](image-url)
5.2.3.2 Middle-end data flow

During the middle-end of the compiler optimizations are applied to the code tree. This is done in multiple passes and by adding an additional pass we are able to convert the MOLEN dummy trees into the correct MOLEN function calls. The FSM shown in Figure 5.2 shows the data flow in the middle-end.

Each tree contained in the main tree will be checked. If the tree is not a MOLEN dummy the pass will be skipped. When a dummy tree is found the dummy is removed from the tree. The original tree representing the function call is extracted, and the corresponding data is read from the XML configuration file. Each statement inside the tree is then handled individually. The statements represent the original call, arguments and return data defined in the source code. The call statement is converted into the MOLEN Set function call, this function call receives two arguments the device id and the accelerator id.

```cpp
// Create the Molen set function call
new_call = build_call_expr (fpga_set, 2, application_id, device_id);
```

Listing 5.2: Create the Molen set function call

Each argument passed to the original function is represented by a statement. These statements are all checked twice. First the arguments are converted into the two movtx function calls. Regular arguments are send using the Molen MovTX function. Pointer arguments are send using the Molen MovTXaddr call. The second time the statements are checked is used to generate the Molen MovFXaddr calls required for the pointer arguments when the execution is finished.

When all movtx calls are genereated the accelerator can be started. This is done by issuing the Molen_Execute call. Before the µBlaze can read the results generated by the accelerator both processors have to be synchronized. This synchronization is handled on the µBlaze by inserting a Molen_Break call. Once the ρ-VEX core has finished it’s execution the results generated on the accelerator and needed by the µBlaze can be read back. The second time the arguments are parsed is therefore scheduled at this moment. This concludes the function call generation for this tree item.

5.3 Wrapper implementation

The wrapper implementation consists of two parts. The control and status logic and the memory logic. The control logic has to generate valid control signals for the encapsulated ρ-VEX core, plus it has to monitor the status information of the ρ-VEX processor. This is the purpose of the status logic. The memory logic will parse incoming memory requests from the bus and pass these on to the ρ-VEX system.

5.3.1 Wrapper control and status logic

Both the control and status logic are implemented as bus accessible interface registers. By writing to the control register the control signals can be changed, and the status
5.3. WRAPPER IMPLEMENTATION

Figure 5.2: Middle-End data flow FSM

information can be read from the status register. In addition to these two registers there are some extra registers that can be utilized. The purpose of these registers is
to aid in debugging a wrapper and ρ-VEX core. Furthermore, they support future development of the ρ-VEX core, such as interrupt support. The impact of the extra addressing space required for these registers is very limited, because there is plenty of address space available on the PLB bus. Furthermore by default only the higher 16 bits are used for addressing a bus device in order to simplify the address recognition logic. As a result 16 bits of the addressing space is available for the registers, even when only a single register is implemented. These extra registers can be utilized to output debugging information such as the wrapper state. Additionally, a register has been defined that is hard-wired with a wrapper version number. Using this version number the version of the implemented wrapper and the ρ-VEX core can be identified. This is especially useful when multiple versions are implemented on the same fabric.

Transfers over the PLB bus to one of the registers can have a size of 32 bits. Our 8 bit wide control and status registers are mapped to the 8 least significant bits. The following code shows the logic implemented to handle writes to the control register.

```vhdl
case slv_reg_write_sel is
  when "10000000" =>
    -- Only read the two least significant bytes. All others are discarded
    if ( Bus2IP_BE(3) = '1' ) then
      rvex_ctrl_r(7 downto 0) <= Bus2IP_Data(24 to 31);
    end if;
  when others => null;
end case;
```

Listing 5.3: VHDL code that handles storing incoming control data from the PLB

The status register cannot be written using the PLB so only logic enabling writing to the control register is implemented. Reading the status register is implemented using the following VHDL code.

```vhdl
case slv_reg_read_sel is
  when "01000000" => slv_ip2bus_data <= x"000000" & rvex_status_r;
  when "00000001" => slv_ip2bus_data <= x"00010605";
end case;
```

Listing 5.4: Register reading logic

Here a read request from the PLB is received. When the requested address equals the status register the two status bits are returned, padded with zeros to create a 32 bit response. In addition to the status register, a hard coded version number can be read.
This version number can be used to identify the wrapper type and implemented ρ-VEX core.

5.3.2 Wrapper memory implementation

As discussed earlier in Section 4.6, the memory signals coming from the bus handler can for the most part be directly forwarded over the memory interface to the ρ-VEX core. The VHDL code shown below takes care of this forwarding.

```
−− Pass the memory requests down to the rvex.
rvex_mem_select <= Bus2IP_CS;
rvex_mem_addr  <= Bus2IP_Addr(C_SLV_AWIDTH-13 to C_SLV_AWIDTH-3);
rvex_write_enable <= not(Bus2IP_RNW);
rvex_mem_write_data <= Bus2IP_Data;
−− Send data to the bus, register (status) data or memory data
IP2Bus_Data <= slv_ip2bus_data when slv_read_ack = '1' else
               rvex_mem_data_out when mem_read_ack = '1' else
               (others => '0');
```

Listing 5.5: Forwarding the memory signals to and from the ρ-VEX core

Two things should be noted from the code. The address sent to the ρ-VEX core is offset by four bits from the original address received from the bus. This is necessary because the address uses byte addressing and we only support word addressing. Also we invert the read-not-write bit signal to obtain the write-enable signal. This is done so the ρ-VEX can directly use this signal its memories.

The acknowledge signals for the PLB bus are also generated by the core, this is done as follows.

```
mem_read_enable <= ( Bus2IP_CS(0) or Bus2IP_CS(1) or Bus2IP_CS(2) ) and
                 Bus2IP_RdReq;
mem_read_ack   <= mem_read_ack_dly1;
mem_write_ack  <= ( Bus2IP_CS(0) or Bus2IP_CS(1) or Bus2IP_CS(2) ) and
                 Bus2IP_WrReq;
IP2Bus_AddrAck <= slv_write_ack or slv_read_ack or mem_write_ack or
                 mem_read_enable;
IP2Bus_WrAck   <= slv_write_ack or mem_write_ack;
IP2Bus_RdAck   <= slv_read_ack or mem_read_ack;
```

Listing 5.6: Code responsible for the acknowledge signals

The mem_read_ack_dly1 signal is the mem_read_enable signal delayed by one clock cycle. The delay is required, because a read takes two cycles. First the correct address is set and the next cycle the contents of the memory can be read.
5.3.3 Interrupt generation

When the \( \rho \)-VEX core indicates it has finished executing the application the wrapper sends an interrupt signal to the \( \mu \)Blaze processor. As discussed in Section 4.5 this is implemented using two provided by the Xilinx XPS software. The Interrupt Handler located in the bus interface handler receives the signal from the wrapper. This signal contains four interrupt signals. At the moment we only implement a single interrupt, the done interrupt. The remaining interrupts can be used in for example future development on exceptions and interrupts on the \( \rho \)-VEX core. The signal is generated as follows.

\[
\text{−− Generate an interrupt when the rvex is done}
\]
\[
\text{IP2Bus\_IntrEvent} \leftarrow \text{"000"} \& \text{rvex\_done\_interrupt};
\]

Listing 5.7: Interrupt signal generation

The rvex\_done\_interrupt signal is generated in the control state machine.

5.4 \( \rho \)-VEX System implementation

In order for the \( \rho \)-VEX core to be connected and controlled by the wrapper some changes had to be made to the \( \rho \)-VEX system. These changes implement the control and status logic, and the interface to the memories residing in the \( \rho \)-VEX core.

5.4.1 Control signals

Two control signals are received from the wrapper. The first one, reset, is directly connected to the reset signal in the \( \rho \)-VEX system. Previously this signal was connected to the chip reset, but with the new bus this is no longer required. Applying the run signal is a bit less straightforward. The run signal is forwarded to the fetch unit and there the signal is implemented. In the fetch unit the run signal is used for the flush and pc\_disable signals, to stop the program counter.

\[
\text{−− Generate the signal that will stop the program counter when execution is finished, or the core is halted by the wrapper.}
\]
\[
\text{pc\_dis: pc\_disable} \leftarrow \text{stop OR (NOT run)};
\]

\[
\text{−− Generate the flush signal. The stop signal is a flag indicating a STOP instruction has been executed.}
\]
\[
\text{lnop: Flush} \leftarrow \text{PCS\_e OR pc\_disable OR reset};
\]

Listing 5.8: Flush and PC\_disable signal generation inside the Fetch Unit

5.4.2 Status signals

The two status signals that we utilize to monitor execution on the \( \rho \)-VEX core are idle and done. Both status signals are readily available in the \( \rho \)-VEX system layer. So the two status signals can simply be output by connecting them to the correct internal
5.4. \(\rho\)-VEX SYSTEM IMPLEMENTATION

5.4.3 Memory implementation

Each \(\rho\)-VEX core has three memories. These are the instruction, data, and status memories. Since the demands for each memory differs all three memories have their own unique implementation.

5.4.3.1 Status memory implementation

The status memory does not contain any storage elements, but instead data readily available in the \(\rho\)-VEX core is directly output when requested. Therefore the status memory is not implemented as a separate component, but is integrated into the \(\rho\)-VEX system. Currently the memory only contains two fields, cycle count and program counter. The cycle count represents the number of cycles the core has been executing a given program. On reset the cycle count is reset to zero and when the program is finished the counter will automatically stop. Using the cycle count an indication of the speed of an algorithm can be determined.

The second record in the status memory, the program counter, can be used to determine if the \(\rho\)-VEX finished execution correctly. By checking the program counter with the expected stop instruction address it can be ensured the core has finished correctly. The following listing shows the implementation of the status memory.

```
--- Status memory process
mem_data: process (mem_read_address, pmem_read_address_1, cycles)
begin
  case mem_read_address(1 downto 0) is
    --- cycles
    when "00" =>
      status_data_out <= cycles;
    when "01" =>
     status_data_out <= x"00000" & "000" & pmem_read_address_1;
    when others =>
      status_data_out <= (others => '0');
  end case;
```

Listing 5.9: Flush and PC disable signal generation inside the Fetch Unit
5.4.3.2 Data memory implementation

The data memory is implemented as Block RAM (BRAM), an overview of interface is given in Figure 5.3. The first read address is used by the $\rho$-VEX and the address received from the wrapper is directly forwarded to the second read port. The first write port is utilized by the $\rho$-VEX, whilst the second port is utilized by the wrapper. The two data out ports are output to the $\rho$-VEX core and the wrapper respectively.

Figure 5.3: Data Memory interface

<table>
<thead>
<tr>
<th>Wrapper signals forwarded to the data memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmem_write_enable &lt;= (others =&gt; mem_write_enable) when mem_select(2) = ’1’ else &quot;0000&quot;;</td>
</tr>
<tr>
<td>dmem_address &lt;= mem_write_address(DMEM_LOGDEPTH - 1 downto 0);</td>
</tr>
<tr>
<td>dmem_write_data &lt;= mem_write_data;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output the data to the wrapper from the selected memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem_read_data &lt;= dmem_read_data when mem_select(2) = ’1’ else</td>
</tr>
</tbody>
</table>
5.4.3.3 Instruction Memory implementation

The Instruction Memory discussed here is implemented for a 4-issue \( \rho \)-VEX core. Currently this is the largest number of issue width that is implemented, and as a result this is the largest memory for any \( \rho \)-VEX core currently available. The read port utilized by the \( \rho \)-VEX core has a width equal to the number of issues times 32 bits. This enables reading the complete instruction, containing syllables for each of the issues, in a single cycle. The read and write port utilized by the wrapper are implemented as 32 bit signals. By fixing these two ports to a single word the wrapper is able to read and write the entire instruction memory independent of the issue width. In addition to these signals a reset signal is implemented. This reset signal is used by the \( \rho \)-VEX core during a flush. When the pipeline is flushed the Instruction Memory outputs empty instructions that are passed to the \( \rho \)-VEX core.

![Instruction Memory Interface](image)

Listing 5.11: Data forwarding for the data memory

---

---

Listing 5.12: Instruction Memory control code
## 5.5 MOLEN functions

All the MOLEN functions that are generated by our compiler have to be implemented on the host processor. As presented in Section 4.3.2, the underlying commands to control and transmit data with the $\rho$-VEX cores is done using macro statements. Most of the MOLEN functions can be implemented as simple bus transactions, these are the `execute`, `movtz` and `movfx` instructions. The two remaining instructions, `set` and `break`, require some extra logic. Definitions like the $\rho$-VEX bus address and program instructions are all configured in the host software.

### 5.5.1 PLB macros

At the basis of the MOLEN functions lies the macros that execute the PLB transactions. These macros receive the address of the $\rho$-VEX wrapper and send the correct information. Table 5.3.1 shows the five macros used, these macros can be divided into two groups. The first group contains the Control and Status macros, `rvex_reset`, `rvex_start`, and `rvex_status`. Contained in the second group are the memory management macros, `rvex_mem_read` and `rvex_mem_write`.

#### 5.5.1.1 Control and Status Macros

As detailed in Section 4.3.2, three different macros handle the control and status transfers. The first macro is `rvex_reset`, this macro is defined as follows.

```c
#define rvex_reset(BaseAddress) \
Xil_Out32((BaseAddress), RVEX_RESET)

Listing 5.13: Macro sending a $\rho$-VEX reset command
```

In this macro definition the command is expanded with a predefined value `RVEX_RESET`, which contains the value for reset command. No register offset has to be used, since the command register is located at register offset zero. The data is directed to a second macro `Xil_Out32`. `Xil_Out32` is a macro provided by Xilinx and is defined in `xil_io.h`.

```c
#define Xil_Out32(Addr, Value) \n(*(volatile u32*)((Addr))) = (Value))
```

---

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rvex_reset</td>
<td>Send a reset signal</td>
</tr>
<tr>
<td>rvex_start</td>
<td>Send a start signal</td>
</tr>
<tr>
<td>rvex_status</td>
<td>Read the wrapper status</td>
</tr>
<tr>
<td>rvex_mem_read</td>
<td>Read data from one of the memories</td>
</tr>
<tr>
<td>rvex_mem_write</td>
<td>Write data to one of the memories</td>
</tr>
</tbody>
</table>

Table 5.3: PLB transaction macros
Listing 5.14: Macro transmitting data over the PLB

Here the supplied value is written to the location pointed by the argument address. Since the PLB is memory mapped, this will result in sending the value over the PLB bus to the wrapper as long as a valid address is written.

The next statement sends the start command to the wrapper. Just like the reset instruction this is implemented as a macro. The only difference is the value passed to the Xil_Out32 function, this is $RVEX_{START}$ which is defined to be the start command.

By using the rvex_status macro the current status of the wrapper is read.

```c
#define rvex_status(BaseAddress) \ 
  Xil_In32 ((BaseAddress) + (RVEX_PLB_WRAPPER_SLV_REG1_OFFSET))
```

Listing 5.15: Macro to read the status of a $\rho$-VEX core

Two differences are clear in this macro, first of all a register offset is added to the address. The status register offset is defined in the constant $RVEX_{PLB\_WRAPPER\_SLV\_REG1\_OFFSET}$ which equals the second register in our implementation. Secondly the macro redirects to Xil_In32 which is the counterpart of the Xil_Out32 function and reads data.

```c
#define Xil_In32(Addr) \ 
  (*((volatile u32 *)(Addr))
```

Listing 5.16: Macro that reads data from the PLB

Again the final instruction is direct access to a memory address, and in this case it reads the given address.

5.5.1.2 Memory transfer macros

To read and write the memories located in the $\rho$-VEX system two macros are used. The definition for these macros is shown below.

```c
/* Write 32 bits of data to the specified memory location */
#define rvex_write_mem(MemAddress, Address, Data) \ 
  Xil_Out32((MemAddress) + ((Address) * 4), (Xuint32)(Data))

/* Read 32 bits of data from the specified memory location */
#define rvex_read_mem(MemAddress, Address) \ 
  Xil_In32((MemAddress) + ((Address) * 4))
```

Listing 5.17: $\rho$-VEX core configuration

Two address arguments are used in the macros. First the base address of the memory is supplied, the second address is the offset for the memory location. Like the other macros the Xil_In and Xil_Out macros are used to place the transaction in the correct memory location. The argument address is multiplied by four as the PLB bus uses byte
addressing, whereas we implemented word addressable memories. By multiplying the address by four the word address is converted to the byte address.

### 5.5.2 Interrupt handling

All interrupts generated by the $\rho$-VEX cores are passed through the Xilinx Interrupt Controller. To use the interrupts first the $\rho$-VEX interrupts have to be initialized, this is done by the following code.

```c
/* Register interrupts from the RVex at RVexInstancePtr with the
 * RVexIntrHandler function.
 * IntelInstancePtr and IntrId specify the Interrupt controller.
 */
Result = XIntc_Connect(IntcInstancePtr, IntrId, (XInterruptHandler)
        RVexIntrHandler, (void *)RVexInstancePtr);
```

Listing 5.18: Interrupt handler initialization

All the interrupts that are registered with the above code are sent to the RVexIntrHandler function. This function receives the instance that generated the interrupt and is responsible for handling the interrupt.

```c
/*** *
 * Interrupt handler for the rVEX generated interrupts.
 * *
 * @param pointer InstancePtr, pointer to the rVEX instance that has
 * generated the interrupt
 *
 ***/
void RVexIntrHandler(void *InstancePtr)
{
    RVexCtr *RVex;

    /* Type cast the pointer to the rVEX instance type */
    RVex = (RVexCtr*) InstancePtr;

    /* Only done interrupts are implemented, so the source of the interrupt
     * is known. Handle this interrupt by retrieving the status from the
     * rVEX core.
     */
    RVex->Status = rvex_status(RVex->BaseAddress);
}
```

Listing 5.19: $\rho$-VEX Interrupt Handler

Only a single interrupt can currently be generated by the $\rho$-VEX cores, so when an interrupt is received the source of the interrupt is known. The done interrupt is handled by updating the status information of the $\rho$-VEX core. As a result when a $\rho$-VEX has finished execution the new status will automatically be updated on the $\mu$Blaze processor. The status information can then simply be used in the molen_break function to check if a $\rho$-VEX core has finished or not.
5.5. MOLEN FUNCTIONS

5.5.3 \( \rho \)-VEX configuration

The MOLEN functions require some configuration data for each of the available \( \rho \)-VEX cores. An example of such an configuration is given below.

```c
// Structure used to store the $\rho$-VEX core configuration and status
typedef struct {
    u32 BaseAddress;       /* Base address of the registers */
    u32 volatile Status;    /* The status of the core */
    u32 DataMemAddress;     /* Start address of the data memory */
    u32 InstrMemAddress;    /* Start address of the instruction memory */
    u32 StatusMemAddress;   /* Start address of the status memory */
    RVexProgram * Program; /* Id of the currently configured program */
} RVexCtr;

#define NR_RVEX_INSTANCES 2
RVexCtr RVexInstances[NR_RVEX_INSTANCES];

void initialize_rvex() {
    /* Initialize the rvex instance 0 */
    RVexInstances[0] . BaseAddress    = XPAR_RVEX_PLB_WRAPPER0_BASEADDR;
    RVexInstances[0] . DataMemAddress = XPAR_RVEX_PLB_WRAPPER0_MEM0_BASEADDR;
    RVexInstances[0] . InstrMemAddress = XPAR_RVEX_PLB_WRAPPER0_MEM1_BASEADDR;
    RVexInstances[0] . StatusMemAddress = XPAR_RVEX_PLB_WRAPPER0_MEM2_BASEADDR;
    RVexInstances[0] . Status          = 0;
    RVexInstances[0] . ProgramID       = -1;

    /* Initialize the rvex instance 1 */
    RVexInstances[1] . BaseAddress    = XPAR_RVEX_PLB_WRAPPER1_BASEADDR;
    RVexInstances[1] . DataMemAddress = XPAR_RVEX_PLB_WRAPPER1_MEM0_BASEADDR;
    RVexInstances[1] . InstrMemAddress = XPAR_RVEX_PLB_WRAPPER1_MEM1_BASEADDR;
    RVexInstances[1] . StatusMemAddress = XPAR_RVEX_PLB_WRAPPER1_MEM2_BASEADDR;
    RVexInstances[1] . Status          = 0;
    RVexInstances[1] . ProgramID       = -1;
}
```

Listing 5.20: $\rho$-VEX core configuration
Each core has an entry in the RVexInstances array, which is a structure containing the relevant information. All initial values are loaded from the xparameters.h constants, which are auto-generated by the Xilinx XPS toolchain. As a result, changing the PLB address of a ρ-VEX core does not require any changes in this configuration. Only when a ρ-VEX core is added or removed does the configuration need to be adopted.

To define a ρ-VEX application a special tool is used. This elf2vhdl tool is a small application that reads an elf file generated by the ρ-VEX compiler and extracts the instruction and data words. Originally the elf2vhdl tool would only generate two vhdl file, pmem.vhdl and dmem.vhdl which are the VHDL representations of the memories initialized with the correct values. We have adopted the tool to generate a third file, program.h. This is a C header file and contains the values that have to be uploaded to both the instruction and data memory. In addition to the memory contents the header file also contains the program id and the memory offsets for the arguments and return values.

```c
#ifndef RVEX_MATRIX_H
#define RVEX_MATRIX_H

/* Define the program id, this variable is used by the hArtes compiler to set the program */
u32 matrix_fpga_text = 1;

RVexProgram RVEX_PROG_MATRIX = {
    /* Offsets in Data Memory for the arguments */
    /* Use the word offset, not the byte offset supplied by the object dump */
    { 0, 1, 2, 3, 4 },
    /* Word offsets in Data Memory for the return values */
    { 5, 6, 7, 8, 9 },
    1, // Number of instructions
    { /* Instruction 0 */
      0xc0000000, 0xc0000000, 0xc2000020, 0xc5820000,
    },
    /* Data memory contents */
    2, // Number of data memory words
    { /* Data memory contents */
      0x00000000, 0xffffff00
    }
};
#endif /* RVEX_MATRIX_H */
```

Listing 5.21: Auto generated ρ-VEX application header file

Each ρ-VEX application requires an unique id. The hArtes compiler inserts the application id by a reference to the variable FUNCTION_fpga_text, so the correct id has to be defined in this variable. In the example the id is set to be '1'. To specify the locations of the argument variables and return variables in the ρ-VEX data memory two arrays are defined. The first array contains five offset entries for the arguments that are moved to the ρ-VEX core. A second array contains the offsets for the variables that are
read back to the µBlaze after execution. These offsets need to be manually configured, and can be obtained by investigating the object dump of the ρ-VEX application. In addition to the above configurations, the header contains two definitions related to memory. Secondly, the actual application data is defined. This application data contains the total number of instructions and data words for this application. Additionally the actual instruction and data values are defined.

```c
/**
 * Initialize the rvex programs, takes the programs defined in the header files and stores them in a programs array
 */
XStatus initialize_programs() {
    RVexPrograms[ matrix_fpga_text.. ] = RVEX_PROG MATRIX;
    RVexPrograms[ matrix_inverse_fpga_text.. ] = RVEX_PROG MATRIX_INVERSE;
    return XST_SUCCESS;
}
```

Listing 5.22: Application initialization

By including the required application header files and calling the two initialization functions the platform is fully configured.

### 5.5.4 Molen Set implementation

When a function is mapped to an accelerator the call is replaced by the MOLEN calls, the first MOLEN call is the *molen_SET* function. The function gets two arguments, the rvex id and the application id.

```c
/* Setup the rvex core */
molen_SET( 1, matrix_fpga_text..);
```

Listing 5.23: The molen_SET call, as generated by the compiler

The set function has to transfer the entire instruction and data memory contents as defined in the corresponding header file. Before the upload starts the ρ-VEX core is first reset. This ensures the ρ-VEX core is ready to be reconfigured. To prevent reconfiguring a ρ-VEX core with a previously configured application a check is done on the application id. If the last configuration was for the same application id the upload can be skipped.

### 5.5.5 Molen MovTX and MovFX implementation

Moving data to and from the exchange registers located in the data memory is performed by using these function. Two variants of the function are used. One function to handle normal variables and a second one to handle pointers. Information about all the exchange registers is retrieved from the application header file. In particular the offset for the variable in the data memory is loaded.
CHAPTER 5. IMPLEMENTATION

/∗
  * Move a value from the rvex core to the given pointer
  *
  * @param id The id of the rvex core
  * @param register_index The index of the register
  * @param addr Pointer to the local address where the value will be stored
  */
void molen_MOVFX(u32 id, u32 register_index, u32* addr) {
    u32 offset;
    offset = RVexInstances[id].ReturnOffsets[register_index];
    // Read the registers
    *(addr) = rvex_read_mem(RVexInstances[id].DataMemAddress, offset);
}

Listing 5.24: Read a return value from ρ-VEX data memory

5.5.6 Molen Execute implementation

To start executing an application on a configured ρ-VEX core only the start signal has to be sent. This is a basic PLB bus transfer as shown in the following code.

/∗
  * Start execution on the rvex core
  *
  * @param id The id of the rvex core
  */
void molen_EXECUTE(u32 id) {
    rvex_start(RVexInstances[id].BaseAddress);
}

Listing 5.25: Starting execution on the specified ρ-VEX core

5.5.7 Molen Break implementation

When a ρ-VEX core finishes execution the wrapper generates an interrupt signal for the µBlaze. This interrupt is handled on the host, by fetching the status information from the ρ-VEX core. When the Molen Break function is called it does not need to poll the ρ-VEX core status over the PLB bus, but instead it can check the local status information of the specific ρ-VEX core. After the interrupt handler has updated the status upon receiving the finish interrupt the Break function will detect this and will return.

/∗
  * Wait until the rvex core has finished execution
  *
  * @param id The id of the rvex core
  */
5.6. **CONCLUSION**

During this chapter we have presented the implementation of our platform, as designed in the previous chapters. We started by implementing the hArtes $\mu$Blaze compiler, discussing the dependencies, scripts and data-flow. After the compiler was implemented the wrapper could be implemented, the main function of the wrapper is to interpret the bus signals and execute the commands on the $\rho$-VEX core. With a functioning wrapper the changes to the $\rho$-VEX processor could be implemented. The control and status signals could simply be connected internally, but the memories required some more work. Each of the memories were individually implemented, to meet the requirements specific for each of the three memories.

```c
/* void molen_BREAK(u32 id) { 
 Xuint32 timeout; 
 RVexCtr * RVexInstance; 
 
 RVexInstance = & RVexInstances[id]; 
 
 /* Loop a maximum of RVEX_BREAK_TIMEOUT times before exiting. This value 
 * should be large enough that the timeout only happens on rvex failure 
 * and not due to long execution time 
 */ 
 for (timeout = 0; timeout < RVEX_BREAK_TIMEOUT; timeout++) { 
 
 /* Test if the status of the rvex has changed to done. This is 
 * updated when an interrupt has occurred 
 */ 
 if (RVexInstance->Status & RVEXDONE) { 
 
 return XST_SUCCESS; 
 
 } 
 
 }
```

Listing 5.26: Break loop, waiting until the interrupt handler updates the status information
Experimental Results

In this chapter we present the results of the experiments performed with our platform. In Section 6.1, we start by giving an overview of the used platform and configurations. Section 6.6 presents the resource utilization for different components of our platform. Subsequently, in Section 6.3 the communication delay is determined. Section 6.5 discusses the application used to perform our tests and the results obtained from these tests. In Section 6.6 presents the steps required to implement the an application on our platform. Finally, in Section 6.7 the conclusions for this chapter are presented.

6.1 Experimental Setup

The system was targeted at the Xilinx Virtex-VI FPGA on the ML-605 board. This offered plenty of area to test different configurations. The wrapper and ρ-VEX core are implemented in VHDL and were simulated using the Mentor Graphics ModelSim SE 6.5e software. To configure the μBlaze processor and Processor Local Bus we used the Xilinx EDK version 12.4. The EDK software is also used to synthesize the entire system. The resulting bitstream is imported into the Xilinx Software Development Kit (XSDK). Within the XSDK the test applications are written and the tests are executed on the FPGA platform.

A single platform has been used to execute the test applications. A summary of the setup is shown in Table 6.1. This platform consisted of a μBlaze processor with two caches for the data and instruction memories. A Processor Local Bus (PLB) of 32 bits wide connects two 4-issue ρ-VEX processors with the μBlaze. To be able to measure the wrapper resource utilization, a ρ-VEX wrapper is added that does not contain a ρ-VEX core.

Additionally a timer is utilized to measure the cycles required for a given task. This timer is provided in the Xilinx toolchain and resides on the PLB bus. It acts as a cycle counter that generates an interrupt when the counter overflows. The 32 bits counter

<table>
<thead>
<tr>
<th>Component</th>
<th>Clock Freq.</th>
<th>Memory</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>μBlaze</td>
<td>100MHz</td>
<td>256 MB SDRAM</td>
<td>2 × 8kB Caches</td>
</tr>
<tr>
<td>Processor Local Bus</td>
<td>100MHz</td>
<td></td>
<td>32 bits wide</td>
</tr>
<tr>
<td>ρ-VEX 1</td>
<td>100MHz</td>
<td>256 kB Data and 128 kB Instr</td>
<td>4-issue core</td>
</tr>
<tr>
<td>ρ-VEX 2</td>
<td>100MHz</td>
<td>256 kB Data and 128 kB Instr</td>
<td>4-issue core</td>
</tr>
<tr>
<td>ρ-VEX wrapper</td>
<td>100MHz</td>
<td></td>
<td>No a ρ-VEX processor</td>
</tr>
</tbody>
</table>

Table 6.1: System configuration
overflows after about 43 seconds when run at 100 MHz. Some of our test applications require longer run time, so the \( \mu \)Blaze counts the number of overflow interrupts received from the timer. Combining the overflow count with the timer results in an accurate cycle count, able to count cycles for more than 3 hours. To start and stop the counter a bus transaction is required. The cycles required for these timer transactions are added to the measured cycles. By starting and directly stopping the timer a cycle count of more than 90 cycles is obtained. Each measured event should be significantly larger than the 90 cycles introduced by the timer. This will reduce the error introduced by the timer transactions.

### 6.2 Resource Utilization

Following the synchronization of the system described in the previous section the resource usage of different components could be determined. Table 6.3 presents the results of the measurements. The implemented version of the \( \rho \)-VEX core has large data and instructions memories, this is clear from the 96 utilized BRAMs. With a total of 416 BRAMs available on the ML-605 board this limits the number of \( \rho \)-VEX cores that can be implemented in a single system. When applications are executed that require less memory the memories could be reduced in size. Reducing the memory sizes would allow more \( \rho \)-VEX instances in a single system.

Comparing the resource utilization for the \( \rho \)-VEX wrapper and the \( \rho \)-VEX core shows that the resource utilization for the wrapper is low. Only 5 percent of the Look Up Tables (LUTs) used in the complete \( \rho \)-VEX system is utilized by the wrapper. A larger percentage, 21 percent, of the Flip Flops are utilized by the wrapper. But because the total amount of Flip Flops of the \( \rho \)-VEX system is only 0.5 percent of the available Flip Flops, this is not an issue. Additionally by removing the registers used for debugging from the wrapper the amount of utilized Flip Flops could be reduced.

### 6.3 Transfer speeds

The \( \rho \)-VEX cores lack direct access to the external memory. As a result all data needs to be transferred to a \( \rho \)-VEX processor through the \( \mu \)Blaze and PLB. Especially when large amounts of data need to be transferred this can influence the speed-up significantly. To determine the impact of the data transfers we have measured the cycles required to transfer data to and from a \( \rho \)-VEX processor. To measure the write transfer time we transferred 10000 words to a \( \rho \)-VEX processor. Sending a fixed set of data words,
eliminating data cache misses on the μBlaze, a cycle time of 8 is obtained for a single transfer. When actual data from memory is transferred the cycle count increases to 11 cycles per write transfer. This shows the cost of a cache miss on the μBlaze to be 3 cycles when averaged over a large data set. The read delay is measured by repeating the tests, but now reading the data from the ρ-VEX core. Reading the data without storing it on the μBlaze results in a read cycle time of 10 cycles. When the read data is stored in memory the cycle time increases to 14 cycles. As was the case with the write transactions, reading data and storing the results increases the latency. Additionally it is clear that read transactions require more cycles than write transactions. One extra cycle is introduced by the ρ-VEX memories. Writing on the ρ-VEX is done in a single cycle, but reading requires two cycles.

### 6.4 Functionality tests

In an effort to test the different modes and configurations that can be utilized we developed a test bench. The main target for this test bench is to test the hArtes compiler and MOLEN functions. Two ρ-VEX applications are utilized during the test bench. These two applications were adpcm and soma. The adpcm application encodes and decodes a sample signal and validates the results. Whilst the soma application recursively executes multiple additions on an array. Both applications are extensively used during the development of the ρ-VEX core. As a result both applications are known to execute correctly on the ρ-VEX processor. The test bench executes the following scenarios, simulating different platform configurations.

1. Both applications are executed on the μBlaze processor.
2. One application is executed on the μBlaze and the other on the ρ-VEX processor.
3. Both applications are executed on a ρ-VEX processor.

With the exception of the first scenario, the tests were performed both running the applications sequentially and in parallel. The first scenario was not tested in parallel, because the μBlaze is not capable of exploiting parallelism. Each test was repeated a thousand times. This was done for multiple reasons. First this simulates a more realistic utilization. The platform is intended to execute kernels which will often be repeated many times during an application. Secondly, this reduces the relative error introduced by the timing logic. Additionally, the multiple executions reduces the ρ-VEX

<table>
<thead>
<tr>
<th>Action</th>
<th>Data type</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>Fixed</td>
<td>8</td>
</tr>
<tr>
<td>Write</td>
<td>Memory</td>
<td>11</td>
</tr>
<tr>
<td>Read</td>
<td>Fixed</td>
<td>10</td>
</tr>
<tr>
<td>Read</td>
<td>Memory</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 6.3: Transfer delays to read and write from a ρ-VEX memory
reconfiguration costs when the application is run consecutively on the same core. Since only on the first execution the instruction and data memory need to be uploaded.

![Figure 6.1: Cycles required for the different actions](image)

To get a baseline each application is also run separately on both the µBlaze and ρ-VEX processors. This helps to get an indication of how the combined cycle counts are obtained. Additionally the applications were issued on a ρ-VEX, but without waiting until execution of the application was finished. The cycle count obtained from this test represents the overhead introduced when an application is executed on a ρ-VEX processor.

Shown in Figure 6.1 are the results of these first tests. Both applications benefit form being run on a ρ-VEX processor, gaining a speed-up of 1.3 in both cases. The obtained speed-up is low, mainly because the execution time for both applications is low. A single iteration of the adpcm kernel takes 616 cycles, and the soma kernel requires 712 cycles. As a result of the short executions times the overhead of the MOLEN functions limits the speed-up. Especially with the soma application the MOLEN overhead has a big impact, it is responsible for 42 percent of the cycles. The overhead is much lower with the adpcm application, just 15 percent. The difference in overhead is a result of the required in input and output data for the applications. Soma receives 20 words of arguments and creates 20 words of results. On the other hand, adpcm only creates a single word of result data.

While testing the different scenarios an bug was found in the hArtes compiler. To allow parallel execution the OpenMP API [5] is utilized. This is an API build into the compiler and can be used to instruct the compiler to schedule tasks in parallel. When used with MOLEN annotations the MOLEN functions will be issued concurrently, as shown in the following code example.
6.4. FUNCTIONALITY TESTS

```c
/* OpenMP code */
#pragma omp parallel sections
{
    #pragma omp section
    {
        #pragma map call_hw RVEX 0
        r = adpcm();
    }
    #pragma omp section
    {
        #pragma map call_hw RVEX 1
        r2 = soma(values);
    }
}
```

Listing 6.1: OpenMP C code, parallel execution of two ρ-VEX applications

```c
/* Configure the two r-VEX cores */
molen_SET(0, adpcm_id);
molen_SET(1, soma_id);

/* Move the arguments to the r-VEX cores */
molen_MOVTXaddr(1, 0, values, 80, 1);

/* Start execution of both cores */
molen_EXECUTE(0);
molen_EXECUTE(1);

/* Wait until both cores have finished execution */
molen_BREAK(0);
molen_BREAK(1);

/* Move the results from the r-VEX cores */
molen_MOVFX(0, 0, &r1);
molen_MOVFXaddr(1, 0, values, 80);
molen_MOVFX(1, 1, &r2);
```

Listing 6.2: Parallel execution of two ρ-VEX applications

Due to the bug, that has been inherited from the original hArtes GCC compiler, the utilization of the OpenMP API is not possible. As a result we had to manually call the MOLEN functions in the correct order, instead of having the compiler generate the calls. Figure 6.2 presents the cycle times utilized by the different scenarios. The cycle times obtained during the serial scenarios all equal the sum of cycle times for the individual actions. This equals expectations, because the two applications are simply executed one after the other.

When the applications are executed in parallel the results are a bit more complex. Ideally the total cycle time is determined by the task requiring the most cycles. But the results show that this ideal situation is not obtained. The difference in cycle time is largely created by the MOLEN functions. These functions are always executed, in series, on the μBlaze processor. Only the execution of the kernel on the ρ-VEX core is executed
in parallel. When both \( \rho \)-VEX cores are utilized to highest speed-up is obtained. This speed-up equals 1.8.

![Figure 6.2: Cycles required for the different scenarios](image)

### 6.5 Test application

To test our platform we have used the MindTCT application. This application is able to extract identification information from a fingerprint image. A kernel containing a Discrete Fourier Transformation (DFT) and some other transformations was implemented on the \( \rho \)-VEX processor. Each iteration the DFT kernel receives \( 34 \times 34 \) pixels of the fingerprint image. The result calculated by the kernel is 64 bytes of power signatures. These power signatures are used to determine the dominant direction flow in the image block.

```c
void sum_rot_block_rows2(const unsigned char *blkptr, ufp27p5_t **powers)
```

Listing 6.3: Kernel function definition

Executing the MindTCT application proved problematic, because some bugs were encountered. By running extensive test on the transferred data, it was concluded the bugs were caused by the \( \rho \)-VEX core and toolchain. Our \( \mu \)Blaze compiler and the MOLEN function were working correctly. As a result the application would run correctly, but the generated results were incorrect. To ensure the correct flow of the application the result read from the \( \rho \)-VEX is overwritten by a value known to be correct. Table 6.3 presents the results obtained from executing the test application.

The first thing that becomes apparent is the impact of (re)configuring a \( \rho \)-VEX processor. The used kernel has a large amount of data and instructions that need to be configured. A total of 392 instructions are transferred, and each instruction contains four words. Additionally to the instructions 9408 data words are required. This data mainly consists of lookup tables. As a result 10978 words need to be transferred when configuring the \( \rho \)-VEX processor. With only a single application executed on the \( \rho \)-VEX
6.6 Platform Usage

One of the main focus points for our platform was ease of use. No simple representation of the ease of use can be measured. Instead of a measurement we will describe the steps taken to implement our test application, as described in the previous section.

1. Generate the application header file. This header file contains the data and instructions for the ρ-VEX processor. Using the elf2vhdl tool this header file is automatically generated after the application is compiled.

2. In the generated header file a unique ID has to be specified to identify the application.

3. The offsets for the variables need to be specified. These offsets can be obtained by viewing the object dump for the application.

4. Inside the main application the application defined in the header file has to be configured.

5. Either the function declaration or the function call for the accelerated function needs to be annotated.
As becomes clear from these steps no knowledge of the actual platform and communication scheme is required. This greatly increases the ease of use. Additionally during the thesis the platform has already successfully been used by Bachelors students. This is another indication the platform can be used with limited knowledge of the hardware organization.

6.7 Conclusion

This chapter presents the results of the benchmarks we have executed. These results were obtained on a Xilinx Virtex-VI ML-605 board, with both the $\rho$-VEX and $\mu$Blaze processors running at 100 MHz. We compared the resource utilization for the $\rho$-VEX wrapper with the utilization for a complete $\rho$-VEX system. The wrapper utilizes 21 percent of the used Flip Flops. This is a relatively high amount, but when compared to the total available Flip Flops this is only 0.1 percent. LUT utilization for the wrapper is 5 percent of the total LUT utilization by a $\rho$-VEX system.

We have implemented the MindTCT fingerprint identification application on our platform. A DFT kernel was selected to be accelerated on the $\rho$-VEX processor. By comparing the cycle time required by the $\mu$Blaze and $\rho$-VEX processors a speed-up of 1.6 was obtained.

Additionally a test bench was created to simulate different utilization scenarios. A maximum speed-up of 1.8 was obtained when two $\rho$-VEX cores were utilized in parallel. The results from the test bench show the impact the MOLEN functions have on the gained speed-up. With the exception of the OpenMP API the test benches were all executed successfully.

Tests to determine the transfer speeds were also executed. The latency for a single write to a $\rho$-VEX memory ranges from 8 to 11 cycles. Reading from a $\rho$-VEX memory costs from 10 up to 14 cycles. Finally, the steps required to implement the test application on our platform were presented. These steps show that no knowledge on the utilized hardware platform is required to implement an application.
Conclusions

In this chapter we present a summary of this thesis. Furthermore, we discuss the contribution made during the thesis and we will suggest future work. The first section of this chapter, Section 7.1 presents a summary of this thesis. Following the summary, Section 7.2 discusses the main contributions made during this thesis. The final section, Section 7.3 presents our recommendations for future work.

7.1 Summary

In this thesis we have presented the research and implementation of a polymorphic processor containing a µBlaze processor and one or more ρ-VEX processors. These processors are interconnected using the Processor Local Bus and can be targeted used the MOLEN paradigm. The purpose of the designed system is to provide an adoptable platform offering a complete toolchain that can easily target the processors. During the design the focus has been to create a future proof platform that can incorporate different processors and bus organizations.

We started with the design of the compiler in Chapter 3. Two MOLEN implementations currently exist, the original implementation targeted at the PowerPC processor and a new implementation targeted at the ARM processor. We determined that newer implementation, the hArtes implementation, of the MOLEN paradigm fits our requirements best. With hArtes no platform specific Arbiter and ρµ-code unit have to be implemented, additionally no changes to the Instruction Set Architecture are required. Instead of these hardware changes, the logic required for the MOLEN paradigm is executed on the host processor using MOLEN function calls. These MOLEN functions are generated by the compiler and replace the MOLEN specific instructions that were originally used. To map a kernel in the source code onto an accelerator annotations are used. Specifically pragma statements are used as annotations. The compiler will filter out the annotated code and insert the different MOLEN function calls to configure and execute that kernel on an accelerator. To implement the MOLEN functions into the µBlaze compiler different aspects of the compiler had to be adopted. The front-end of the compiler, tasked with interpreting the source code supplied by the user has to be extended to handle the new annotations. Within the front-end, the map call_hw pragma statement is checked and filtered out. The source code following the pragma is wrapped in a dummy tree, so it can be handled later on in the process. During the middle-end of the compiler these dummy trees are detected and are converted into MOLEN function calls, handling the data transfers, set-up and control. Apart from the source code the compiler uses an architecture description file to define and validate the available accelerators. We have extended the architecture file to be able to define the ρ-VEX cores and the available programs for these processors.
In Chapter 4, we discussed the design of the $\rho$-VEX bus wrapper and the corresponding changes required in the $\rho$-VEX processor core. First a bus specification is selected which will be used to interconnect the $\mu$Blaze and $\rho$-VEX processors. Requirements for the bus are a high throughput and scalability. To reduce the overhead of the data and instruction transfers the throughput has to be high. To be able to use multiple $\rho$-VEX cores and other accelerators in a single system the bus has to be scalable to accommodate these units. From the buses available on the $\mu$Blaze processor the Processor Local Bus is the preferred option. It meets the requirements and because the bus is also available on PowerPC processors offers backwards compatibility to older FPGA platforms. By applying the bus and MOLEN related logic into a wrapper a bus independent core can be designed. As a result the different versions of the $\rho$-VEX core can easily be integrated into our system. Additionally, changing the used bus protocol does not require changes inside the $\rho$-VEX core. The interface contains signals that control and monitor the status of the $\rho$-VEX core. With a single memory interface the local memories residing in the $\rho$-VEX system can be written and read. By defining a standard memory interface the wrapper becomes independent of the memory implementation used in the $\rho$-VEX core. Again this aids in creating a flexible system, as a single wrapper can be used irrespective of the $\rho$-VEX memory organization.

In Chapter 5, the implementation of the designed system is presented. A bottom-up implementation strategy was used to increase testability and decrease implementation complexity. As a starting point for the compiler the original $\mu$Blaze GCC compiler is taken. By adopting the hArtes functionality and inserting this into the compiler a we obtain a $\mu$Blaze targeted compiler with the MOLEN functionality build in. Both the front-end and middle-end of the compiler have to be adopted to incorporate the new functions.

Implementing the wrapper started with a very basic bus slave. With the basic communication tested a start was made on the interface with the $\rho$-VEX processor. The control and status signals were defined and implemented in the wrapper. Following the wrapper the $\rho$-VEX core was adopted to allow external control and return the internal status. This resulted in a basic bus slave that could execute a predefined program and returned a done signal when execution was finished. Now the memory access remained to be implemented.

The memory interface between the wrapper and the $\rho$-VEX core matches the common memory interface used, so the wrapper mainly just forwards the incoming requests to the $\rho$-VEX core. Within the $\rho$-VEX each request is mapped to the correct memory and because all memories have a different organization each memory request is then handled separately. The status memory is not a real memory, but instead it forwards internal registers when addressed. This removes the need to duplicate the information and can easily be expanded to hold more data when needed. The Data memory is implemented as standard Block RAM (BRAM), with two read ports and a single write port. The access to the write port is shared by the wrapper and the $\rho$-VEX core. Since the wrapper is not allowed to change the data during execution on the $\rho$-VEX core sharing the single write port does not introduce any conflicts. In the original design of the $\rho$-VEX an UART unit was used to transmit the data memory contents after execution had finished. With the availability of memory access through the PLB this UART unit has become redundant.
and can be removed. The removal frees one of the two read ports on the data memory, and this port is used for read requests from the wrapper. Implementing the Instruction memory differs the different requirements for the $\rho$-VEX core and $\rho$-VEX wrapper need to be taken into account. The $\rho$-VEX core reads complete instructions, containing a word for each issue. On the other hand, the $\rho$-VEX wrapper reads and writes using word-addressing, irrespective of the number of issues. The resulting instruction memory is implemented in BRAM and contains two different read ports, fitted to either the $\rho$-VEX core or wrapper. Only a single write port is implemented, because the only the $\rho$-VEX wrapper is capable of writing to instruction memory.

Once the wrapper and $\rho$-VEX processor were finished the MOLEN functions remained to be implemented. These functions handle the MOLEN logic on the host processor, such as the configuration and data transfer. Novel in on our platform is the ability to transfer instructions to an existing accelerator. This is done by recognizing the accelerator type in the MOLEN set instruction and changing the instruction memory when required.

Chapter 6 presented the experimental setup and results. A test platform was implemented and synthesized to measure the resource utilization and perform benchmarks. All the test were performed on the XILINX Virtex-VI ML-605 FPGA board. The resource utilization by our wrapper was determined by implementing a wrapper without a $\rho$-VEX core. This wrapper utilizes 21 percent of the Flip Flops used by a complete $\rho$-VEX system. Only 5 percent of the utilized Look Up Tables (LUTs) is required by our wrapper. When compared to the total available Flip Flops and LUTs the resource utilization decreases to 0.1 percent and 0.3 percent. These low percentages show the small impact on resource utilization by wrapper, and allows multiple $\rho$-VEX systmes to be implemented on a single platform. The main limiting factor is the number of available Block RAMs (BRAMs). Both the instruction and data memories are implemented using these BRAMs. Our test application requires a large amount of data memory, which results in high BRAM utilization. Each $\rho$-VEX system uses 96 BRAMs, and with only 416 BRAMs available on the FPGA, this limits the number of $\rho$-VEX processors that can be implemented. Due to the parametrization available in the $\rho$-VEX core, the size of the utilized memories can easily be adopted. When smaller memories suffice the BRAM utilization would decrease, which would allow more $\rho$-VEX systems to be implemented on a single platform.

By measuring the cycles required to transfer data to and from a $\rho$-VEX processor, the effects of the transactions can be determined. Four different situations were tested, writing and reading both fixed data and data from memory. A large data set of 10000 data words is used to average out any external influences. By reading and writing fixed data cache misses are eliminated, thus this gives the lowest transaction latencies. When data is used located in memory the effects of cache misses is also measured. Writing fixed data was measured to take 8 cycles, whilst reading fixed data costs 10 cycles. Once the data is retrieved or stored into memory these cycle counts increase with three to four cycles.

To test the platform a test bench utilizing two $\rho$-VEX kernels was implemented and executed. By issuing the two $\rho$-VEX kernels to either the $\mu$Blaze or $\rho$-VEX processor different scenarios were created. Our platform was able to successfully execute the test
bench, with the exception of a single bug. The OpenMP API utilized to issue tasks in parallel could not be used. This was the result of a bug inherited from the original hArtes compiler. Until the bug is fixed the problem can be circumvented by manually calling the MOLEN functions to execute the tasks in parallel. A maximum speed-up of 1.8 was obtained when the kernels were executed in parallel on separate $\rho$-VEX cores. The results clearly showed the impact the MOLEN functions have on the available speed-up. All the MOLEN functions are executed sequentially on the MicroBlaze. This limits the gain from parallel execution.

Additionally a second test was performed. During this test a DFT kernel taken from the MindTCT fingerprint application is utilized. By comparing the execution time when running the kernel on $\mu$Blaze and $\rho$-VEX, the speed-up can be determined. The initial configuration of the $\rho$-VEX core is costly, mainly because the DFT kernel uses a very large amount of data memory. After the initial configuration the execution time of the kernel is greatly reduced, because no reconfiguration is required. Running the kernel on $\mu$Blaze requires $604 \times 10^3$ cycles. Our 4-issue $\rho$-VEX core only requires $406 \times 10^3$ cycles, this includes the transfer of the input and result data. This shows that a speed-up of 1.6 was obtained by utilizing the $\rho$-VEX processor.

When implementing an application on our platform only a few steps need to be executed. These steps mainly create and configure the application that is executed on the $\rho$-VEX processor. No knowledge of the hardware organization is required during the implementation and execution, this greatly improves the ease of use when using our platform.

7.2 Main Contributions

In this section the main contributions made within this thesis are highlighted.

- **hArtes Compiler:** a new implementation of the hArtes compiler has been created. This compiler is targeted at the Xilinx $\mu$Blaze processor and has the added functionality to configure $\rho$-VEX processors as accelerators.

- **$\rho$-VEX wrapper:** we have developed a wrapper that allows full control and monitoring of a $\rho$-VEX processor over the Processor Local Bus.

- **$\rho$-VEX interface:** Designed a new $\rho$-VEX interface allowing external control over the execution state and local memory contents.

- **Performance analysis:** Benchmarking the $\rho$-VEX processor as a MOLEN accelerator. Comparisons between running the application on a $\mu$Blaze General Purpose Processor and the $\rho$-VEX VLIW processor were performed.

7.3 Future Work

We have the following recommendations for future work.
7.3. *FUTURE WORK*

1. Within the new compiler code targeting a $\rho$-VEX processor is discarded by the hArtes compiler. Here lies the opportunity to extract these code sections and automatically compile these for the target. This eliminates the need to pre-compile and define the applications that are executed on a $\rho$-VEX processor.

2. Improve integration of the application header file generation. This would eliminate manual changes required to configure the application ID. Additionally the memory offsets for the arguments and results would not need to be manually determined and configured.

3. Shared memory directly available on the $\rho$-VEX cores, this would eliminate most of the data transfers between the host processor and the $\rho$-VEX core.

4. A dedicated bus device responsible for configuring the $\rho$-VEX cores. The host processor would instruct this unit to reconfigure a specific $\rho$-VEX processor with a certain application. This would free the host processor to continue execution concurrently whilst a $\rho$-VEX is reconfigured. Additionally this reconfiguration unit could implement burst transactions, which would speed-up the reconfiguration process.

5. If the size of the Instruction Memory is large enough, one or more applications could be loaded simultaneously into a single $\rho$-VEX core. This would eliminate the need to load these applications when they are the only applications that are executed on that core.
Bibliography


77


This appendix contains the manual describing the configuration and utilization of different aspects of our platform.

A.1 Creating the hardware bitstream

To create the bitstream containing the hardware platform the Xilinx XPS toolchain is used. This toolchain handles the configuration of the $\mu$Blaze processor and the connections with the $\rho$-VEX processor. All steps described here are based on the version 12.4 of the XPS toolchain. The minimal requirements on the hardware configuration are a $\mu$Blaze processor and a $\rho$-VEX processor connected through a Processor Local Bus (PLB). Both processors and the PLB need to operate at the same clock frequency. Currently the clock frequency is limited by the $\rho$-VEX processor to about 100 MHz. Table A.1 describes the standard configuration for the $\mu$Blaze used during this thesis.

A.1.1 Add $\rho$-VEX core(s)

To add a $\rho$-VEX processor to the XPS project a new peripheral needs to be added. Each type of $\rho$-VEX processor needs to be defined as a separate peripheral in the XPS project. When multiple identical $\rho$-VEX processors are used these can be created from a single peripheral. The first step is to create a new peripheral using the wizard. This peripheral needs to be configured using the settings described in Table A.1.1. After the peripheral is generated the $\rho$-VEX wrapper and core files need to be copied to the source folder. This folder is located in XPS_PROJECT/pcores/rvex_plb_wrapper_v1.00a/.

Once the peripheral is created an instance can be generated in the system. Ensure that the correct size is specified for the memories under Addresses. The data memory is available at MEM0 and the instruction memory is available under MEM1. Connect the $\rho$-VEX instance to the PLB bus and connect the interrupt ports. The interrupts are connected as follows. Each interrupt coming from the $\rho$-VEX instances should be added to the xps_intc_0 device on the Intr port. The Irq port exiting the xps_intc_0 device needs to be connected to the $\mu$Blaze interrupt port.

<table>
<thead>
<tr>
<th>Single core</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 MHz clock frequency</td>
</tr>
<tr>
<td>Processor Local Bus</td>
</tr>
<tr>
<td>Caches enabled</td>
</tr>
</tbody>
</table>

Table A.1: $\mu$Blaze configuration
At this point the hardware can be synthesized and the resulting bitstream can be exported to the Xilinx SDK tool. When updating an existing hardware bitstream check that the export has actually replaced the old bitstream, this is not always the case.

A.2 Create the \( \rho \)-VEX Kernel

To create the kernel software that will be executed on a \( \rho \)-VEX processor the following steps need to be taken. First a \textit{void main(\textit{void})} function has to be created, this function will call the original kernel function. Each of the arguments and return for the kernel function values have to be placed in globally defined variables. A small example is given below.

```c
int a;
int b;
int result;

/* The kernel function */
int vex_add(int c, int d) {
    return c + d;
}

/* The main function */
void main(void) {
    result = vex_add(a, b);
}
```

The resulting source code can then be compiled and linked. To generate the header file used by the software toolchain the elf2vhdl tool has to be used. This tool will generate two vhdl files, which can be discarded, and a \textit{prog.h} header file.

```bash
elf2vhdl vex_add
```

In addition to the header file the memory locations for the arguments have to be determined. This is done by reading the object dump.
A.3. Setup the hArtes compiler

When a pre-compiled version of the compiler is available only a single step is required. The path containing the executables has to be added to the PATH environment variable.

```bash
PATH=/data/hgcc_mb/release/lin/bin:/$PATH
export PATH
```

To compile the hArtes compiler the two build scripts need to be executed. First the binutils have to be created, this is done by executing `build_binutils.sh`. When this has successfully finished the compiler can be created by executing `build_gcc.sh`. If successful the binaries are created in `release/lin(64)/bin/`. Logs created during the build are available in `build/lin(64)/gcc/logs`.

A.4 Configure the Xilinx SDK project

To setup the XSDK the hardware needs to be exported from the XPS toolchain. This is done by exporting and launching the Xilinx SDK from XPS. Open XPS and go to `Project -i; Export hardware design to sdk`. Include bitstream and BMM file and then `Export & Launch SDK`. This step needs only be done when new hardware is created, if the hardware platform is unchanged the XSDK can directly be opened by executing `xsdk`.

A.4.1 hArtes compiler configuration

By default the `XSDK` toolchain will invoke the standard µBlaze GCC compiler. When the hArtes compiler should be used the following settings need to be changed for each project.

1. Right click on the project and select `C/C++ Build Settings`.
2. Select `MicroBlaze gcc compiler`.
3. Change the command into `mb-hgcc`.
4. Select `Miscellaneous`.
5. Add the flags: 
   ```
   -fhartes-configuration-file=hartes.xml -ffpga -ffpga-pass-debug-level=13
   ```
6. Click `OK`
A.4.2 Application configuration

In the C application a few things need to be configured. The hArtes configuration file hartes.xml has to be placed in the folder src/. Each ρ-VEX instance needs to be set-up in software, the following code shows the set-up of two ρ-VEX instances.

```c
/* Initialize the interrupt controller */
initialize_intc();

/* Initialize the rvex instance 0 */
RVexInstances[0].BaseAddress = XPAR_RVEX_PLB.WRAPPER0_BASEADDR;
RVexInstances[0].DataMemAddress = XPAR_RVEX_PLB.WRAPPER0_MEM0_BASEADDR;
RVexInstances[0].InstrMemAddress = XPAR_RVEX_PLB.WRAPPER0_MEM1_BASEADDR;
RVexInstances[0].StatusMemAddress = XPAR_RVEX_PLB.WRAPPER0_MEM2_BASEADDR;
RVexInstances[0].Status = 0;
RVexInstances[0].ProgramID = -1;

/* Initialize the interrupts for rvex 0 */
rvex_init_interrupts(&RVexInstances[0], XPAR_XPS_INTC0_RVEX_PLB.WRAPPER0_IP2INTC_IRQT_INTR);

/* Initialize the rvex instance 1 */
RVexInstances[1].BaseAddress = XPAR_RVEX_PLB.WRAPPER1_BASEADDR;
RVexInstances[1].DataMemAddress = XPAR_RVEX_PLB.WRAPPER1_MEM0_BASEADDR;
RVexInstances[1].InstrMemAddress = XPAR_RVEX_PLB.WRAPPER1_MEM1_BASEADDR;
RVexInstances[1].StatusMemAddress = XPAR_RVEX_PLB.WRAPPER1_MEM2_BASEADDR;
RVexInstances[1].Status = 0;
RVexInstances[1].ProgramID = -1;

/* Initialize the interrupts for rvex 1 */
rvex_init_interrupts(&RVexInstances[1], XPAR_XPS_INTC0_RVEX_PLB.WRAPPER1_IP2INTC_IRQT_INTR);
```

In addition to each ρ-VEX instance the header files containing the kernels should be defined. First include the header files and then define the programs as follows.

```c
RVexPrograms[vex_add_.fpga.text_] = RVEX_PROG_vex_add;
```

At this point the configuration is complete and the platform is ready to be used. Add the annotations to the function calls and/or declarations to execute these functions on a ρ-VEX processor. The following code is a simple example of such an annotation.
int main() {
    int r_local, r_remote_0, r_remote_1;

    /* Run on the MicroBlaze */
    r_local = vex_add(8, 13);

    /* Run on rvex instance 0 */
    #pragma map_call hw RVEX 0
    r_remote_0 = vex_add(8, 13);

    /* Run on rvex instance 1 */
    #pragma map_call hw RVEX 1
    r_remote_1 = vex_add(8, 13);
}

The following Architecture Description File is used by the hArtes compiler. This file can be used as is for platforms containing at most two $\rho$-VEX instances.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE ORGANIZATION SYSTEM "architecture.dtd">

<ORGANIZATION>
  <HARDWARE>
    <FUNCTIONAL_COMPONENT>
      <NAME>RVEX</NAME>
      <IDGEN>4000</IDGEN>
      <TYPE>FPGA</TYPE>
      <MASTER>NO</MASTER>
      <FREQUENCY>100</FREQUENCY>
      <DATA>
        <MAXSTACKSIZE>10240</MAXSTACKSIZE>
        <DATA_TYPE>
          <NAME>char</NAME>
          <PRECISION>8</PRECISION>
        </DATA_TYPE>
        <DATA_TYPE>
          <NAME>int</NAME>
          <PRECISION>16</PRECISION>
        </DATA_TYPE>
        <DATA_TYPE>
          <NAME>short</NAME>
          <PRECISION>16</PRECISION>
        </DATA_TYPE>
        <DATA_TYPE>
          <NAME>float</NAME>
          <PRECISION>32</PRECISION>
        </DATA_TYPE>
        <DATA_TYPE>
          <NAME>long</NAME>
          <PRECISION>32</PRECISION>
        </DATA_TYPE>
      </DATA>
    </FUNCTIONAL_COMPONENT>
  </HARDWARE>

  <OPERATIONS>
    <!-- Dummy operation used to configure the rvex cores -->
  </OPERATIONS>
</ORGANIZATION>
```
<NAME>RVEX_CORES,DUMMY</NAME>
<COMPONENT>
  <NAME>RVEX</NAME>
  <!-- Each instance needs an implementation with valid ID -->
  <IMPLEMENTATION>
    <!-- Core ID number -->
    <ID>0</ID>
    <SIZE>100</SIZE>

    <START_INPUT_XR>0x00000000</START_INPUT_XR>
    <START_OUTPUT_XR>0x00000000</START_OUTPUT_XR>

    <SET_ADDRESS>0X00000001</SET_ADDRESS>
    <EXEC_ADDRESS>0X00000001</EXEC_ADDRESS>

    <SET_CYCLES>100</SET_CYCLES>
    <EXEC_CYCLES>60</EXEC_CYCLES>
    <FREQUENCY>100</FREQUENCY>
  </IMPLEMENTATION>
  <IMPLEMENTATION>
    <!-- Core ID number -->
    <ID>1</ID>
    <SIZE>100</SIZE>

    <START_INPUT_XR>0x00000000</START_INPUT_XR>
    <START_OUTPUT_XR>0x00000000</START_OUTPUT_XR>

    <SET_ADDRESS>0X00000001</SET_ADDRESS>
    <EXEC_ADDRESS>0X00000001</EXEC_ADDRESS>

    <SET_CYCLES>100</SET_CYCLES>
    <EXEC_CYCLES>60</EXEC_CYCLES>
    <FREQUENCY>100</FREQUENCY>
  </IMPLEMENTATION>
</COMPONENT>
</OPERATION>
</OPERATIONS>
</ORGANIZATION>
M. de Zeeuw was born on January 15, 1983 in Leiden, The Netherlands. From 1995 to 2001 he attended the Rijnlands Lyceum te Oegstgeest, where he obtained his HAVO and VWO diplomas. In September 2001 he started his Electrical Engineering studies at the Delft University of Technology. After obtaining the Bachelors of Science degree, he continued his studies at the Computer Engineering Laboratory where he hopes to receive the Master of Science degree.

His research interests include, but are not limited to: Embedded System Design, Reconfigurable Computing, Computer Architecture, Embedded Application Development and Microprocessor Architectures.

Besides his academic studies he likes to play sports, such as cycling, running and kite surfing. He enjoys remote control aircrafts, travelling and photography. During his studies he has worked as software developer and assistant project manager at Digitalization in Delft.