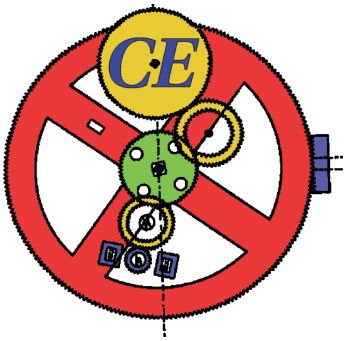


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2016

MSc THESIS

Implementing Virtual Address Hardware Support on the ρ-VEX Platform

Jens Johansen

Abstract

The ρ-VEX is a run-time reconfigurable Very Long Instruction Word (VLIW) processor. This unique processor allows separation of its issue lanes to form independently operating processing cores. Switching between these configuration during run-time allows optimizing the platform for the task(s) it is performing. Porting an Operating System (OS) to this platform is an important step towards a software layer that can control these reconfigurations.

All contemporary operating systems implement a virtual memory abstraction layer. This technique makes the physical memory layout transparent to the processes hosted by the operating system. Implementing virtual memory is impossible without hardware support. This thesis presents the design and implementation of the addition of this hardware to the ρ-VEX platform. The platforms unusual architecture puts specific requirement on the memory system. The implemented hardware fully supports the platforms abilities for static configuration and dynamic reconfiguration.

To verify that the implemented solution is able to support an OS, software is designed that emulates the task switching and memory virtualization tasks of an OS. By running this software it is proven that the hardware support all features required by a real OS.

Finally, the performance of the implemented hardware is measured running benchmarks in different static configurations and several dynamic reconfiguration scenarios. These measurements are compared and recommendations are made for performance optimization of real applications.

CE-MS-2016-05

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Implementing Virtual Address Hardware Support on the ρ-VEX Platform

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Jens Johansen
born in The Hague, The Netherlands

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

Implementing Virtual Address Hardware Support on the ρ-VEX Platform

by Jens Johansen

Abstract

The ρ-VEX is a run-time reconfigurable VLIW processor. This unique processor allows separation of its issue lanes to form independently operating processing cores. Switching between these configuration during run-time allows optimizing the platform for the task(s) it is performing. Porting an OS to this platform is an important step towards a software layer that can control these reconfigurations.

All contemporary operating systems implement a virtual memory abstraction layer. This technique makes the physical memory layout transparent to the processes hosted by the operating system. Implementing virtual memory is impossible without hardware support. This thesis presents the design and implementation of the addition of this hardware to the ρ-VEX platform. The platforms unusual architecture puts specific requirement on the memory system. The implemented hardware fully supports the platforms abilities for static configuration and dynamic reconfiguration.

To verify that the implemented solution is able to support an OS, software is designed that emulates the task switching and memory virtualization tasks of an OS. By running this software it is proven that the hardware support all features required by a real OS.

Finally, the performance of the implemented hardware is measured running benchmarks in different static configurations and several dynamic reconfiguration scenarios. These measurements are compared and recommendations are made for performance optimization of real applications.

Laboratory : Computer Engineering
Codenumber : CE-MS-2016-05

Committee Members :

Advisor:	dr. ir. Stephan Wong, CE, TU Delft
Chairperson:	dr. ir. Stephan Wong, CE, TU Delft
Member:	dr. ir. Arjan van Genderen, CE, TU Delft
Member:	dr. ir. René van Leuken, CAS, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

ASIC Application-Specific Integrated Circuit

ASID Address Space Identifier

BRAM Block RAM

CAM Content Addressable Memory

CE Computer Engineering

CLB Configurable Logic Block

CPU Central Processing Unit

DDR Double Data Rate

DSP Digital Signal Processing

EEMCS Electrical Engineering, Mathematics and Computer Science

FPGA Field-Programmable Gate Array

GRLIB Gaisler Research Library

HDL Hardware Description Language

ILP Instruction Level Parallelism

IC Integrated Circuit

IP Intellectual Property

ISA Instruction Set Architecture

ISR Interrupt Service Routine

LUT Lookup Table

MMU Memory Management Unit

NOP No Operation

NRE Non-Recurring Engineering

OS Operating System

PT Page Table

PTE Page Table Entry

PTP PT Pointer

PIPT Physically Indexed Physically Tagged

PIVT Physically Indexed Virtually Tagged

PC Program Counter

PTP Page Table Pointer

RAM Random Access Memory

RFI Return from Interrupt

SOC System-on-Chip

TLB Translation Look-aside Buffer

TLP Thread Level Parallelism

TW Table Walker

UART Universal Asynchronous Receiver/Transmitter

USB Universal Synchronous Bus

VEX VLIW Example

VHDL VHSIC HDL

VIPT Virtually Indexed Physically Tagged

VIVT Virtually Indexed Virtually Tagged

VLIW Very Long Instruction Word

Acknowledgements

There are a number of people that helped and supported me in the course of this project. First of all, I would like to thank my supervisor, Stephan Wong, for granting me the opportunity to work on this cool project. During the course of this project he helped me to maintain a high level view of the project but also helped me by discussing some low-level implementation issues.

I also want to express my thanks to my fellow students Klaas and Hugo who have been a lot of fun to hang out with the last year. We all started around the same time working on the ρ-VEX platform which gave us the really useful opportunity to reflect on each others projects.

Jeroen, thank you for rewriting the entire core and providing many useful tools. This gave me an excellent starting point for my implementation. Your skills and productivity puts the rest of us to shame.

Finally I would like to thank Joost and Anthony for guiding me through this process. You have spent a lot of time helping me with this project even though you had plenty of other stuff to do. Joost especially for proofreading my entire thesis report, which is much more than I would have dared to ask.

Jens Johansen
Delft, The Netherlands
February 5, 2016

1 Introduction

This thesis describes the design and implementation of hardware support for virtual memory on the ρ-VEX processing platform. In this chapter, the context of this project is explained and the target platform is introduced. Subsequently, the problem statement and the goals for this project are defined. In the last part of this chapter the structure of the rest of this report is outlined.

1.1 Context

Computers have shown an astronomic increase in computing power since the advent of the electronic computer in the first half of the 20th century. Since the transition to integrated circuits, technology scaling has been one of the main driving forces of this development. Shrinking transistors allowed increasingly more complex system to be realized while signal propagation delays decreased. This increase in raw processing power has made computer processors very power hungry devices. These devices dissipate so much energy that cooling has become a serious challenge.

This is one of the reasons why traditional processor architecture has shown diminishing returns in performance increase. One of the industries solutions was the move to multiprocessor designs. However, it has proven difficult to write programs that make efficient use of these resources. These circumstances push the development of new processing paradigms.

Reconfigurable computing is one of these paradigms. In a reconfigurable computing platform, some aspects of the hardware such as the processor or interconnection network can be changed. This allows adapting the system resources to a specific task while, or just before it is executed. Traditionally, implementations could either be implemented in silicon, which has the best performance but is static and can never be modified. Or alternatively, a software solution could be created, which is flexible but often orders of magnitudes slower. Reconfigurable computing tries to find a middle ground between these approaches, combining the performance of hardware while not completely fixating the implementation.

Liquid architectures is one of the main research topics of the Computer Engineering (CE) group. The ρ-VEX processor [3] is one of the active projects in this direction It was initially designed to be used a a co-processor in heterogeneous processing platforms such as the MOLEN machine [4]. However, after years of extending the ρ-VEX, it is now a experimental processing platform that is developed at its own. This thesis project is conducted in the context of the larger ρ-VEX project.

The ρ-VEX is a dynamically reconfigurable Very Long Instruction Word (VLIW) processor. It can separate its issue lanes which can form separate processing cores that can operate independently. Being able to switch between these configurations during

run-time allows the processor to adapt to its work load. The control layer responsible for this task is currently not yet developed. A logical place for such a control layer would be at the Operating System (OS) level. The hardware to fully support an OS such as Linux is not completely implemented in the ρ-VEX platform. More specifically, the hardware required for virtual addressing, an feature present in every contemporary OS, needs to be designed and implemented.

1.2 Problem Statement, Project Goals, and Methodology

In the development of the ρ-VEX platform, porting an OS such as Linux is an importing step towards maturity. All full-fledged contemporary OS's implements a virtual memory abstraction layer. It is impossible to implement virtual memory without hardware support. The problem statement for this thesis project therefore is:

How to implement hardware support for virtual memory on the ρ-VEX platform?

To answer this question, it is first necessary to investigate what solutions already exist and which are suitable for the ρ-VEX platform. Because the ρ-VEX is a experimental processor, it has properties which are not encountered in other systems. Its dynamic reconfigurable nature has implication for the memory subsystem. This requires designing novel hardware for the Memory Management Unit (MMU) which is able to support this feature. This leads to the first goal of this project:

(1) Designing and implementing memory translation hardware for the ρ-VEX platform.

To reach this goal, the following approach will be taken:

- Perform a literature survey to catalogue existing memory virtualization techniques and architectures.
- Identify how the ρ-VEX's unique features impact the design of the MMU.
- Select an architecture best suited for the ρ-VEX based on its target application, implementation platform, and its unique features.
- Design the interface of the virtual memory hardware to the the core and the cache.
- Design and implementation the hardware components and integrate them with the rest of the ρ-VEX system.

This project is an crucial step towards porting an OS to the platform. Therefore, it is essential that the hardware designed in the course of this projects meets the requirements posed by an OS. Because there is no OS port available to test with at this time, another method must be derived to ascertain it does. The second goal therefore is:

(2) Proving that the platform is able to support an OS which implements virtual memory.

The following methodology will be applied to reach the second goal:

- Compile a list of the bare minimum features the platform must posses to support an OS.
- Design and write software which emulates the function of an OS and relies on the MMU for the previously specified hardware support.
- Run the verification software on modified ρ-VEX system on the an Field-Programmable Gate Array (FPGA).

Because the ρ-VEX is currently the only dynamically reconfigurable VLIW processor, this thesis also describes the first implementation of an MMU for such a system. Therefore, an investigation will be made into how the performance of the virtual memory hardware is affected by dynamically reconfiguring the ρ-VEX. This leads to the final goal:

(3) Measuring how the implementation performs in different static configurations and dynamic reconfiguration scenarios

The steps toward this goal will be:

- Select or create benchmarks which intensively use the virtual memory hardware of the ρ-VEX system.
- Measure and compare the performance of different static configurations of the ρ-VEX MMU.
- Measure and compare the performance of the virtual address hardware in different dynamic reconfiguration scenarios.

1.3 Overview

In Chapter 2, background topics are discussed and related work is presented. Chapter 3 details the design of the MMU on a conceptual level and discusses the architectural decisions. In Chapter 4, the low-level implementations of the different components are explained. Chapter 5 tries to prove that the implemented hardware is actually able to support an OS. This is done by running software on the system that interfaces in a similar way with the hardware as an OS would. The chapter also evaluates the implementation cost in terms of area and performance penalty. Chapter 6 evaluates the implementation comparing performance and area increase. Additionally, a performance comparison is made between several configurations of the system. Finally, the chapter explores how the virtualization hardware performs in different dynamic reconfiguration scenarios. Chapter 7 summarizes the project, lists the main contributions, and suggests future work.

2 Background

In this chapter, some preliminary topics will be discussed that are related to the ρ-VEX Memory Management Unit (MMU). The goal of this chapter is to introduce or refresh some of the topics that are important in understanding the background and goal of the project.

In Section 2.1, Field-Programmable Gate Arrays (FPGAs) will be introduced which is the technology that servers as the platform for this project. The application domain within this platform is softcore processors. These will be explained in Section 2.2. Section 2.3 is about Very Long Instruction Word (VLIW) processors, the class the ρ-VEX processor falls in. Subsequently, in Section 2.4 the ρ-VEX itself and its unique features are discussed. In Section 2.5, an overview of the concept of memory management and different options for implementation are given. Finally, in Section 2.6, a list is compiled of the minimal set of functions that an MMU should posses to support an Operating System (OS).

2.1 FPGAs

An FPGA is a chip of which the functionality can be reconfigured after production or *in the field*. This technology allows designers to describe the desired behaviour of the chip using a Hardware Description Language (HDL) such as Verilog or VHSIC HDL (VHDL) in a similar manner to software design. This description is then synthesized by a software tool which converts the high-level description into a bit stream which is uploaded to the FPGA. The bit stream is a binary file that holds the setting for each configurable element in the chip. All configurable resources are configured and connected in such a way that the FPGA as a whole behaves similarly to the description in the HDL.

FPGAs fill a traditional gap between Application-Specific Integrated Circuits (ASICs) and software run on a general purpose processor. ASICs are the fastest solution for computationally intensive applications because they are specifically tailored and can employ a large degree of parallelism. However, their production is only viable in high volumes due to the expensive masks used in the photolithographic production process and the high Non-Recurring Engineering (NRE) cost. Software solutions, on the other hand, are relatively cheap and easy to implement but can be slow due to its sequential nature. FPGAs combine the parallel processing power of ASICs with the ease of implementation of software. There are of course some drawbacks to FPGAs since they have not replaced the alternatives. The reconfigurable nature of FPGAs incurs a high price in overhead regarding Integrated Circuit (IC) area utilization. Often designs take up to 20 times as much area as a functional equivalent ASIC implementation [5]. Another drawback is that because of the reconfigurable routing, path delays are larger than in an ASIC. This makes typical FPGA implementations an order of magnitude

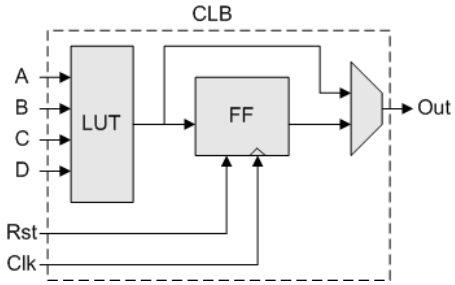


Figure 2.1: Configurable logic block

slower than their ASIC counterpart in terms of operating frequency. When comparing FPGAs to a solution implemented in software, the increased performance comes at the the price of more expensive ICs and higher development costs.

FPGAs are used often as a prototyping platform when designing ASICs. Because of the high cost of producing ICs, it is not possible to have multiple design cycles in silicon. Another application is in high performance embedded systems which have are produced in volumes which are to low for custom ASICs. Often a combination is made between a microprocessor running software accelerated by custom hardware implemented on an FPGA. This enables moving only the computationally intensive parts of the application to reconfigurable fabric while the rest can be implemented in software. This keeps the price of the solution low because a much smaller capacity FPGA is required. Hybrid ICs exist that combine reconfigurable fabric with a microprocessor. Alternatively it is also possible to implement the processor entirely in reconfigurable fabric. This is referred to as a softcore processor.

FPGAs consist of a large number of Configurable Logic Blocks (CLBs) on a die together with a reconfigurable interconnect. In their simplest form, a logic block consists of a Lookup Table (LUT) whose output is connected to a flip-flop. The LUT is basically a truth table that can be filled at when the FPGA is programmed. Depending on its contents it fulfils a certain combinational function on its inputs. The output is either synchronous or asynchronous depending on the whether the output flip-flop is bypassed. Figure 2.1 depicts the CLB described. The CLBs present in an FPGA are embedded in a programmable interconnect. This enables the output of any CLB to be connected to any other CLB on the chip. Combining the CLB allows creating much more complex combinatorial functions than a single CLB can perform. In Figure 2.2 the CLBs are shown together with the reconfigurable interconnect. When the flip-flops are also incorporated, a finite automaton can be created. Computability theory dictates that this can be used to solve any problem which is *computable* [6]. Since the introduction of FPGAs in the eighties the amount of transistors on silicon dies has risen significantly. This allowed manufacturers to include additional resources on their FPGAs such as Block RAMs (BRAMs) and multipliers. While these units can also be created from reconfigurable fabric, these types components are so commonly used that it makes sense to implement them statically, which saves die area and increases their performance.

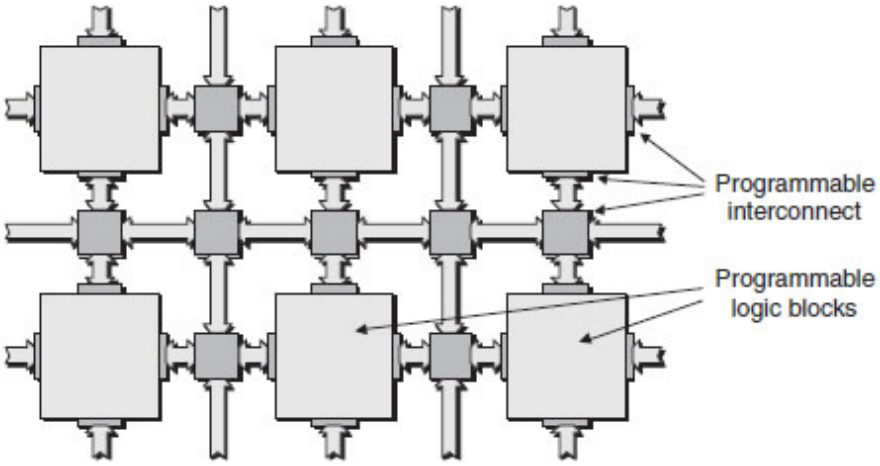


Figure 2.2: FPGA structure

2.2 Softcore Processors

As explained in Section 2.1, it is not always efficient to implement an entire application in reconfigurable hardware. Parts that are sequential in nature often do not benefit from implementing them on a FPGA. Combined with the relative high cost of FPGA ICs it is more effective to move only the most critical parts of the application to reconfigurable hardware. The rest of the application can be performed in software running on a processor. A general-purpose processor can be combined with the FPGA in one of three ways:

- A FPGA IC with a separate processor together on a circuit board.
- A hybric IC which combines a hard-wired processor with reconfigurable fabric.
- A FPGA IC where a part is configured as a softcore processor.

The advantage of the first two methods is that, because the processor is implemented directly in silicon, the operating frequency can be much higher than in a softcore processor. The last two methods share the advantage that the the core and the reconfigurable fabric are tightly coupled because they are on the same die. This enables high speed data transfer between them because the signals do not have to go off-chip. The softcore might seem like the inferior solution because it is inefficient in die size and it has a significantly lower operating frequency than the other two options. However, its power comes from its customizability. A softcore can be tailored to the specific application. For instance, custom instructions can be added if required, or an entire additional core can be added after a system has already been designed.

2.3 VLIW Processors

Processors are sequential machines by nature. They perform operations on data dictated by a stream of instructions. There are basically two approaches to speed up this pro-

cess. The operating frequency can be increased, decreasing the time to execute a single instruction Or alternatively, more instructions can be executed every clock cycle, raising the Instruction Level Parallelism (ILP).

A VLIW processor is a processor that uses compiler techniques to improve performance in the latter way. Contemporary general-purpose Central Processing Units (CPUs) are of the superscalar class. These types of processors try to increase the ILP by dynamically checking data dependencies and executing instructions concurrently and sometimes out of order. This process is referred to as *dynamic instruction scheduling*, which is a complex process and requires extensive hardware support. Because of the complexity and high cost of implementing such a system it is generally not used for softcore processors.

An alternative way is to have the compiler extract the ILP from the program, statically, at compile time. This moves the effort from the processor to the compiler. The major drawback of this approach is that programs need to be compiled specifically for a certain platform, thus impeding code portability. As a result of this, VLIW processors are most useful in an embedded system where the software and hardware are usually tailored to each other and the target application. VLIW (softcore) processors can achieve high performance for a low price especially in applications which have a high level op parallelism such as Digital Signal Processing (DSP) applications.

2.4 The ρ-VEX Reconfigurable VLIW Processor

The previous sections of this chapter have introduced the concepts of softcores and VLIW processors in general. Now the target platform for this thesis can be discussed, the ρ-VEX processor. ρ-VEX stands for reconfigurable VLIW Example (VEX) which refers to the ability of the processor to be statically configured and dynamically reconfigured. The precise meaning of these terms will be explained in the following subsections. The ρ-VEX processor is not the first VLIW softcore processor. The first appearance in literature of such a processor is the Spyder introduced in [7] in 1993. Other implementations are presented in [8] [9] [10]. All of these implementations suffer from one of the following drawbacks:

- It relies on a closed source compiler or processor design.
- Lack of good toolchain support.
- There is limited possibility for parametric customization or extensibility.

The ρ-VEX processor was introduced in [3] in 2008. The processor is entirely developed by students and Ph.D. candidates at the Computer Engineering group at the Electrical Engineering, Mathematics and Computer Science (EEMCS) faculty at the TU Delft. Currently the ρ-VEX system is in its third version. After the first implementation was finished, a lot of major improvements have been made to the system. These include pipelining, dynamic reconfiguration capabilities, caches, integration to the GRLIB platform [11], several compiler back-ends, and a Linux port.

2.4.1 The VEX Instruction Set Architecture

The ρ-VEX processor is based around the VEX Instruction Set Architecture (ISA). This ISA is introduced as an example ISA in [12]. This simplified example ISA is in turn loosely modelled on the Lx platform by HP and STMicroelectronics [13]. HP created a C compiler and simulation system that is freely available at [14]. Unfortunately, this compiler is closed source.

The VEX ISA is well suited for parametrically configurable softcores because it is highly flexible. Many aspects of the architecture can be specified as parameters. This includes the number of issue lanes, the number and type of functional units, and the size of the register file. The compiler also fully supports these configuration and outputs code which is tailored to the specified configuration. By using the simulation system, the execution times of a program can be evaluated for different architectural configurations. This enables determining the optimal configuration of a core for a specific application.

2.4.2 Static Configuration

Complying with the rationale of the VEX system to enable the processor to be configured optimally for a specific application, the ρ-VEX also supports a large amount of architectural parameters. The design parameters currently supported by the ρ-VEX processor are given in Table 2.1.

Table 2.1: Static configuration parameters supported by the ρ-VEX processor

Resource	**Parameters**
General	Issue Width, number of hardware contexts,
Functional units	Number, type and location of functional units, supported operations
Register file	Register file size
Interconnect	presence of forwarding logic, memory bandwidth
Caches	presence of caches, cache size, cache line size

2.4.3 Dynamic Reconfiguration

The aspect that is most innovative about the ρ-VEX processor is its capability to be dynamically reconfigured. This feature is first introduced in [15]. Dynamic reconfiguration entails splitting and merging cores creating VLIW processors with narrower or wider issue widths during runtime.

In the default setup of the current version of the ρ-VEX, the core is configured as a single wide 8-issue VLIW processor. This core can be split up into a maximum of four 2-issue cores. These cores can execute processes completely independent of the other cores. The possible configurations are one 8-issue, two 4-issues, four 2-issue, or a combination of two 2-issues and one 4-issue. These different combinations allow the core to adapt to the application during runtime. When a problem which has high ILP must be solved quickly, the core can be configured as a wide vector processor, thereby drastically reducing the number of cycles required for program completion. If there is not enough ILP to fill all the issue slots and a core executes a large amount of No Operations (NOPs), the core

can be split into two or more smaller ones. This allows multiple processes to run in parallel, increasing the processors Thread Level Parallelism (TLP)

When the decision is made to split the core, the original process can continue on a narrower core while starting another process on the core that has become available. When no processes are available to run on the newly freed core, it can be shut down. The core is accompanied by a cache that can split and merge in a similar fashion. One of the following subsections will give a detailed description of the cache.

At some point in the future development of the ρ-VEX system, a software layer will be developed which will decide during runtime when the core will be merged or split. To facilitate this, a proper port of the Linux operating system must be developed first. The implementation of a MMU is indispensable to accomplish that. One major contribution that has been made already in [16] is generic binaries. These pieces of executable code are suitable to be run on any configuration of the ρ-VEX independent of the issue-width. Without this technique multiple versions of programs must be available for different configurations, which must be switched upon a reconfiguration.

2.4.4 Reconfigurable Cache

The ρ-VEX core is accompanied by a cache memory that has properties similar to the core regarding dynamic reconfiguration. In the standard configuration, the cache consists of four blocks which each have their own port for reading and writing. This is required for the most demanding scenario in which the processor is divided in four 2-issue blocks. When two or more smaller cores are merged into a larger one, their respective cache blocks merge in the same way. The coalesced core can access data and instructions located in any of these blocks. This effectively creates larger caches when less cores are active. This technique allows re-purposing of the otherwise unused cache blocks.

2.5 Memory Management

In this section, the advantages of incorporating a memory management system in a computer are discussed, as well as the techniques used to provide dynamic address translation functionality to the processor.

2.5.1 Virtual Address Spaces

In a system without memory management and address translation, the programs executed by the processor directly use physical memory addresses to access code and data. This means that programs require a contiguous memory segment and either runtime relocation or position independent code. Because every program can access the entire memory, running multiple programs concurrently is dangerous when one of the programs is flawed. When a program erroneously accesses memory outside its designated region, it can destroy others programs or their data. A program can even overwrite the operating system OS's code or data structures.

In a system which does use address translation, each process has its own virtual address space. This address space is exclusive to the process and is independent of

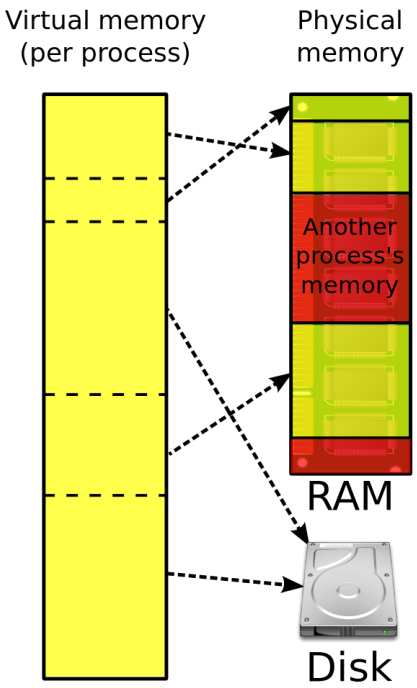


Figure 2.3: This diagram shows how a program divided into pages is mapped to physical memory frames. The image also illustrates that not all parts of the program need to be in memory at the same time. Some pages can be located on backing storage such as a hard drive.

the physical address range both in mapping and in size. The only restriction is the amount of memory addressable by the the bit width used for addresses. The mapping between virtual addresses and physical ones is generated by the OS while the translation of addresses for instruction fetching and data access is done dynamically by the memory management unit. Mapping between virtual and physical addresses is done for blocks of memory of fixed size called pages and frames respectively. This means that, the least significant part of the virtual and physical address, are the same.

The OS thus divides the address space of a program into regular sized block, called pages, and maps these to frames of the same size in the physical memory. The frames the pages are mapped to do not have to be contiguous. This enables more efficient use of the memory since programs can be loaded into the memory even when there is no contiguous area of free memory available. Another advantage of paging is that not all pages of a program need to reside in memory for while a program is running. The OS simply maps pages to memory when they are needed, a technique called demand paging. This can reduce the memory footprint of a process dramatically and allow more processes to be loaded concurrently. Programs can even be started without any part loaded into memory, requesting the pages it needs to be loaded whenever it accesses them. Another advantage of demand paging is that programs which are larger than the physical memory can be run on the system. This is accomplished by swapping its pages

in and out of memory to the backing storage which is usually comprised of a magnetic disk memory. In Figure 2.4 a diagram shows how a programs is mapped from its private virtual address space to the physical main memory.

Because the OS creates the mapping of addresses for each process, it can ensure that programs can not access memory allocated to other programs. In this way, paging also enables memory protection. Moreover, it also facilitates shared memory simply by mapping pages of different processes to the same frame. This is very useful for shared data segments or shared libraries, which have to be present in memory only once. Contemporary OS's use this feature for a technique called copy-on-write. This entails sharing memory resources between threads as long as they do not modify the data. Duplicating the data is postponed until one of the thread performs a write.

Summarizing, address translating through paging has the following advantages:

- Programs larger than the physical memory can be executed.
- Significant reduction of memory usage by demand paging, allowing more concurrent processes.
- Memory protection is implicit due to process specific address mappings.
- The ability to share sections of memory between processes enabling shared data segments, shared libraries, and copy-on-write protocols.

2.5.2 Page Tables

The mappings between virtual pages and physical frames created by the OS and referenced by the MMU are stored in a data structure called a Page Table (PT). A PT is maintained for each process running on the CPU and these are stored in main memory. Usually, a special register holds the address of the page table and must be updated together with the rest of the context when a process switch occurs. As the name indicates, PTs are tables which hold physical tags which are indexed by virtual tags creating a one-to-one or sometimes many-to-one mapping between them. When PTs are implemented as one large table they can consume a lot of memory. In a system using a 32 bit addresses and 4kB pages, the PT needs to hold a mapping for $2^{32} - 2^{12} = 2^{20}$ different pages. Assuming a PT entry is 4 bytes this amounts to 4 MB. Because the OS needs one for every process this can quickly become a large burden, especially for smaller systems or systems running a lot of processes. Most systems solve this by implementing a multilevel PT [17]. Figure 2.4 depicts a multilevel page table with two levels. An address lookup in such a table is performed as follows. First, the most significant part of the tag is used to index the outer table often called Page Directory. This lookup returns the address of a the second level called Page Table which holds physical tags. This table is then indexed with the least significant part of the tag, resulting in the address of the requested frame. Finally, the offset part of the address is added to the physical tag, resulting in the memory location of the requested instruction or word of data. When the entire two level PT is present in memory, there is no advantage in memory usage over a single level PT. However, a typical process uses only use a part of the total memory range. This combined with the technique of Demand Paging as explained in Section 2.5

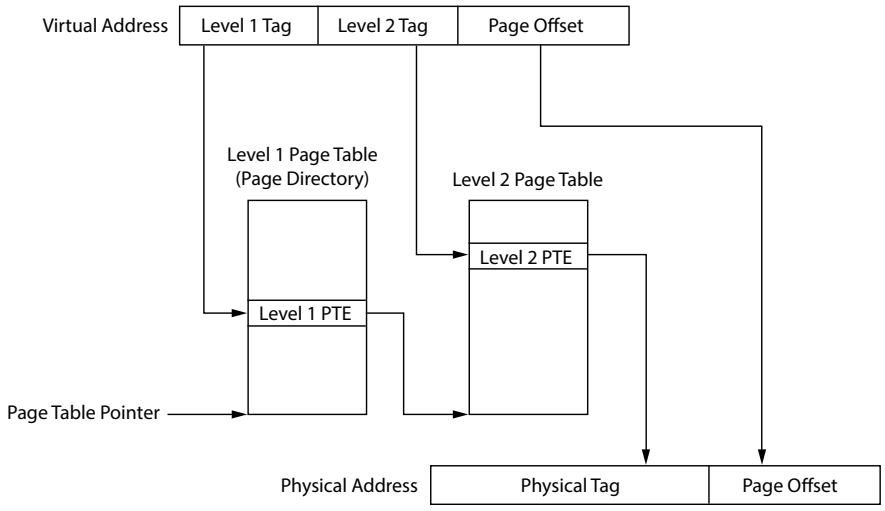


Figure 2.4: Multilevel page table

makes it possible for the system to only allocate the tables which are actually used. The OS can thus start by only allocating space for the Page Directory Table and extend the PT with Page Tables when they are referenced. The two level table as described is how 32 bit systems often deal with the page table. For 64 bit systems however, two levels do not suffice and PTs with more levels are used.

2.5.3 Hardware Support

As already described in Section 2.5, the OS generates mappings between virtual pages and physical frames. It does this often on demand and distributes the program through memory based on which frames are free. When all of the memory is occupied, it can swap out another page and invalidate its mapping. The OS stores its valid mappings in a data structure called a PT which itself also resides in main memory. When an address is issued by the processor to fetch an instruction or to access data, the memory management unit accesses the PT in memory to find the pages location in memory. This process is often performed dynamically by a component called a Table Walker (TW). The base address of the frame is then added to the offset within the page/frame to determine the location of the instruction or data. Because of this process, memory accesses requires one or multiple memory accesses to the PT before the requested instruction or word of data can be retrieved. Having to perform this procedure for every memory access would increase the memory latency of the system unacceptably. To mitigate this overhead, a small cache memory, called a Translation Look-aside Buffer (TLB), is used to store mappings from the PT. Most systems implement their TLB(s) as a fully associative cache, sometimes referred to as a Content Addressable Memory (CAM). Unlike Random Access Memory (RAM) in which an address is supplied and the memory returns the data entry at that location, in a CAM a data word is supplied and the entire memory is searched checking for a match. The CAM then returns the address where the data was found and/or a piece of associated data. This type of memory is expensive in terms

of area and power consumption, so the size of TLBs is often kept small. In the general case in which the CPU issues an address which falls in a page which mapping is cached, the translation can be performed in a single cycle. When the TLB does not hold the translation, it needs to update its content stalling the processor for some time.

Some systems exist in which multiple levels of TLBs exist. These configurations usually have very small single cycle TLBs and larger ones that are referenced in case the higher level miss. These larger ones then do not need to be single cycle and therefore not fully associative.

2.5.4 Operating System Support

While memory management and address translation is transparent to user programs running on the processor, this is not the case for the operating system. Since one of the main tasks of the operating system is to allocate resources to programs running on the processor, it also divides the main memory between them. While the translation of memory addresses is done automatically in hardware, the allocation and deallocation of memory is done on a higher level in software by the operating system. This means that in virtually all modern systems, the MMU reads address mapping from the PT autonomously but relies on the processor to update it when a mapping is not present. When this occurs, the MMU issues an interrupt signal, notifying the processor it is unable to continue. This triggers an Interrupt Service Routine (ISR) containing code to load the requested page into main memory and to update the corresponding entry in the PT. When the control flow is given back to the original program the memory access is reissued and the TW should now find the mapping in the updated PT.

Besides the address mapping, the PT usually also contains bit-flags to keep track of access right and other page properties. Among these flags are bits used to mark a page as *dirty* and *accessed*. These bits denote whether a page is modified or accessed recently. This information can help the OS decide which pages are good candidates for swapping and whether a page must be written back to memory or can simply be invalidated. These bits are set by the TW hardware.

This division of work between the OS and the MMU holds for the majority of systems. However there are also systems which approach the problem differently. Some (lightweight) systems rely on the OS to update the TLB and do not have a TW. One example of such a system is the ST200 microprocessor closely related to the ρ-VEX [18]. This approach saves hardware resources by moving a task to software but increases latency of a TLB miss. When such a system encounters a TLB miss, an ISR must be invoked to perform the table walk in software.

2.5.5 Caches and Virtual Memory

The operating speed of processors and their memory have not been developing at the same rate. Historically processor speed has increased at a faster rate than memory speed, therefore the gap between them has kept increasing since the early days of computers [19]. This development has lead to a situation where memory accesses have become the bottleneck for processors in a lot applications. To mitigate this problem, a smaller but faster memory is placed in between the processor and the main memory, called a

cache, which holds a fraction of the main memory. Due to the principles of spacial and temporal locality in most programs, sections of the memory can often be loaded once and referenced multiple times, leading to a lower average memory access latency. All but the most basic contemporary processors incorporate a cache memory. For a complete discussion of the topic of caches caches refer to [20].

At some point in the memory hierarchy, between the address issued by the processor and the physical main memory, the address must be translated. This leaves the cache somewhere in the middle and thus raises the question whether the cache should use virtual or physical addresses. It is also possible to create hybrid caches in which the part of the address used to index the cache and the part used to check for a hit, referred to as the *tag*, differ in the sense that one is virtual and the other is physical.

Purely virtual caches have the main advantage that address translation is only required when the cache misses and new data has to be be brought in from the main memory. In the general case of a cache hit, the address translation hardware is not needed. This relaxes the latency requirements of the TLB. The major drawback of this approach is the possibility of synonyms inside the cache. A synonym occurs when different virtual pages map to the same physical frame. When a write is performed using one of the virtual translations, the other becomes outdated. To enforce coherence, synonyms must either be avoided or a mechanism to update them dynamically must be implemented [21].

Physical caches do not suffer from this problem allowing easy coherency maintenance. Their main drawback is that the address translation step needs to be performed before every cache access. This increases the latency of the fetch and write-back stages of a CPU pipeline. To avoid lowering the operating frequency extra pipeline stages must be inserted. This increases the pipeline latency which in turn increases the branch penalty. A detailed discussion of the different types of caches these design choices yield can be found in Section 3.4.

2.5.6 Memory Management in Multiprocessor Systems

Supporting virtual addressing in a multiprocessor system is more involved then in a uniprocessor system. These complications stem from potential coherency problems which arise in a system with multiple caches and TLBs [1]. Since a single ρ-VEX core can be configured as multiple independent processors which have private caches and TLBs, a system with a single ρ-VEX core is actually a multiprocessor system.

Coherence maintenance schemes always operate on system wide physical addresses. Whether a directory based scheme is used or, more often, a snooping based protocol. Since virtual caches use local virtual addresses to store its contents, some reverse translation mechanism is required to translate the physical addresses to virtual ones. Such a mechanism is complicated and thus costly to implement.

Including multiple TLBs in a computing system introduces the same problem as when multiple caches are present. Each processor in a multiprocessing system has its own set of TLBs. When one of the processors allocates or deallocates a page and modifies a process' PT, all TLB entries holding that translations must be updated or invalidated. Most system enforce this using a mechanism referred to as *TLB shootdowns* [22]. When a

processor updates a PT, it uses interprocessor interrupts to stall the other cores present in the system. When all other cores are stalled, the initiating core modifies the PT. After it releases the other cores, they can invalidate the affected entries in their TLBs. This technique is described in detail in [23].

The memory hierarchy in a multiprocessing system is often divided in multiple layers of caches. Often each processor have at least one level of private cache. Higher cache levels can sometimes be shared among the different cores. It is not uncommon to have a layer of private virtual cache close to the processors, and a shared layer of physical cache memory. The current implementation of the ρ-VEX uses a single layer of cache memory.

2.6 Minimal functional requirements of an MMU

This section list the absolute minimal functionality an MMU should offer to support an OS implementing a virtual addressing scheme. The list is composed based on several books and articles [17] [24] [25]. In Chapter 5 the final implementation will evaluated based on these functionalities.

Minimal functional requirements

1. Autonomous address translation of mappings stored in the TLB.[1]
2. Automatic checking of access rights (read/write and protection level) on every page access.
3. A way to insert entries in the TLB; either a software interface or an TW unit.
4. In case of a hardware managed TLB, a mechanism to stall the core upon a table walk and a clearly defined PT layout.
5. A mechanism to issue a trap when a page fault or access violation occus.
6. A software interface to flush TLB entries.
7. A mechanism to bypass the MMU for system initialization and kernel code that is executed in physical address space.

2.7 Conclusion

In this chapter, first some background information about the implementation platform for the project was discussed. The first section of the chapter explained how FPGAs work and what their benefits are. Subsequently, it was explained how processors, referred to

[1] Note that the addition of a TLB is not strictly necessary since the MMU could access the PT instead on every translation request. However since this requires multiple memory accesses to serve a single access requested by the CPU this effectively multiplies the systems memory latency by the number of PT levels. Since this cost is unacceptable the addition of a TLB is regarded as a strict necessity.

as softcore, can be implemented in this technology. In the following section, the general concept of VLIW processors and their advantages were explained.

This established the basis that allowed for the introduction of the ρ-VEX processor. The ρ-VEX is a VLIW softcore processor implemented in the reconfigurable fabric of an FPGA. Being a VLIW processor, the ρ-VEX issues instruction in bundles of two or more. This increases the ILP and thereby the performance of application with sufficient levels of parallelism.

The ρ-VEX can be configured in a number of ways at design-time. This enables tailoring the ρ-VEX system to an specific application, optimizing performance and minimizing area. Besides these static configuration parameters, the ρ-VEX is also dynamically reconfigurable during run-time. This allows the ρ-VEX processor to switch between a single wide VLIW processor or multiple smaller cores. This feature enables running a single process with with high ILP or multiple processes with less parallelism, thereby increasing TLP. Being able to switch between these configurations has implications for the memory system. The ρ-VEX system already features a cache that supports the core in each possible configuration. One of the challenges of this thesis project is to design and implement an MMU with similar properties.

The second part of this chapter explained the key concepts of virtual memory using paging. In a system that supports paging, each process has its own virtual address space. The mapping of this address space to physical memory is controlled by the OS and is transparent to user programs. The translation of addresses is done dynamically by dedicated hardware called the MMU. The MMU contains multiple caches called TLBs which are used to cache the virtual to physical mappings. The address mappings created by the OS are stored in main memory in a process specific data structure called a PT. When an address is referenced that is not contained in the TLB, another hardware component called the Table Walker (TW) searches the PT for the missing translation. When the translation is not present in the PT, the TW generates an interrupt to call for OS intervention. The OS then invokes a routine that bring the missing page into memory and updates the PT.

In the last section of this chapter, a listing was made of the minimum set of functions that virtual address hardware should posses to support an OS. In Chapter 5, this list is revisited and used to evaluate the final implementation.

3 Architecture

In Chapter 2, the background for this project was explained. The ρ-VEX processor was introduced an its unique features were explained. Then an overview of the subject of memory management was given. The required hardware support was discussed, the division of work between hardware and software, and the data structure used to hold each process set of address mappings, the Page Table (PT).

In this chapter, the architectural design for the ρ-VEX Memory Management Unit (MMU) will be described. All the important high-level design decisions will be explained and substantiated. In Section 3.1, an overview of the MMUs components and the most important signals connecting them is given. Subsequently in Section 3.2, a quick overview of caches and related terminology is treated, which is essential knowledge to understand the following sections. For the same reason, in Section 3.3, the terminology used in relation to memory management is clarified. In Section 3.4, the most important high-level decision problem is outlined. This will determine where in the memory hierarchy the Translation Look-aside Buffers (TLBs) are placed. The solution space is explored, and in Section 3.5, one architecture is selected. The way the TLB will be managed will be explained in Section 3.6 Then in Section 3.8, the unique requirements for the ρ-VEX MMU will be explained. This is essentially what distinguish the MMU presented in this thesis from general MMUs. Finally, in Section 3.7 a number of other unique optimizations are proposed, which are possible in the ρ-VEX system.

3.1 Address Translation Hardware

In Figure 3.1 a very simple diagram is shown depicting the bare minimum components required in a MMU, that supports hardware TLB management. Also depicted are the most important internal signals and the interface to its tightly coupled neighbours, the core and the memory.

The core supplies the MMU with virtual page numbers of every memory location it wants to access. This part of the address is referred to as the *tag*. Physical tags are translated to virtual tags dynamically by the MMU. Recall from Section 2.5 that the rest of the address, called the *index*, is the same for both physical and virtual addresses. The TLB checks its internal cache if it contains the requested mapping and if so, passes it on to the memory. The memory can then perform the memory operation based on the translated address.

If the TLB does not have the requested mapping, the TLB stalls the core and signals a *miss* to the Table Walker (TW) which then searchers the PT to look if the requested page has a mapping and is actually held in memory. If the TW is able to find a valid mapping, it updates the TLB signalling it the processor can continue.

In case the TW does not find a valid mapping, a page fault trap is generated and

control is handed to the Operating System (OS) which then can load the page into memory and update the PT. When the OS returns the control, the Program Counter (PC) is reset to the instruction containing the memory access that caused the page fault. The whole procedure above is repeated but now the TW will find the new mapping in the PT so it can update the TLB and the program can continue.

When the Central Processing Unit (CPU) switches to kernel mode to run the Interrupt Service Routine (ISR) which is responsible for updating the PT, it either bypasses the MMU entirely, executing in physical mode, using physical addresses. Or alternatively, it use a separate PT which is used exclusively for kernel code, containing pages that are never swapped out. Both these techniques ensure that no page faults are encountered when servicing an earlier fault. Which of these techniques the OS uses differs, but most contemporary OS's use the second technique and also execute kernel code using virtual addresses.

3.2 Caches

In Section 3.1, the concept of caches was introduced. While the assumption is made that the reader has basic knowledge of computer architecture, the operation of a cache will be briefly explained because it is required to understand the discussion in the following sections.

Caches are used to hold a subset of the main memory fast accessible storage. Because the size of caches limited, some mapping is required from main memory entries to cache entries. In the simplest form caches are directly mapped. In this system the cache address is the address in main memory modulo the size of the cache. This means that main memory addresses contend for the same location in the cache if the stride between them is the size of the cache. Address are divided in an *index* and a *tag*, which are the lower and higher slices of the address respectively. When the core performs a memory access, the index is used to select which cache entry to access. Each cache entry holds a tag which is compared to the tag of the issued address. When these two match, the indexed cache location holds the requested data. If they do not match, another memory location is cached in that entry. If this happens the data must be retrieved from main

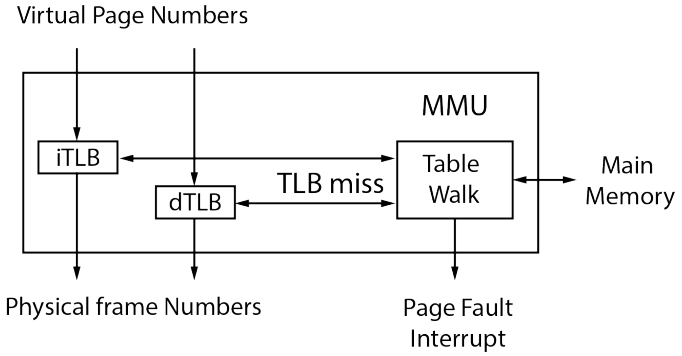


Figure 3.1: Simplified diagram of an MMU including the bare minimum components and the most important signals.

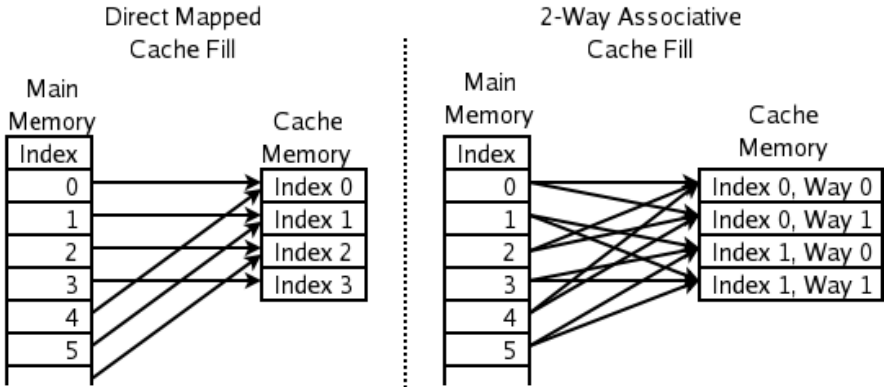


Figure 3.2: This diagram shows how memory locations map to cache entries for a direct mapped cache and a 2-way set associative cache.

memory and the cache updated. Caches can also be fully- or set-associative which means that a memory location can be mapped to different cache entries. This allows more freedom but requires multiple comparator circuits in the cache to compare multiple tags in parallel. Figure 3.2 shows how memory locations are mapped to cache entries for a direct mapped and 2-way set associative cache. For a more extensive discussion about different types of caches refer to [20].

3.3 Memory Management Terminology

The convolution of cache systems and virtual memory systems is complicated by the fact that both use the term *tag* when referring to the most significant part of a memory address. While they refer to the same general concept, it is not always necessarily the case that they refer to the exact same slice of the address. In fact there are many systems in which they differ. For instance in the ST231 processor which is based on a similar Instruction Set Architecture (ISA) as the ρ-VEX processor, the page size is fixed at 8 kB while the size of the instruction cache is 32 kB (directly mapped) [18]. This leads to a 19 bit page tag and a 17 bit cache tag. In the ρ-VEX system both the page size and the cache sizes are parametric so both tags can refer to different slices of the address in different configurations. While it is possible to refer to the page tag as the virtual or physical page *number*, the term tag is used to comply with literature. Which tag is referred to is either specified or should be clear from the context.

This indistinction does not exist for the least significant part of the address. When referring to this part of the address in the context of caches, the term *index* is used because it is used as an index to the cache. In a page context, it is referred to as the (page) *offset*.

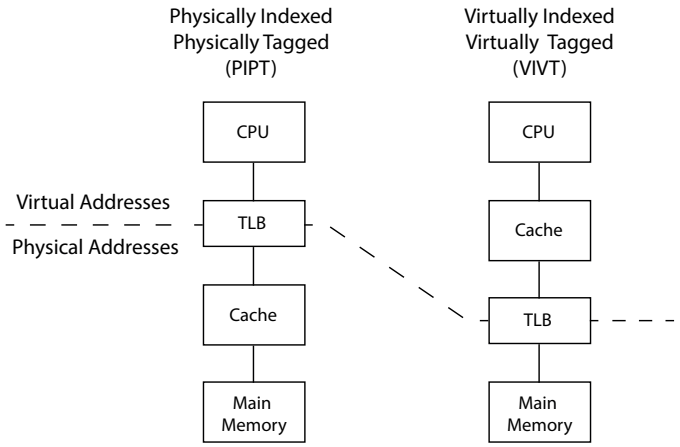


Figure 3.3: The distinction between a virtual and physical cache.

3.4 Virtual Memory Hierarchies

When adding virtual memory support to a processor, one of the first high-level decisions to address is where in the memory hierarchy the address translation will take place. Almost all contemporary processors include cache memories and these can be accessed with either virtual or physical addresses. Figure 3.3 illustrates the difference between the two hierarchies. The current implementation of the ρ-VEX has a single level of cache memory with separate caches for instructions and data. Another property of the ρ-VEX which is important to take into account is that the processor can be split into multiple cores with separate caches, which creates the need for cache coherence. The decision to place the caches in virtual address space increases the complexity of cache coherence protocols significantly [1]. This is because cache coherence is based on system wide physical addresses while the data in the caches is indexed based on local physical addresses. This discrepancy raises the need for a reverse address translation mechanism which both increases the complexity and area footprint of bus based coherency mechanisms.

Caches are indexed with the bottom part of an address and subsequently the top part of the address, the tag, is compared with the tag stored in the cache. This is a check to see whether the instruction or data corresponding to the requested address is present in the cache. These two parts of the address can be both taken from virtual or physical addresses, but it is also possible to index a cache with a virtual address and compare physical tags and vice versa. This leads to four possible combinations of indices and tags. In the rest of this thesis abbreviations are used to refer to these types of caches. For instance a cache which is physically indexed and physically tagged is called a Physically Indexed Physically Tagged (PIPT) cache. In the following sections, the advantages and disadvantages of the different types of caches are explained and the type best suited for the ρ-VEX system is selected.

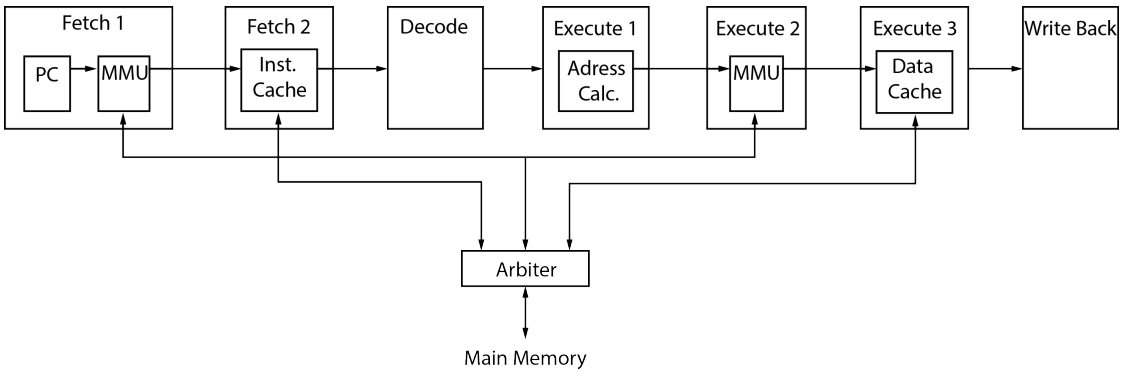


Figure 3.4: Extended pipeline required for a PIPT cache hierarchy. This pipeline has seven stages compared to five for the original pipeline.

3.4.1 PIPT Caches

In a PIPT cache hierarchy address translation is performed before the cache lookup. This places the MMU between the core and the cache. The advantage of this configuration is that the cache does not have to be modified since the virtual address space used inside the core is not extended to the cache. Another advantage is that coherency mechanisms are easy to implement because the cache holds system wide physical addresses which are the same for all cores. The main drawback of the PIPT cache is that the address translation hardware is in the critical path for memory accesses. This means that the latency for fetching instructions and data becomes larger. This can lead to a lower operating frequency or an extra pipeline stage. The diagram in Figure 3.4 shows how the ρ-VEX pipeline would look if a PIPT cache would be implemented. Both the fetch and execute stage would be extended to allow the address translation to take place in an extra cycle. While this probably would not decrease the operating frequency. However, the pipelines latency would increase by two clock cycles, leading to higher penalties for pipeline flushes on branching.

3.4.2 VIVT Caches

In a Virtually Indexed Virtually Tagged (VIVT) cache, the cache is more tightly coupled to the processor than to the memory. This means that the cache is indexed by and contains only virtual addresses. Only when the cache misses, these are translated to physical addresses to update the cache from the main memory. This means that the whole address translation process and the associated latency is avoided in the general case where the cache hits. Because of this, the VIVT cache is potentially the best configuration performance wise. This aspect also relaxes the performance constrains of the TLB. Since TLB lookups do not occur every cycle but only on cache misses, it is no longer a necessity that the address translation is completed in a single clock cycle. Additionally, it also allows multiple cores to share a single TLB without a significant performance penalty. However, this type of cache introduces quite a few difficulties, especially in the case of a multiprocessor system.

One of the issues of VIVT caches are the occurrence of homonyms. A homonym occurs when address mappings of different processes map the same virtual address to different physical addresses. A mechanism to distinguish between these needs to be implemented to avoid flushing the entire cache when a process switch occurs. One solution to this problem is to include an Address Space Identifier (ASID) with each TLB entry. Each process is also assigned an ASID and the TLB only hits when the active process' ASID matches with that of the entry.

A more difficult problem of VIVT caches is the synonym problem. This occurs when different processes map different virtual addresses to the same physical address. This can happen when two processes use the same library or share a section of memory for data sharing. When one of the cache entries is updated, the other one becomes outdated. Normal coherency mechanisms do not detect this because it is not directly apparent that the two entries refer to the same physical memory location. Most systems deal with this problem by not allowing synonyms to exist in the cache at the same time. This entails performing a reverse translation each time a cache entry is updated. When the reverse translation hits, the synonym is flushed from the cache.

A related issue concerning VIVT caches is maintaining coherency between caches of different CPUs in a multi-core system. Coherency schemes enforce coherency on system wide physical addresses. Therefore the reverse translation hardware is also required to implement bus snooping cache coherency mechanisms.

Even more issues exist. In a VIVT system, not every memory access goes through the TLB. This means that access right bits need to be stored in the cache together with every entry. Other cache configurations simply store these in the TLB together with the address mapping. While this is a minor issue compared to the ones previously described, it does lead to a larger cache footprint.

Another issue related to storing access permissions inside the cache is that cache data can become invalid when a pages access rights are changed. When this happens the cache must be flushed [21] which can result in a substantial performance penalty.

Because the ρ-VEX can be configured as multiple individual cores, it must be considered a multicore system. This means that, if the choice is made for a VIVT cache, the synonym problem within a single cache block and between cache block needs to be solved. In the following sections two different methods that are able to cope with these problems are presented.

3.4.3 VIVT Cache with Dual Directory

As previously explained, the main problem in a VIVT cache is that the use of virtual addresses complicates maintaining cache coherence both within and between cache blocks. The external coherency problem stems from the fact that the cache is accessed with virtual addresses by the CPU while the bus snooping mechanism uses physical addresses. Indexing the cache with a virtual address leads to another entry than when it is indexed with a physical address. Figure 3.5 illustrates this problem. The part of the address that is used to index the cache which is different a virtual and physical address, i.e., falls

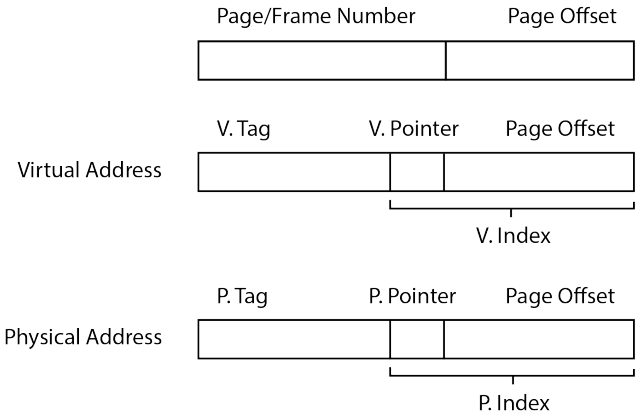


Figure 3.5: The address layout of physical and virtual addresses when dealing with a virtual dual directory cache.

outside the page offset, are called *superset* bits [1].

The solution explained in this section solves both coherency problems by expanding the cache with a dual directory and is taken from [26]. The term dual directory refers to a second tag memory for accesses using physical addresses. This secondary tag memory is used by the bus snooping system and holds physical tags for coherency operations. Additionally, this memory also holds pointers to each entry's location in the caches data memory. In Figure 3.6 the components in such a system are illustrated. Whenever a coherency operation must be performed, the bus snooping system uses the dual directory to check if the entry is present in the cache. When the dual directory hits, the data- and virtual tag memories are updated or invalidated using the virtual pointer found in the dual directory. Note that only the superset bits need to be maintained to store the virtual pointer in the dual directory.

Besides supporting bus based coherence protocols, the dual directory can also be used to prevent inserting synonyms in the cache. Whenever the cache misses, the virtual address is translated to a physical one. This physical address is then used to index the dual directory. If the dual directory hits, this means that the data is already in the cache under a different synonym, which is called a *short miss*. In this case, the data is already present in the cache and can simply be be moved. See Figure 3.7 for a schematic depiction of this situation.

Besides the added cost of implementing a dual directory, this mechanism has a serious drawback which affects effective cache utilization. This problem is referred to as *paired eviction*. Every entry in the cache must have a valid backpointer in the dual directory. Therefore it can happen that when replacing a cache entry, its corresponding entry in the dual directory clashes with another entry. When this happens the entry in the dual directory and its corresponding cache entry need to be removed from the cache.

[1] When the part of the address used to index the cache falls completely within the page offset, i.e., the cache size (per way of associativity) is not larger then the page size, the problem is circumvented and the VIVT cache can be turned into an Virtually Indexed Physically Tagged (VIPT) cache which will be discussed later.

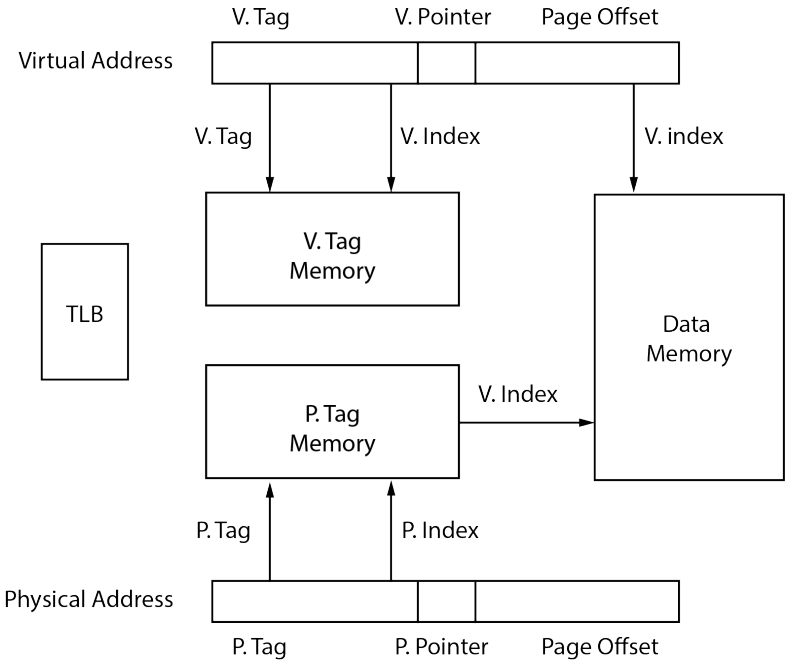


Figure 3.6: Schematic diagram of a VIVT dual directory cache. This image shows how the cache can be accessed by both virtual and physical addresses.

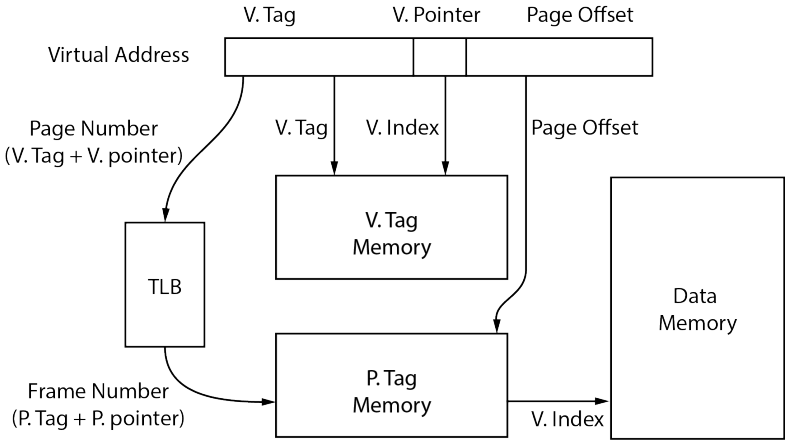


Figure 3.7: Schematic diagram of a miss in a VIVT dual directory cache. When a miss is encountered, first the translated address is used to index the dual directory to check if the data was already present under a different synonym.

In this situation, a single cache miss requires replacing two cache entries. See Figure 3.8 for an example and a graphical representation of the paired eviction problem. The effective cache occupancy due to paired eviction is at most $\frac{N}{2*N-1}$ where N is the number of superset bits. This relation is proved in [27]. In a system with 4 KiB pages and a 16 KiB caches addresses have two superset bits. This is a realistic scenario for ρ-VEX configuration and leads leads to an effective cache occupancy of 67%. When the cache

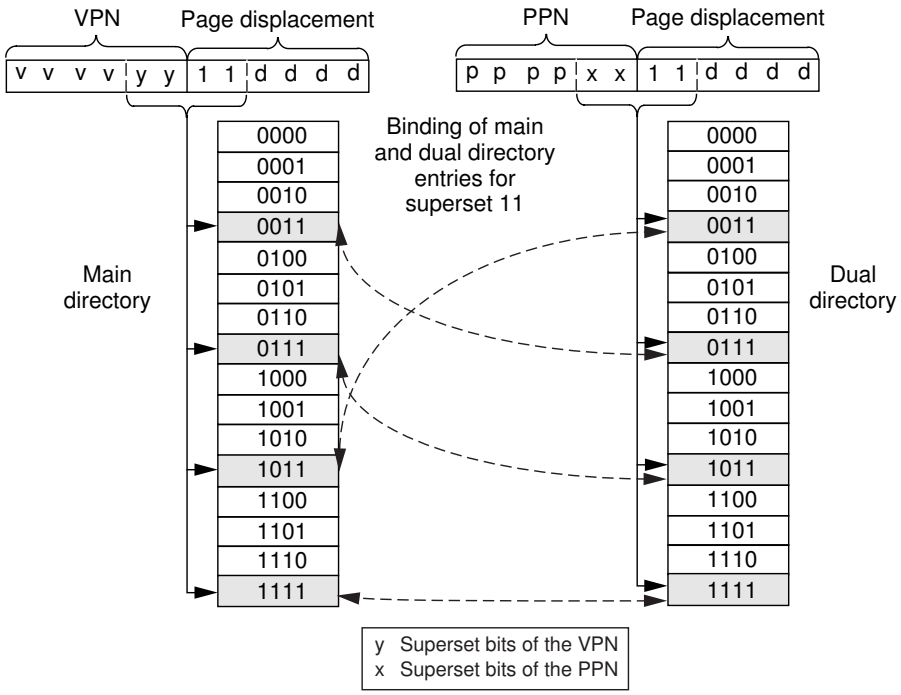


Figure 3.8: This diagram depicts the paired eviction problem. VPN and PPN refer to virtual- and physical page numbers, which are referred to as tags in this thesis. This diagram is taken from [1].

size rises in relation to the page size, the effective cache occupation drops and approaches 50%

3.4.4 VIVT Multilevel Cache

Literature suggest that a second level (L2) cache can easily solve all problems associated with virtual caches [1] [28]. In a single level virtual cache system the cost of the dual directory significantly increases the area of the cache due to having to maintain two tag memories. When a second level physical cache is present, the dual directory is almost entirely in place already. The added cost is only in keeping backpointers to the location of the L2 entries in the L1 cache. This means that the only addition to both tag memories is the superset bits of the translated address. In Figure 3.9 a schematic is shown of the two cache levels and their main components.

Several restrictions must be enforced to make this scheme work. Implementing a write through policy for the L1 cache ensures that the data in both levels is coherent with each other. This allows maintaining coherency only on the physical L2 cache as long as invalidations are propagated to the L1 one cache using the backpointers stored in the L2 cache. The synonym problem is solved in the same way as in the dual directory scheme with a single level of cache. When the L1 cache misses, the address is translated and the L2 cache is checked if the requested data is present under a synonym. When this is the case, updating the pointers in the L1 and L2 cache is sufficient. When accessing

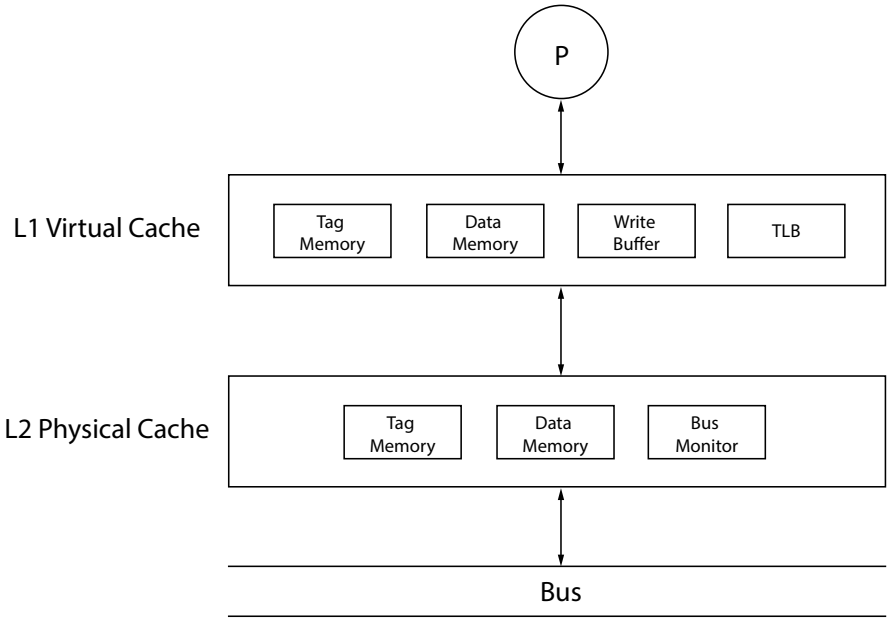


Figure 3.9: Cache hierarchy with a virtual first level cache and a physical second level cache. The L2 cache hides most of the cost of the dual directory.

the L1 cache, the address is always translated in parallel in case the L1 cache misses.

The paired eviction problem also exists in this configuration. However, since the L2 cache is usually larger than the L1 cache, not all L2 entries are also present in L1. This means that often when replacing a L2 entry, there is no associated L1 entry. When the L2 cache is set associative, the replacement policy can also prefer entries that are not present in the first level.

3.4.5 VIPT Caches

VIPT caches are a intermediate solution between pure physical and virtual caches. In this configuration the cache is indexed directly with the virtual index. This means that the cache lookup can start directly when the memory address becomes available in the fetch or execute stage for instruction and data accesses respectively. Therefore an extended pipeline is not required as with a PIPT cache. The tag memory in the cache does contains physical tags. The translation of the virtual tag issued by the core can occur in the same cycle as the cache lookup. When both the cache tag and the page tag are available, a comparison is made leading to either a TLB hit or miss. This process is shown graphically in Figure 3.10.

This type of cache allows bus access to maintain cache coherence without reverse translation if a restriction is placed on the cache size. When the size of the cache is at most the page size^2, it can be indexed by both virtual and physical addresses. This is possible because with this restriction, only the page offset is used as index which is

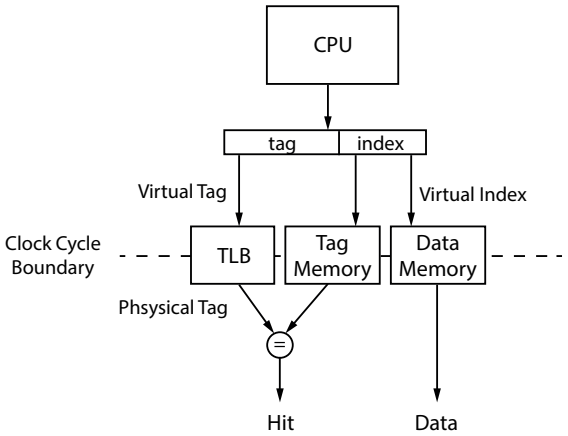


Figure 3.10: Schematic diagram of how the CPU accesses a VIPT cache.

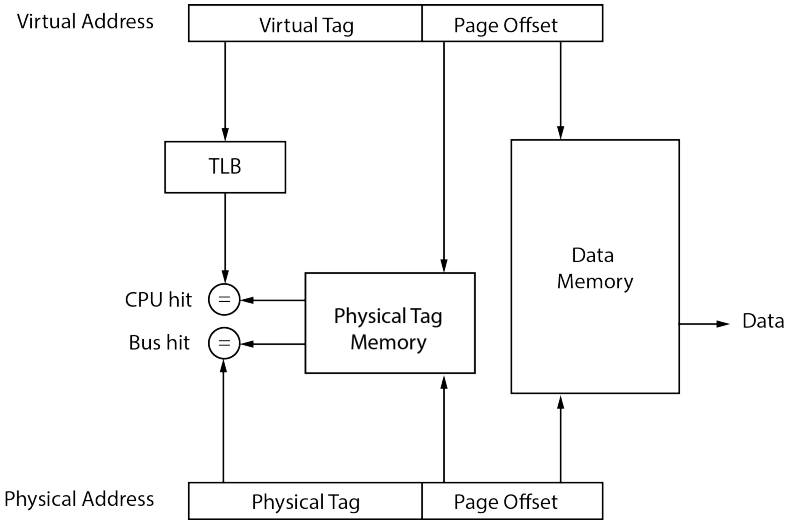


Figure 3.11: In a VIPT cache CPU access and bus access can be done as long as the cache size (per way of associativity) is not greater than the page size.

identical for virtual and physical addresses. Figure 3.11 shows how such a system works.

Besides allowing easy cache coherence, the VIPT cache also avoids the synonym problem. Because cache indexing is done based on universal tags, synonyms simply map to the same location. When the cache is set associative, the synonyms can also not coexist within a set because the synonyms have the same physical tag.

There are several methods to increase the cache size beyond the size of a page. The first is already mentioned and is to increase the set associativity of the cache. Another way, that requires OS support, is a technique called *page coloring*. The requirement for this type of cache is that it is indexed by the part of the address which is the same for physical and virtual addresses. This is normally limited to the page size. When the

[2]One way to still increase the size of the cache beyond the page size in VIPT system is to increase its associativity. This is possible since increasing the associativity does not lead to more index bits.

OS maps virtual pages to physical frames in such a way that the least significant bits of the tag are always the same, the part that is allowed as cache index in increased. The drawback of page coloring is that it limits the OS's freedom to allocate memory for pages, which can affect effective memory utilization. Page coloring is used in many ARM processors. A detailed explanation can be found in [29].

3.5 Memory Hierarchy selection

In the previous section, four different cache hierarchies are presented. Excluding very exotic solutions, these options should cover the design space for the ρ-VEX MMU. Table 3.1 summarizes the most important strengths and weaknesses of the proposed solutions.

Apart from having the highest complexity of all solutions, the multilevel configuration has the least significant drawbacks. However, it does require two levels of cache to be implemented. The current version of the ρ-VEX only has a single level and an additional cache level would drastically increase the area of the design. Because the ρ-VEX is currently only available as a softcore design, adding a second level cache would not allow the system to be implemented on smaller Field-Programmable Gate Arrays (FPGAs). Therefore this type of cache hierarchy is currently not a viable option.

The other three solutions all have one drawback that has the largest negative impact. In case of the PIPT cache, this is the increase in pipeline length by two stages. For the VIVT cache with a dual directory the increase in area utilization by the cache is the largest drawback. The VIPT cache suffers from the limitation in cache size.

Taking this all in consideration, the VIPT cache is selected as best suited for the ρ-VEX. As stated before the cache already occupies a large portion of the total area. Choosing the VIVT solution increases this even further. Furthermore, the complexity that VIVT caches introduce in multiprocessor systems are not well suited for the ρ-VEX. The ρ-VEX is a embedded processor that also needs to be suited for lightweight applications.

When discarding VIVT solutions entirely, the question remains whether to implement a PIPT or VIPT cache. Both their weaknesses affect to processor throughput. The increased pipeline required for the PIPT cache increases the branch penalty. As the ρ-VEX currently does not include a dynamic branch prediction mechanism, this cost can be significant. The VIPT solution potentially decreases throughput by increased memory stalls due to cache misses. To quantify this effect, the ρ-VEX simulation system is used to test several Powerstone benchmarks. These benchmarks are chosen because they have been used extensively in previous publications relating to the ρ-VEX. The results are shown in the graph depicted in Figure 3.13. For these programs only the smallest cache sizes affect the execution times of some benchmarks.

Alleviating the cache size limitation

To alleviate the cache size limitation of the VIPT cache as much as possible, several steps can be taken. Primarily the page size will be made parametric so that larger page sizes can be selected for application which require larger cache sizes. Almost all computer systems both historically and contemporary use a 4 kB page size. Some sources suggest

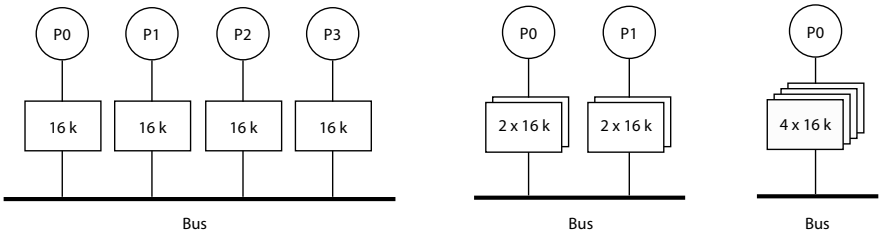


Figure 3.12: This diagram illustrates how the cache blocks can be combined as sets forming larger caches when pipelanes are combined.

that a larger page size has advantages in modern systems which have much more memory available [30]. The additional advantages include less TLB misses since each TLB entry spans a larger portion of the memory. The adverse effect of increasing the page size is internal memory fragmentation in pages of which only a small part is used. In case of a ρ-VEX core running on an ML605 development board with a Double Data Rate (DDR) Random Access Memory (RAM) module, the amount of memory available is extensive.

Another way to increase the size of the cache in a VIPT system is to increase the associativity of the cache. In Section 2.4.4, the ρ-VEX reconfigurable cache was briefly explained. The cache consists of four blocks that can operate independently or that can be merged to form larger caches. It is possible to coalesce these blocks as sets. Although it is implemented differently in the current version of the ρ-VEX system, This would combine in an efficient way with the choice for a VIPT cache. It would allow the size of a single cache block to be as large as the page size while larger cores with four or eight issue lanes would have caches double or four times the page size to their disposal. There is work in progress in the Computer Engineering (CE) group relating to reconfigurable set associative cache blocks for the ρ-VEX. This work has not been published yet so it is not possible to refer to it at this time.

A third way which is already mentioned in Section 3.4.5 is page coloring. This requires OS support and can negatively affect memory utilization. However, since the ρ-VEX system running on an ML605 board does not have a shortage of memory, this technique could be used if cache size becomes a bottleneck.

Finally, it is important to note that the limitation of the cache size is only an issue for the data cache. In the ρ-VEX system is is not possible to write to the instruction cache. Therefore, cache coherence is never an issue for the instruction cache. This relieves the instruction cache of all size limitations.

3.6 TLB Management

For the ρ-VEX, the choice has been made to implement a hardware managed TLB. In Section 2.5.4 systems were discussed that manage their TLBs in software. One notable example is the ST200 processor that uses an ISA closely related to the one the ρ-VEX uses. Managing the TLB in software does not avoid much area utilization and complexity since the TW hardware is rather straightforward to implement. It does however greatly increase the latency of a TLB miss because it will generate a trap and needs to execute

Table 3.1: The advantages and drawbacks of the memory hierarchies discussed in this chapter.

Type	Pros	Cons
PIPT	- Cache design is unaffected - Simplest to implement	- Increased pipeline length
VIPT	- Cache design is only slightly affected - Simple to implement	- Limits page size
VIVT dual directory	- TLB not in critical path	- High complexity - Increased cache area - Cache invalidation on context switch and page table changes - Paired eviction
VIVT multilevel	- TLB not in critical path - L2 cache shieds the L1 cache from coherence traffic	- Highest complexity - Requires at least two levels of cache - Slightly increased increased cache area - Cache invalidation on context switch and page table changes - Paired eviction (but less often)

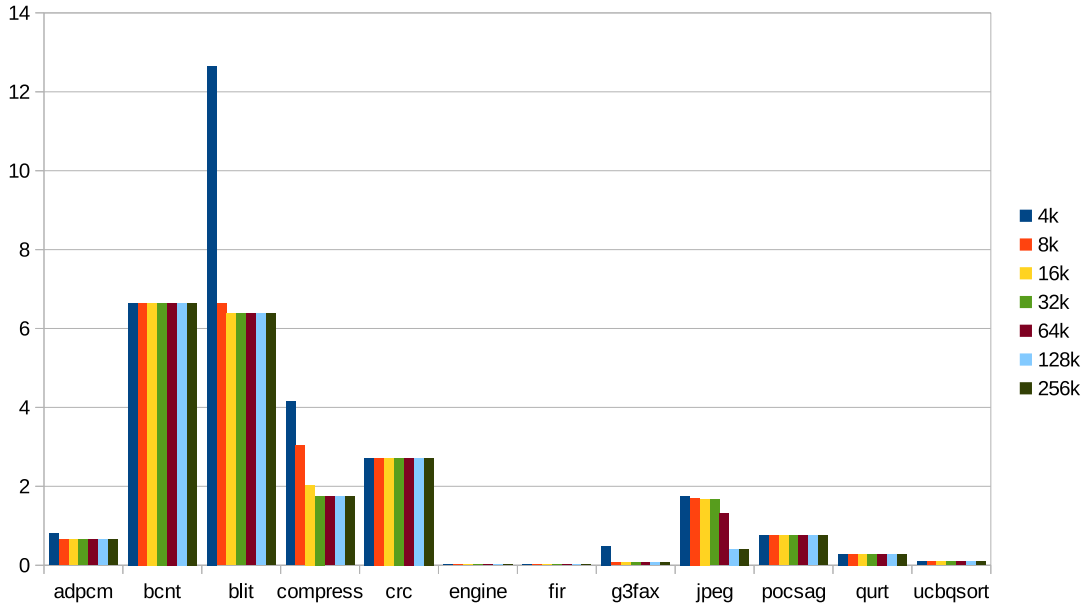


Figure 3.13: Powerstone benchmark results for different cache sizes. The y-axis denotes the data cache miss percentage.

possibly uncached ISR code. Apart from the cycles spent waiting for the main memory when accessing the PT, a TLB miss serviced by a TW takes tens of cycles where a miss in a software managed TLB can take hundreds of cycles and can be highly variable [31]. The main reason why many systems still implement a software managed TLB is that a

hardware TW module fixates the format of the PT and forces the OS to comply to this. However, in the case of a softcore processor like the ρ-VEX, it is relatively easy to adapt the behaviour of the TW module to other PT layouts.

3.7 Optimization Opportunities

In the ρ-VEX system, the MMU component is a single subsystem encapsulating every TLB in the system. This is unusual because generally each core has its own MMU with its own TLB(s) and a TW. See Figure 3.14 for a diagram of the memory hierarchy of an Intel Core i7 processor. The fact that in the ρ-VEX system all the TLBs are close together allows for some sharing of resources. For instance, all TLBs can share a single TW unit. Since a table walk occupies the bus, it does not have any benefit to instantiate multiple TW units since they cannot operate in parallel. Another interesting opportunity is in maintaining TLB coherence. The TLB shoot-down scheme used by many system, detailed in Section 2.5.6 is not needed in the ρ-VEX. Because the TLBs are tightly coupled to each other, it is possible to broadcast TLB invalidations to every TLB present in the system thereby updating them all at once. Issuing a flush command in this way stalls all lanes until all TLBs completed the flush operation, making it an atomic operation. This mechanism makes TLB coherence fast and completely transparent to the programmer. The following lists summarizes the optimizations proposed in this section:

- Quick and transparent TLB coherence.
- Updating multiple instruction TLBs simultaneously.
- Sharing a single TW between all TLBs.

3.8 MMU Support for Dynamic Reconfiguration

The dynamic reconfigurability of the ρ-VEX is what makes this project interesting and what takes it beyond simply implementing existing ideas on a new platform. This ability of the ρ-VEX system places some challenging requirements on several subsystems including the MMU. On the other hand, it also provides opportunities for optimizations which are not possible in more general processing platforms. These aspects of the MMU that are unique for the ρ-VEX, can be divided into two categories: requirements and optimizations. Requirements are the necessary abilities the MMU must posses to support the ρ-VEX in its dynamic reconfiguration ability. The optimizations are not essential for the functioning of the MMU, but are desirable features. These optimizations allow for efficient use of the instantiated resources and can increase the performance of the address translation hardware.

Requirements
The primary ρ-VEX-specific requirement of the MMU is that it must be able to support dynamic address translation for a single wide core or multiple smaller cores. Moreover it must be able to switch between these functionalities at run-time.

From the TLBs point of view, it does not matter how lanes are functionally coupled inside the core. Fetch operations and data accesses are issued per lane and need to be handled irregardless. The fact that different hardware context can run on different lanes can be solved by including ASIDs in the TLBs entries. These allow the TLBs to distinguish between different contexts address mappings. The responsibility of changing these accordingly on context switches lies with the software.

The TW also needs to be aware of the current configuration to service TLB misses in a correct way. When one of the TLBs asserts a miss, the TW needs the Page Table Pointer (PTP) of the context currently running on the corresponding lane. this information is required to search the correct PT for the requested address mapping.

Also dependent on the configuration is the amount of address translations which must be performed per cycle. When the core is configured as a single large issue core, only one data operation can be performed every cycle. In other configurations, however, multiple data operations can be requested simultaneously. To accommodate for the most demanding situation, where the amount of cores is the highest, the ρ-VEX MMU must have as many data TLBs as the maximum number of cores. How to make use of every data TLB in the system even when only one request is issued every cycle is considered an optimization and will be discussed in Section 4.7.1.

Instruction fetches are also influenced by the configuration. In any configuration, every lane generally fetches an instruction each cycle. But when the ρ-VEX is configured as a single wide issue core, it would be possible to translate each instruction address with a single TLB if instruction bundles are page aligned. When multiple cores are active however, each cores instruction address is very likely to fall in a different page. Therefore, the MMU also needs as many instruction TLBs as the maximum number of cores to avoid the MMU becoming a bottleneck.

Optimizations

As described above, the reconfiguration ability of the ρ-VEX requires that both a data and instruction TLB is present for each of the smallest supported lanegroups in the system. While the instruction TLBs are referenced every cycle irrespectively of the configuration, the addition of multiple data TLBs to the system can be considered wasteful when they are not used in all configurations. The mechanism proposed to use these resources in every configuration is to merge data TLBs together with the issue lanes. When issue lanes are coupled together to form a wider core, the virtual tag of the data request made in one of these lanes can be broadcasted to the data TLBs of all coupled lanes. If any of the TLBs holds the corresponding physical tag, the data operation can be executed immediately. This effectively increases the size of the cores data TLB together with the issue width.

This proposed mechanism leads to another optimization which will be implemented and evaluated. When multiple data TLBs are available to a wide issue core, this grants the freedom to decide which of them is preferred to be updated on a miss. Consider a situation in which a process is running on a wider core and it is known it will migrate to a smaller core on the next reconfiguration. If the process is able to direct all its TLB updates to a lane which it will be running on in the future, the process can ensure that it will not need to repopulate its TLB after the the migration. This mechanism

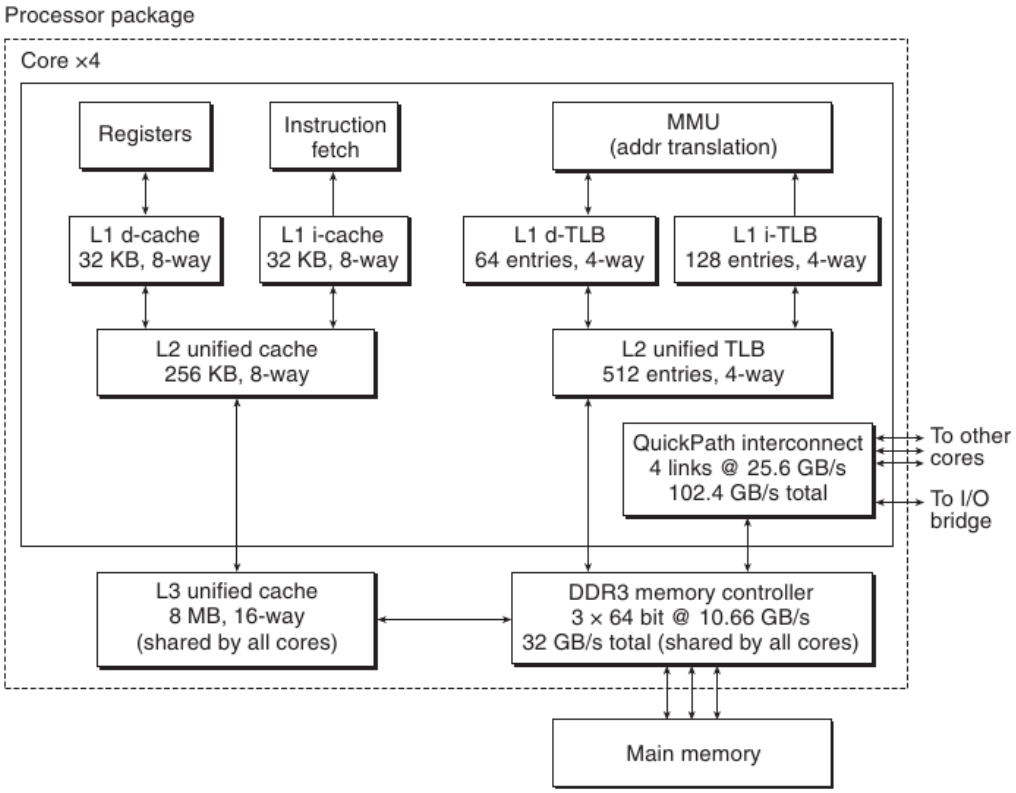


Figure 3.14: This diagram shows the memory hierarchy of the Intel Core i7 processor. This diagram illustrates that multi-core processors generally have a separate MMU for each core. This image is taken from [2].

can mitigate some of the latency costs of of migrating processes after a reconfiguration is performed. This functionality is inspired by other work conducted in the CE group where this mechanism is implemented in the ρ-VEX cache.
To summarize this section:

Requirements:

- The MMU should support all possible configurations.
- The MMU should be able to switch between these configurations during run-time.

Optimizations:

- Using the instantiated TLB units efficiently in all configurations.
- Supporting TLB update direction.

3.9 Conclusion

In this chapter the high level design choices regarding the ρ-VEX MMU have been explained and substantiated. First an overview was given of the different memory hierarchies possible in a system with both virtual address support and caches. After weighing the advantages and drawbacks of each of these architectures, the choice for a VIPT type was cache explained. This type of cache avoid most of the complexities associated with a VIVT cache, especially in a multicore system like the ρ-VEX. Additionally, because address translation is not required before indexing the cache, the VIPT cache does not increase the length of the pipeline like a PIPT cache does. The main drawback of a VIPT cache is that its size is restricted by the page size. This can be extended by increasing the set associativity of the cache. Other ways to circumvent this restriction are increasing the page size and page coloring. In the last sections of this chapter the implications of the ρ-VEX dynamic reconfigurable nature for the MMU were discussed. The challenge lies in using the instantiated resources efficiently in all configurations. In the chapters last section, a few additional optimizations were proposed. These are relatively easy to implement and do not incur a high price complexity wise. These optimizations are possible in the ρ-VEX because the TLBs are tightly coupled to each other.

4 Hardware Implementation

After having discussed the most important design choices of the ρ-VEX Memory Management Unit (MMU) on a conceptual level in Chapter 3, this chapter focusses on the individual components of the MMU, their implementation in reconfigurable fabric, and the interfaces there components have to each other and the rest of the ρ-VEX system.

This chapter starts by presenting the target implementation platform for this project in Section 4.1. Section 4.2 discusses the interfaces the MMU has to the core and the cache. The modifications to the cache which are required for interoperation with the MMU are listed in Section 4.3. After these preliminary topics are discussed, the actual MMU hardware can be discussed. Section 4.4 describes the Translation Look-aside Buffer (TLB) and its subcomponents. In Section 4.5, the Table Walker (TW) is explained. Section 4.6 discusses the MMUs design-time configuration features. Section 4.7 describes the mechanisms implemented in the MMU to support run-time reconfiguration. In Section 4.8 the layout of the page table in memory is explained. This is not fixed because it is dependent on the configuration of the MMU. Finally, in Section 4.9 and overview is given of the control registers which are added to the system to control, and interface with the MMU.

4.1 Implementation Platform

The hardware platform used in this project is the ML605 evaluation board by Xilinx [32]. The board features a Virtex 6 Field-Programmable Gate Array (FPGA), specifically the XC6VLX240T from their high performance Virtex series. Besides the FPGA the board holds many peripheral components to extend its functionality. It features a wide range of input and output interfaces and it can be programmed over Universal Synchronous Bus (USB) without needing an external programmer. One feature that is useful for the ρ-VEX system is the Double Data Rate (DDR) Random Access Memory (RAM) slot which allows the ρ-VEX to access a large main memory.

The ρ-VEX core has been coupled to the Gaisler Research Library (GRLIB) platform [11] by students who have worked on the ρ-VEX system in the past. GRLIB is an open source Intellectual Property (IP) library designed for System-on-Chip (SOC) development. The platform is centred around an AMBA bus originally designed by ARM and contains a library of peripheral components. Figure 4.2 depicts an example of a SOC build with GRLIB. In the diagram, the bus master is a LEON3 processor which is also part of GRLIB. In our system, the ρ-VEX processor takes its place as bus master. The peripheral components that are relevant for this thesis project are the DDR RAM since it holds the (virtualized) memory and the Universal Asynchronous Receiver/Transmitter (UART) which is used for uploading programs and outputting results and status information. GRLIB also features a timer unit and an interrupt controller.

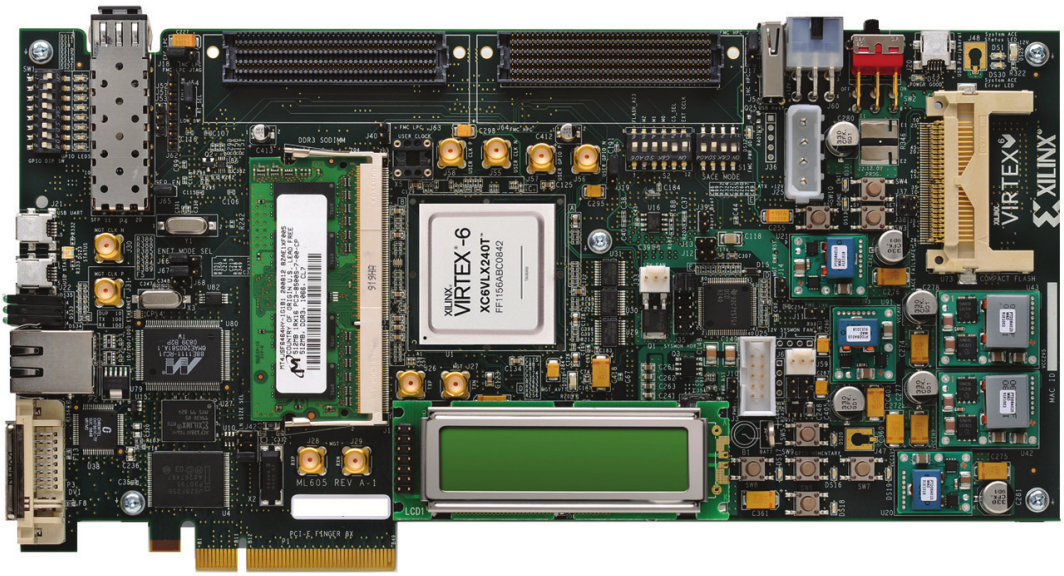


Figure 4.1: The ML605 development board

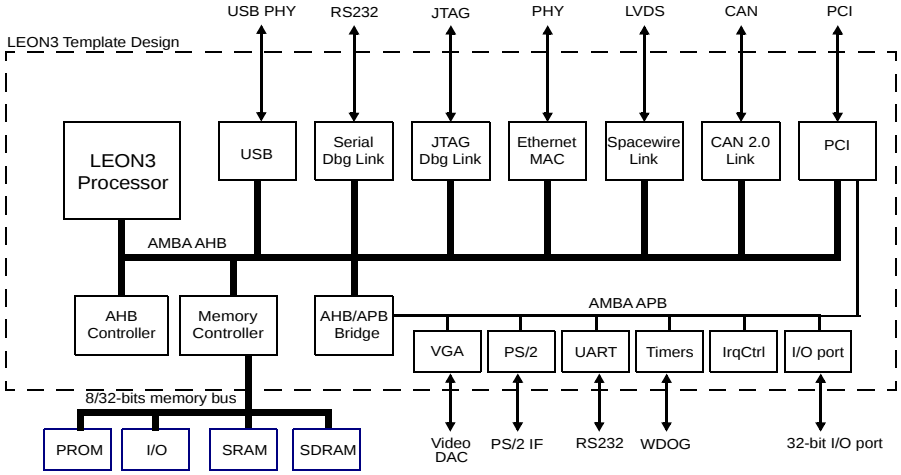


Figure 4.2: An example of a SOC based on the GRLIB platform.

These components will be used to supply the verification software, described in Chapter 5, with a periodical timer interrupt used for task switching.

4.2 Interface to the Processor and Cache

A simplified diagram of the MMU's components and the most important internal and interfacing signals is shown in Figure 4.3. In the diagram only one memory pipeline is shown. In the ρ-VEX processor the path from the core, through the TLB, to the cache is duplicated for every pipeline. The TW is not duplicated because servicing TLB misses

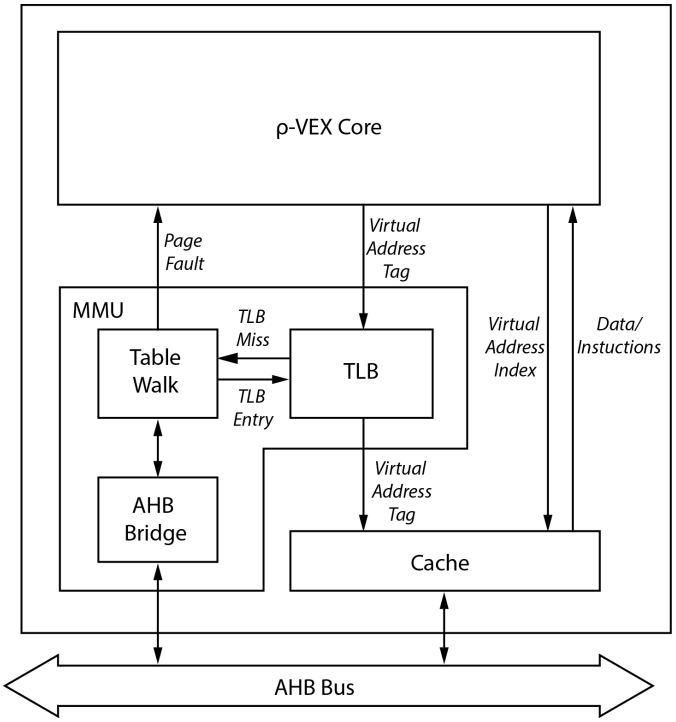


Figure 4.3: This diagram shows where the MMU is situated in the memory hierarchy. Also shown are the most important internal and interface signals.

requires memory access and thus must be serialized anyway. In Chapter 3, a overview of different memory hierarchies was given and the best place to insert the TLB was selected. Optimizing for speed and simplicity, the TLB will be located alongside the cache so that the cache lookup and page tag translation can be done simultaneously. This turns the cache into a Virtually Indexed Physically Tagged (VIPT) cache.

Stall Signals

Besides receiving and supplying tags from and to the ρ-VEX pipelines and cache respectively, the MMU also receives and asserts various stall signals top and from the core and the cache. These signals are used to synchronize the operations of the core and the memory subsystems, the MMU and the cache. When either the cache or the TLB misses, the ρ-VEX pipeline that issued the memory request and all coupled lanes must be stalled until the miss is serviced. When the TLB services a miss, the MMU must also stall the cache. This is necessary because the cache cannot perform a tag comparison when the TLB is not able to supply the correct physical tag.

Memory Interface

To update the TLB when a miss occurs, the TW must access the Page Table (PT) located in main memory. To do so, the MMU requires an interface to the AHB bus. The MMU uses the same type of AHB bus bridge as the cache. This means that a ρ-VEX system, which includes the MMU, has an extra bus interface compared to the baseline

ρ-VEX implementation.

Register Interface
Configuring the MMU is done through a memory-mapped register interface. It can be used to enable the MMU and to inform it of the current configuration of the ρ-VEX core. This is necessary because the MMU needs to know which contexts are running on which issue lanes. The process-specific parameters, the Page Table Pointer (PTP) and Address Space Identifier (ASID) are also supplied through this interface. These can be changed upon a process switch to let the MMU translate addresses issued by the lanes into another address space. Flush requests are also be submitted through the MMU control registers. A detailed overview of the MMU specific (context) control registers, including a description of the flushing interface, can be found in Section 4.9.

Memory Transaction Model
The memory transaction model of the ρ-VEX system that includes the MMU is an extension of the baseline version, which includes the cache. This section will detail the interface between the core and the cache, which partly runs through the MMU. The interface between the cache and the AHB bus falls outside the scope of this project. In the old situation, memory reads for both instruction and data were implemented as two cycle transactions that could be stretched by the cache. Memory writes always took a single cycle because the cache has a write buffer. In the first cycle the address and control signals are asserted in combination with the data in case of a write. In case of a read, the data is returned in the second cycle. The second cycle of a memory transaction is overlapped with the first cycle of the following. Each transaction can be stretched by the cache when it is not able to return the data, which happens when the cache encounters a miss. To do so, the cache has a stall signal it can assert. This halts the core until the data becomes available.

In the new system the MMU can also encounter a miss, which requires a similar stretching mechanism. When a TLB encounters a miss, both the core and the cache are stalled until the miss is resolved. Note that in the new situation, stalls can also happen in case of memory writes. When the TLB hits, the latency of the TLB is hidden and no extra cycles are needed.[1]TLB misses either results in a successful table walk, after which the TW can update the TLB, or alternatively, in a pagefault. When a pagefault is generated, the MMUs stall signal to the core is de-asserted so the core can register the trap.

In Figure 4.4 a timing diagram is drawn which shows these different situation in more detail. The diagram shows a sequence of six instruction fetches, which illustrate the different situation that can occur. The following list refers to the instruction fetches, which are similarly named in the diagram:

A) A normal instruction fetch without any stalls.

B) TLB miss with a successful table walk.

C) A cache miss.

D) An instruction page fault.

E) An instruction fetch cancelled by the trap generated by the previous fetch.

F) The first instruction of the trap handler.

In the diagram all core signal are coloured black, the internal MMU signals and signals from the MMU to the core and cache are colored blue, the signals from the cache are coloured red. Both the core and the MMU keep their requests valid when a component downstream in the memory hierarchy asserts its stall signal. Note that the *rv2mmu_Ptag* and *rv2cache_index* operate in lock-step. This is because these are simply different slices of the address issued by the core. While the diagram shows the data path for instruction fetches, the mechanisms are the same for data reads and writes.

4.3 Cache Modifications

The general operation of the cache is not changed by the addition of the MMU. There are however some adaptations needed to make the cache function in conjunction with the MMU. These changes are mostly because some of the signals that originally ran directly from the core to the cache, are now routed through the MMU. This applies foremost to the address tag which needs to be translated by the MMU. In the design without the MMU, the tag is simply delayed by one cycle inside the cache while the index is used to perform the cache lookup. This is required because the tag comparison can only be done after the lookup is performed, which takes one clock cycle. In the new design, the tag must be translated, which in the general case of a TLB hit, also takes one clock cycle . Because of this, the cache does not longer need to delay the tag, but can directly use it the cycle in which it is output by the MMU.

Another new scenario is when the MMU fails to translate the tag in single cycle because of a TLB miss. This leads to either a table walk or a pagefault. In either case the cache must be made aware that it must wait until the tag becomes available. To support this, the stall mechanism is expanded to include a stall signal from the TLBs to the cache.

[1] see the discussion in about virtual memory hirchies and the benefits of a VIPT cache in Section 3.4

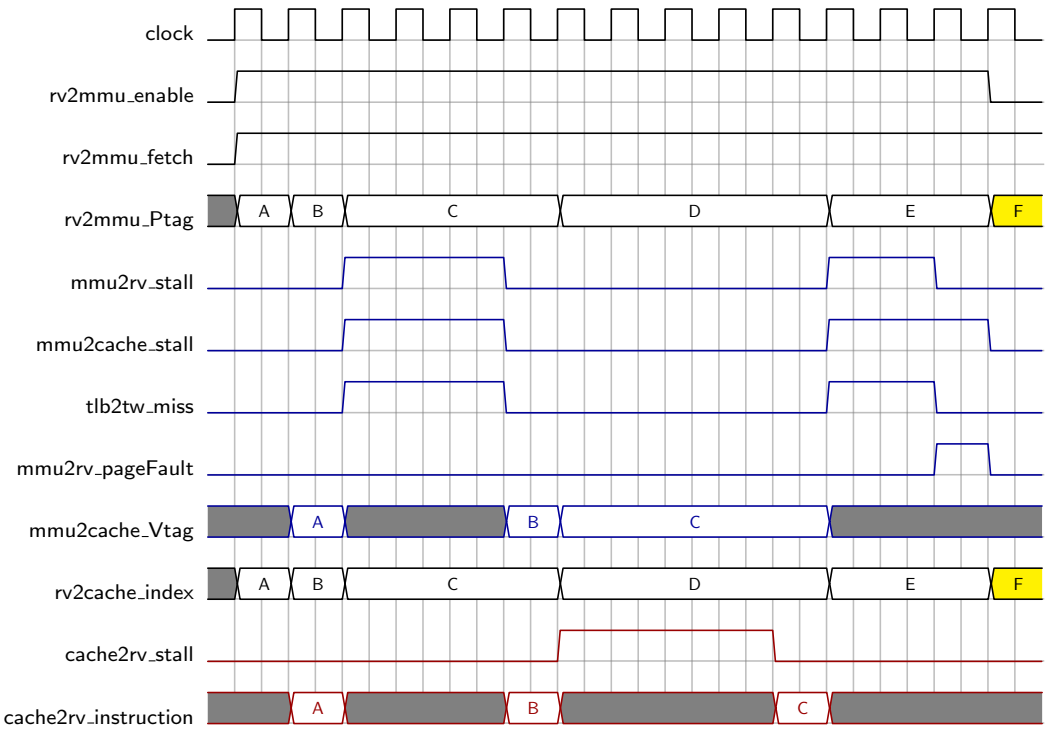


Figure 4.4: Timing diagram of the instruction fetch interface of the ρ-VEX.

4.4 Translation Look-aside Buffer

The TLB is the most important component in the the MMU. This is because it lies in the critical path for instruction fetches and data reads and writes. Other components like the TW are only involved in a small fraction of memory accesses. Because of this, TLB latency is an important metric. As explained in Section 3.1, TLBs are usually implemented in the form of Content Addressable Memory (CAM), sometimes referred to as fully associative memory. This enables single cycle lookups. Because the TLB is in the memory path of the processor, every extra cycle needed to perform a TLB lookup also requires an increases of the Central Processing Unit (CPU)'s pipeline length. Besides speed considerations, the TLB is also the most the most complex component of the MMU. Figure 4.5 shows a schematic diagram of the different components that compose the TLB.

CAMs are expensive components in terms of hardware resources on any implementation platform. This holds true for Application-Specific Integrated Circuit (ASIC) designs, but even more so for FPGAs. The reason is that the FPGAs resources are not well suited to implement this type of memory. Because the CAM is the heart of the TLB, this component will be discussed first.

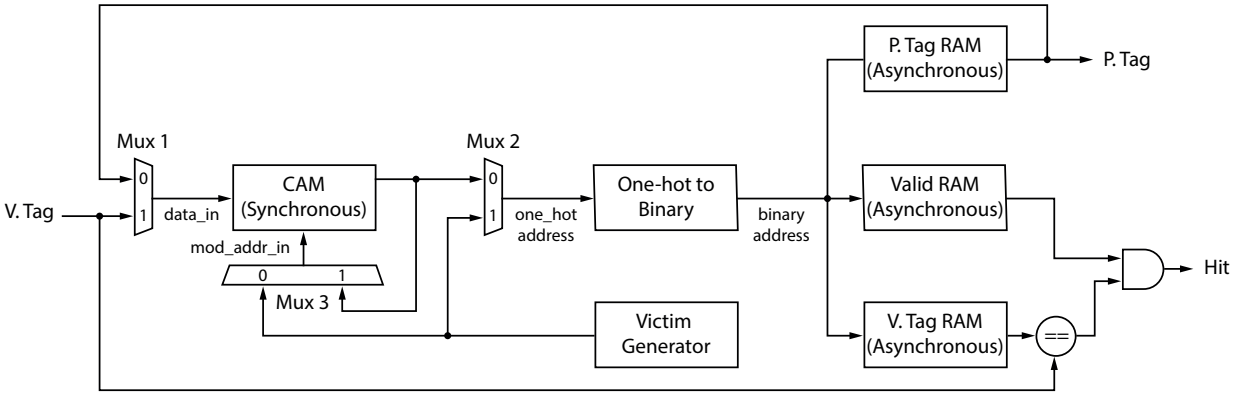


Figure 4.5: Simplified connection diagram of the TLB.

4.4.1 Context Accessible Memory

A CAM, sometimes referred to as fully associative memory, is a type of computer memory that is used in high speed applications. It differs in the way it operates compared to regular memories such as RAM. Where normal memories receive an address and return the data stored in that particular location, CAMs are supplied with a search key and subsequently search the entire memory for any matches. The memory then returns either the address of the (first) match or some word of data associated with the search key.

In an FPGA, there are three ways to implement a CAM [33]. They can be implemented using Lookup Table (LUT)s, registers, and BRAMs. The LUT approach is limited in the sense that the entries can only be changed by programming the FPGA and so it is read-only during run-time. This is not suitable for a TLB because it must be both readable and writeable. The register approach is basically an array of registers each with a comparator circuit. This can be a straightforward and fast implementation for small CAMs with narrow entries. For larger memories however this quickly becomes quite expensive as the resources scale more or less linearly with the entries width and the

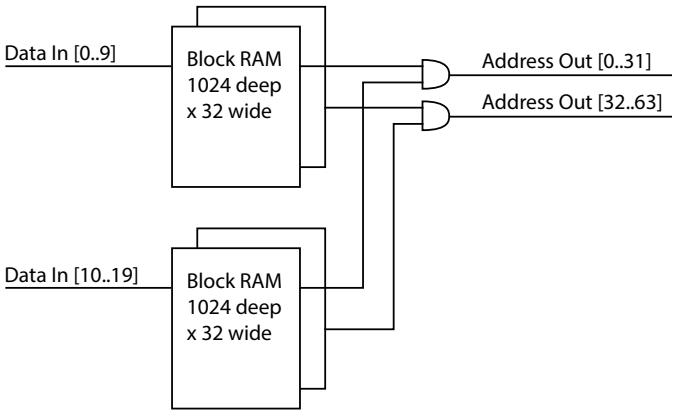


Figure 4.6: This diagram shows how deeper and wider CAMs are build from multiple smaller CAMs implemented in BRAMs.

memories depth. RAM based CAMs scale to the size of the memory in a similar fashion but mostly use up BRAM resources and are capable of supporting much larger memories operating at speeds similar to the register based approach. Because most FPGAs have an abundance of BRAMs and they are not heavily used in the ρ-VEX, the decision was made to use the latter approach. This also enables more freedom to parametrize the depth of the TLB which is a useful feature in a reconfigurable processor. It is also useful in the scope of this project to measure the effect the TLB depth has on the ρ-VEX performance.

The BRAM blocks available in the FPGA are fundamentally different from how hardware CAMs would be implemented. Luckily there are ways to still use them to implement fully associative memories. Xilinx provides an application note [34] detailing how to do this. Most of the ideas used in the ρ-VEX TLBs CAM come from this document. Where a normal RAM receives an address and returns the word of data stored at that location, a CAM works the other way around and receives a word of data and subsequently returns the address of the location containing the data. A RAM can be used in such a way by using the data word as an address and effectively creating a unique location for every possible search key. This memory location can then be filled with the address or data associated with the key, or a pointer to a secondary memory holding these. The problem when implementing this naively is that the size of the RAM grows exponentially with the input data's width. When supplying the CAM with 20 bits wide virtual tags to find the corresponding physical tag (also 20 bits wide), this amounts to a memory with two million 20 bit entries, which is the equivalent of 72 32 KiB BRAMs.

To implement a CAM that scales better with the memory's width and depth, the CAM is not implemented as one large contiguous memory, but by connecting multiple smaller ones in a smart way. The building blocks are single BRAM blocks configured as 1024 32-bit memories. This can be used as a CAM with 10-bit wide search key and 32-bit output addresses. The entries in the CAM are one-hot encoded, where each bit represents a location in a secondary memory holding the associated data.

To support deeper CAMs, with more than 32 entries, the BRAM can be duplicated with each block receiving the same input but storing one-hot encodings for another 32 entries. The outputs of the BRAMs can be concatenated to form a large one-hot vector.

To increase the CAMs width, the search key is cut into slices of 10 bits wide. Each slice is then routed to one of the BRAM, which effectively matches the slice of the input data to the same slice of the CAMs entries. To check if the entire input data matches any of the CAMs entries, the one-hot encoded output vectors of the different BRAMs are AND-ed. If all slices of the input data match for a single entry in the one-hot encoded vector, there will be a '1' in the corresponding location after the AND operation. Figure 4.6 shows a CAM which can hold 64 20-bit entries. It is a single BRAM, duplicated once to double the depth, and once to double the width. This composite CAM thus uses four BRAM resources.

4.4.2 TLB States

The TLB has multiple modes of operation. In Figure 4.7 a simplified state diagram of these different modes is depicted. The state of the TLB determines how the muxes in

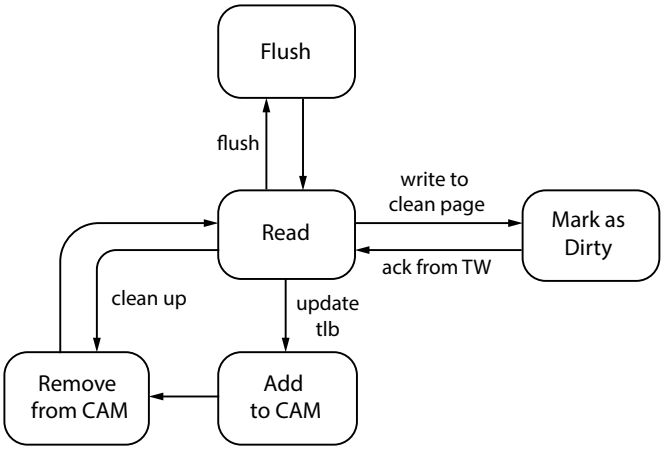


Figure 4.7: Simplified state diagram of the TLB.

Figure 4.5 are configured. This dictates how data flows between the different components and the inputs and outputs of the TLB. In this section the TLBs modes of operation are explained.

TLB Reads

Read mode is the default state of the TLB. In this mode the TLB receives virtual tags and outputs physical ones. It stays in this mode as long as the TLB holds the requested translations. Translating the tag of an address takes a single cycle. As explained in Section 3.4.5, this happens in parallel with the cache lookup which uses the index part of the address. The states of the muxes as depicted in Figure 4.5 are all set to '0'. The state of the third mux is actually not relevant in this mode because it is only used for updating the contents of the CAM, which is not needed in read mode.

The data path of the TLB in this mode is as follows. The virtual tag is concatenated with the current ASID and they are supplied to the CAM. The CAM then searches its entries and outputs the address it found at the location of the match. The address is one-hot encoded as explained in Section 4.4.1. After the address is translated to binary format by the one-hot to binary converter, the address is used to index three RAM memories also present in the TLB. These RAMs support asynchronous reads so a read operation of the TLB of a whole takes a single cycle. One of these holds the physical tag which is output in case of a hit. The other two are used to check if the translation found is actually valid. The corresponding bit in the valid ram must be set for the translation to be valid. When the Operating System (OS) needs to invalidate TLB entries it can clear bits in this RAM to do so. Because the contents of the CAM memory are not always known, for example when the ρ-VEX has been reset, it is possible for false positive hits to occur. Therefore the third RAM is used to check if the CAMs match actually corresponds to the virtual tag and ASID that where used to initiate the read. If these also match, the hit signal is asserted signalling a correct address translation. When a false positive is discovered, a clean up action is required to clear the RAM of the stale entry. This mechanism is further explained in Section 4.4.3.

As long as the TLB hits, its operation is continuous, only delaying tags by one cycle. When a miss occurs however, the TLB moves to another state and it is not able to service reads. When this happens, the stall signal to the processor is asserted, stalling the pipeline until the TLB is updated. This is also the case when the TLB services a flush request. It is also possible that no address translation is required for memory accesses. This can happen when the processor operates in *real* mode as opposed to *virtual*. This happens on boot-up, when the initializations required for virtual address spaces is not completed yet. Some systems also use real addresses when operating in kernel mode, bypassing the MMU. Bypassing the TLB can also be useful when writing to memory mapped devices such as an UART.

TLB Updates
When a TLB miss occurs the address translation cannot be completed. The contents of the TLBs memory must be updated to enable further execution. All writes to the TLB are initiated by the TW. When a new entry is added to the TLB, this sometimes requires that an old one must be removed. The *victim generator* component tries to select an empty or invalid entry and otherwise randomly replaces a valid entry. The *victim generator* supplies the address of the victim in one-hot encoding. Muxes 1 and 2 in Figure 4.5 are set to '1' the mux 3 is set to '0'. When the mux 2 is set to '1', the victims physical tag is output from the RAM. This tag is looped back to the CAM and is used to remove the victim address from the entries corresponding to the victim physical tag. This is a read-modify-write operation. After the victim address is removed from the CAMs entries that belong to the victim, the same address can then be added to the entries belonging to the replacement. To do so, the mux 1 is set back to the '0' position. This directs the updated address to the replacements entries. This operation is also a read-write-modify operation. In total this brings the amount of cycles needed to write a new entry to the TLB to four cycles.

TLB Flushes
Whenever the OS makes changes to the PT, all the TLBs present in the system must be updated too. If this would not happen, the outdated information in the TLB could lead to invalid memory accesses. The mechanism used is to simply invalidate all TLB entries affected by the change(s) to the TLB. The TW can then repopulate the TLB with the updated entries whenever a TLB miss occurs. As can be seen in the TLB state diagram depicted in figure 4.7, flushing the TLB halts all other activities such as reading for as long as the flushing process takes. Flushing the TLB can be done in a number of ways. The most straightforward one, which is also the quickest way, is to flush all the entries at once. This is done by writing *zeros* to all entries of the *valid RAM* depicted in figurer 4.5. This RAM is implemented using LUT resources, which makes it possible to clear all entries in a single cycle. The CAM is updated through the deferred cleaning mechanism explained earlier. While this method is the fastest in the short term, often it is rewarding to be as conservative as possible with invalidating entries, since it generally leads to more misses. To facilitate this, the TLB can also be flushed in a more fine grained fashion, in which every entry is compared to parameters supplied through the register interface. For instance, all entries which have a specific ASID can be flushed, effectively removing all

translations belonging to a single process from the TLB. Alternatively, all translations in a range of address tags can be flushed. These mechanisms can also be combined. When instructing the TLB to flush in this way, the process takes longer than when the entire TLB is flushed. This is because all entries must be compared to the supplied parameters one by one. Therefore, the amount of cycles needed has a linear relation to the TLBs depth. The exact modes and parameters available for flushing the TLB are described in section 4.9.

4.4.3 Stale CAM Entries

Sometimes the CAM can become polluted with garbage entries. When the ρ-VEX system powers up, the contents of the CAM are clean. When the system is reset however, all the old entries are still present. Another situation which leaves stale data inside the CAM is when a TLB entry is invalidated. This is done by clearing the valid bit in the valid RAM in the TLB, leaving the data still in the CAM. These situations can easily lead to false positive hits.

When the a false positive is identified, the TLB must remove the false (one-hot) address from the entries in the CAM corresponding to the search key. To do so the third mux in Figure 4.5 is set to '1'. The logic surrounding the CAM (not depicted) is then instructed to remove the address bit from the one-hot vector. This is a read-modify-write operation and thus takes two cycles. This mechanism of deferred cleaning of the CAM is useful because stale entries only need to be removed when they cause trouble, which saves cycles. This mechanism also enables single cycle TLB flushes.

4.4.4 Large and Global Pages

Most MMUs support large and global pages. Large pages are used to allocate portions of memory which are larger than a single page. This allows a single mapping to be stored in the TLB thereby saving precious space. Global pages are virtual to physical mappings that are valid for every process. Global mappings are usually never chosen as victim by the TLBs replacement policy.

In Section 4.4.1 it is explained that the CAM memory in the TLB is created by using multiple BRAMs that act as a CAM for a specific slice of the input. The output of each CAM is a one-hot encoded address vector. These vectors are then AND-ed to form the address returned by the CAM as a whole.

This approach facilitates implementing tag matching for large and global pages. Recall from Section 2.5.2 that PT lookups are performed in multiple steps. In each level, a part of the address tag is used as an index in a table. The entry in the table points to the next level or in the last level, holds the physical tag. These tag slices are referred to as a level one (L1) or level two (L2) tag.

For normal pages the virtual address needs to match both the ASID and the entire tag of one of the entries in the TLB. For large pages however, only the L1 tag needs to be matched. The L2 tag is treated as *don't care*. Similarly, for global pages, the ASID should be ignore when matching.

It is not possible to implement a ternary [2] CAM using BRAMs [34]. To still support large and global pages, a method was designed in which entire CAM blocks are ignored. See Figure 4.8 for a diagram that depicts this method. The input to the CAM is divided into three parts. The ASID, the L1 tag, and the L2 tag. Each slice is connected to a different CAM block. By selectively combining the results of the individual CAM blocks it is possible to ignore parts of the input to the CAM.

This can easily lead to false positive matches for large and global pages. To filter these out, the outputs of the CAM are masked by vectors that indicate which entries of the TLB actually hold large and global pages. A hit in the TLB occurs if any of the masked outputs of the CAM are non-zero. It is also possible that multiple outputs are non-zero. The following priority order is implemented:

1. Large page
2. Regular page
3. Large global page
4. Global page

Non-global pages overrule global ones. This is implemented because they are more specific than global pages. It allows specifying a global page in general and overruling this for a specific process. Mappings for large pages should not occur in parallel with mappings for pages that fall within the large page. This is the responsibility of the OS. When a L1 PT entry is flagged as being a large page, this actually makes all pages that fall within this page inaccessible in the PT. Since the pointer to the L2 PT is replaced with the mapping of the large page. To still be able to function when the OS ignored this rule, the large page was chosen to have priority.

The CAM blocks are sized such that each can match a 10-bit wide input slice. When a larger slice must be matched, the CAM must be constructed from multiple BRAM blocks. This imposes limits on the size of the ASID and L1 an L2 tags for a minimal implementation area-wise. A minimal TLB implementation uses three BRAMs resources and can be instantiated if the following conditions are met:

- ASID $width <= 10$
- $large\,pagesize >= 4\,\text{MiB}$
- $\frac{large\,page\,size}{page\,size} <= 1024$

These rules enforce that the ASID, L1 tag, and L2 tag fields are not larger than 10 bit wide. They follow from the rules for the sizes of the L1 and L2 tags which will be discussed in Section 4.8.

[2] A ternary CAM which enables specifying bits as don't care. These bits are ignored during matching.

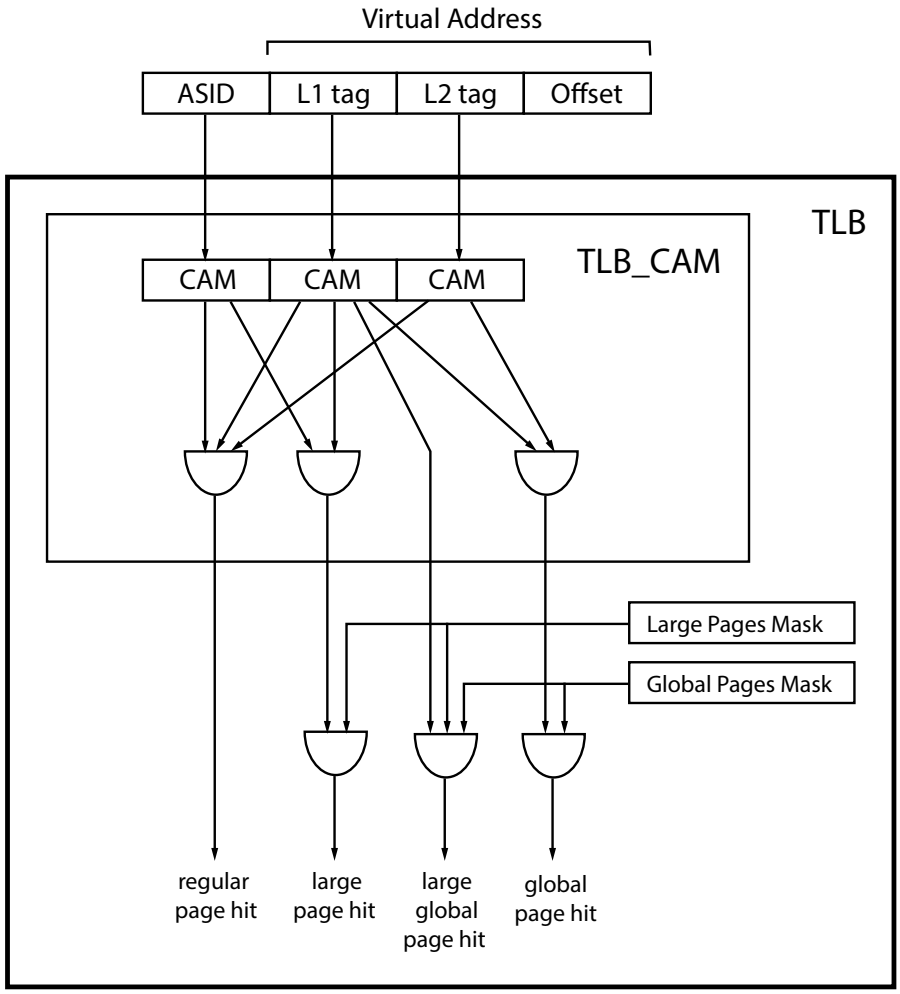


Figure 4.8: This diagram shows how the TLB matches global and large pages by treating parts of the input as *don't care*.

4.4.5 TLB Coherence

In a multiprocessor system supporting virtual address spaces, it is important that each processor operates on the same mappings. For the PTs this issue is trivial since each process operates on a single PT and the coherency problem does not exist. In the case of TLBs, this does become an issue since multiple TLBs in the system can hold the same mappings. Therefore, it is necessary that TLB invalidations are executed in all TLBs in an atomic way.

In Section 3.6, it was already mentioned that the ρ-VEX MMU handles TLB coherence in a way that is transparent to the programmer. This feature is achieved by broadcasting a TLB flush request from one context to every TLB in the system. The MMU then stalls the core and the cache until each TLB signals it is done flushing. It may also occur that multiple contexts, running simultaneously, issue flush requests in the same cycle. This situation is handled by serializing the requests. Since each flush

stalls the entire system until it is complete, and serialized flush request are handled immediately after each other, these simultaneous request are handled in a single cycle from the cores point of view. Stalling the core until all flush requests are handled also ensures the flush parameters of each request are kept stable until it is serviced. The flush parameters can only be changed by writing to the MMUs control registers, which cannot happen as long as the core is stalled.

4.5 Table Walk Hardware

The TW is a rather straightforward component. It consists of a state machine and an interface to the main memory. When one of the TLBs does not hold a translation requested by the corresponding issue lane it asserts its *miss* signal. The TW then uses the virtual tag issued by the corresponding lane to traverse the PT. The pointer to the base of the PT is supplied by the control register interface of the context active on that lane. This process is similar for both instruction and data TLB misses. When the TW finds the address mapping the TLB needs, it updates the TLB and the core can continue. When the PT does not hold a mapping for the virtual tag, a page fault occurred and the OS must load the frame from memory and update the PT. To signal a page fault, a trap signal is asserted by the TW. Because trapping the core automatically disables the MMU, the OS can run its Interrupt Service Routine (ISR) in physical address space.

Searching the PT for mappings requested by the TLBs can be performed using only read operations. There are however some bit flags accompanying mappings in the PT that can be maintained dynamically in hardware which require the TW to write to the PT as well. These bits are the *accessed* and *dirty* bits. For the function of these bits, refer to Table 4.1.

Accessed bits can easily be updated every time the TW retrieves a mapping from the PT. When the mapping has its accessed bit cleared, the TW performs a read-modify-write operation to set the bit in the PT. Dirty bits are somewhat more difficult and require cooperation between the TLBs and TW. These bits are also maintained in the TLB. When a write operation passes the TLB on a page that has its dirty bit cleared, the TLB signals the TW that it needs to set the corresponding dirty bit in the PT. Additionally the TLB sets the dirty bit in its own memory to ensure this procedure only needs to be performed once. In Figure 4.9 the TWs state diagram is drawn.

4.6 Design-time Configurability

The ρ-VEX system is designed to be tailored to specific applications to maximize efficiency in terms of area and performance. Refer to Section 2.4 for an overview of the parameters that can already be modified in the baseline implementation. The MMU has been designed to be parametric in the following ways:

- MMU enable
- Page size
- Large page size

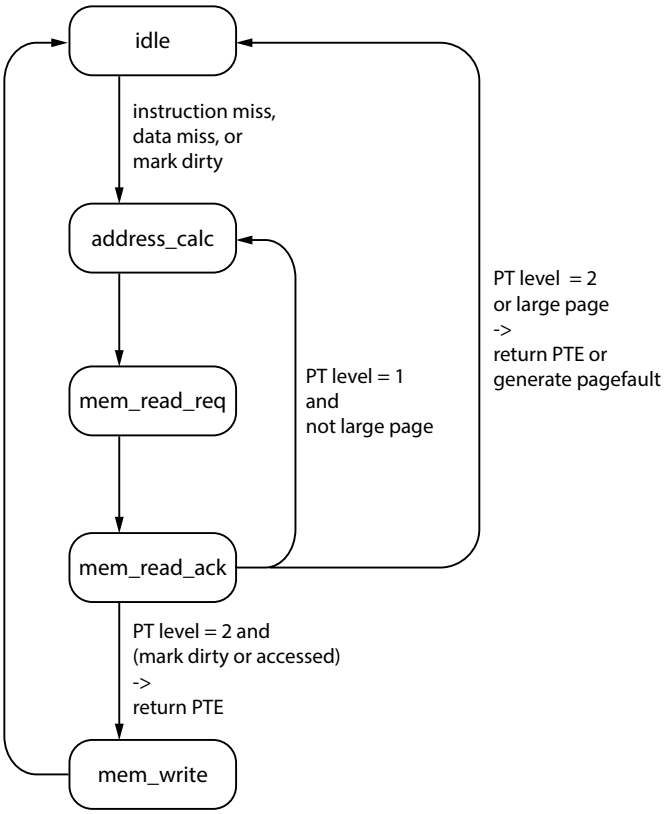


Figure 4.9: Simplified state diagram of the TW.

- TLB depth
- ASID width

The addition of the MMU is entirely optional. When the MMU enable flag is disabled, the system has the same functionality as the baseline implementation and the MMU is not instantiated. The signals which are routed through the MMU will be routed directly between the core and cache. The cache is also dependent on this enable flag since it has slightly different behaviour when the MMU is not present. In Section 4.3 an overview was given of these differences.

The size of pages and large pages size can also be specified. This is useful because some applications can benefit from a larger page size since this potentially decreases TLB misses. Another advantage is that increasing the page size allows for a larger data cache. This relation is due to the VIPT cache and was explained in section 3.5. The MMU implementation only allows one size for large pages but the size can be specified.

Parameters that relate to the TLB are the number of entries each TLB holds and the number of bits available for the ASID. Increasing the TLB depth can decrease TLB misses simply because it can hold more entries at once. The width of the ASID determines how many active processes can be supported without having to reassign an ASID. When this happens it is necessary to flush all TLB entries associated with the

ASID which is reassigned. Deeper TLBs and wider ASIDs are more demanding for the TLBs CAMs resulting in higher resource usage. In its minimal configuration, the TLB uses three BRAMs and supports up to 32 entries and ASIDs not wider than ten bits.

4.7 Run-time Reconfigurability

Besides having many static configuration parameters, the novelty of the ρ-VEX processor is its ability to be reconfigured dynamically during runtime. This feature was discussed earlier in Section 2.4.3. Later in Section 3.8 the implications this feature has for the MMU were discussed. In this section the actual implementation of those features will be detailed.

4.7.1 Coalescing data TLBs

Depending on the configuration of the ρ-VEX as one wide Very Long Instruction Word (VLIW) processor or multiple smaller ones, the issue lanes operate independently or in lock-step with each other. The system that governs this mechanism uses a decouple bit for every lane, which indicates whether the lane runs autonomously or is coupled to its neighbour. This system is already present in the baseline implementation of the ρ-VEX, which is the starting point of this thesis. The reason for this is that the dynamically reconfigurable nature of the ρ-VEX system places similar requirements on the cache as on the MMU. Moreover, the concept of coalescing data TLBs is taken from the design of the cache where the same mechanism is used to distribute data to, and read from, the cache blocks of all coupled lanes.

The ρ-VEX architecture dictates that only one data access is permitted each cycle for each individual core. When one lanegroup in a group of coupled lanegroups issues a data memory access, all associated data TLBs are searched for the address mapping. This is accomplished by a routing network governed by the decouple bits mentioned earlier. In Figure 4.10, a schematic is drawn which depicts this network for a ρ-VEX processor with 4 lanegroups. The depth of the routing network scales logarithmically with the width of the core. The input routing network delivers the data request signals which consist of the virtual tag and read or write enable signals to each TLB. A second output routing network collects the results of all the TLBs. When any of the TLBs holds the requested memory mapping, the core can continue without interruption. If all the TLBs miss, one of them must be updated with the missing address mapping.

Lanes are always coupled to their higher indexed neighbour. Groups of coupled lanes must always be powers of two and must be contiguous. Furthermore they must be aligned by the width of the coupled cluster. These and other rules are defined in the ρ-VEX user manual [35].

4.7.2 TLB update direction

The previous section described how wider cores have multiple data TLBs to their disposal. One question that rises is which TLB to update whenever a miss occurs. In the normal situation a round-robin scheme is used to select which TLB is updated. This

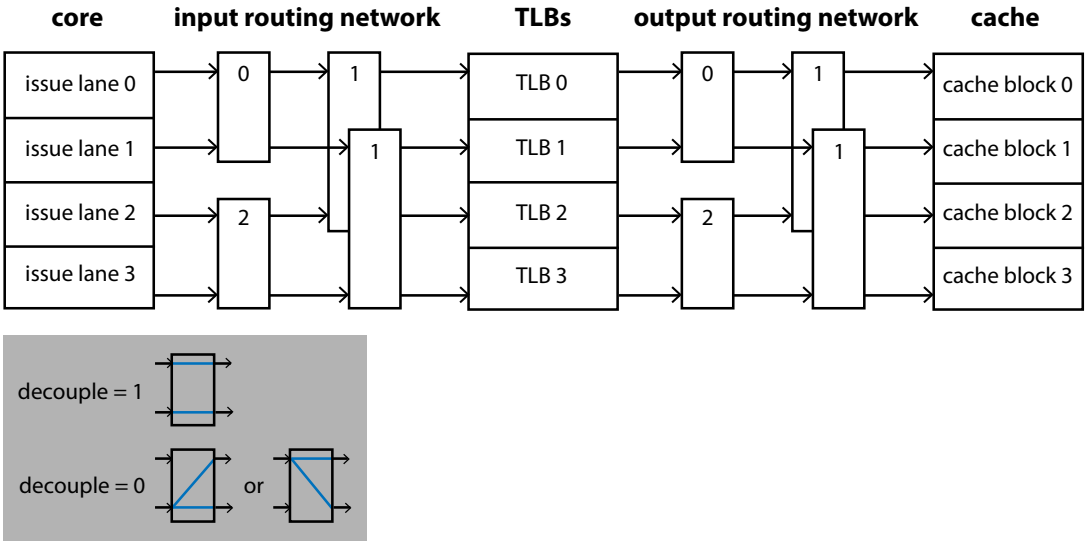


Figure 4.10: The network that routes virtual tags to each TLB in a group of coupled lanes. A similar network is used to route the associated physical tag back to the lane that issued the read or write operation.

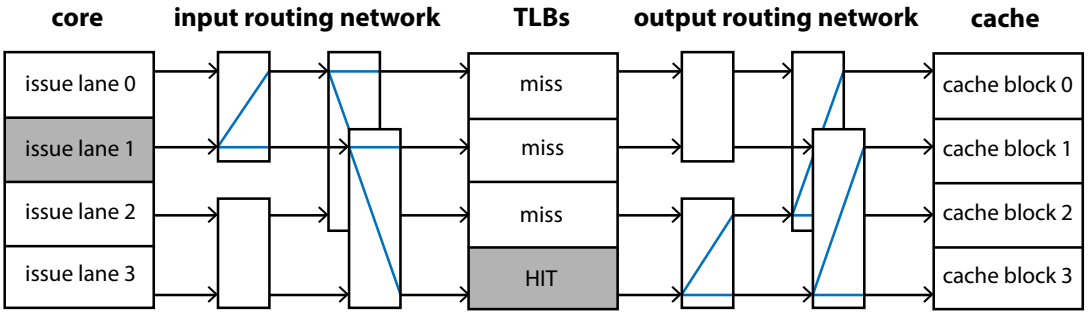


Figure 4.11: This diagram gives an example of how requests for data address mappings are distributed over a group of coupled lanes. In this example, lane 1 issues a data read or write. The virtual tag is routed to all TLBs by the input routing network. How it is routed is based on the read and write enable signals, which can only be asserted by one of the coupled lanes. TLB 3 turns out to hold the requested physical tag. The tag is then distributed to all coupled cache block by the output routing network.

policy is chosen to distribute mappings evenly over the TLBs. It is also possible to specify manually which TLB is preferred to be updated through a the MMUs register interface. This mechanism was discussed earlier in Section 3.8. Directing mappings to a specific issue lane can be useful in a situation where a reconfiguration is immanent. Sometimes it is known that in the near future the current application will be run on a subset of the lanes it is currently running on. If all TLB updates are directed to those lanes, this will avoid any unnecessary TLB misses. The specific implementation of the interface to this mechanism is described in Section 4.9. An evaluation of the benefits of

this feature will be conducted in Section 6.3.3.

4.8 Page Table Organization

In Section 2.5, it was explained why it is not efficient to implement the PT as one large contiguous structure. Instead, the page table is implemented in several levels. While breaking up the PT drastically lowers the memory footprint, each level also adds an extra level of indirection. Virtually all systems which use 32-bit addresses use a two level PT. This is also holds for the ρ-VEX in most configurations. However in the ρ-VEX the sizes of both a regular and large page are determined by parameters. This also influences the sizes of the first and second level page tables. This means that whatever OS is running on the ρ-VEX needs to take these parameters in consideration when traversing or modifying the PT. To facilitate this, a global register is added in which the page sizes of the current configuration can be read out. The L1 and L2 page tables are sized in the following ways [3]:

- $L1\,table\,size = (2^{32} - large\,page\,size) \cdot 4$
- $L2\,table\,size = (large\,page\,size - page\,size) \cdot 4$

The sizing of (large) pages is left up to the designer as much as possible, however some limitations do exist. The following restrictions are imposed on the sizes of regular and large pages:

- $4\,\text{KiB} <= page\,size <= 1\,\text{MiB}$
- $4\,\text{KiB} <= large\,page\,size <= 1\,\text{GiB}$
- $large\,page\,size >= page\,size$

The lower threshold on the page size is imposed by the cache. The upper limit is chosen because the range from 4 KiB to 1 MiB can be specified with a single byte in the control register. Since the upper limit of 1 MiB is a ridiculously high value for a page size, this limitation is acceptable. The lower limit of the size of a large page is the same as a regular page. Sizing a large page smaller than a page would not make sense and is prohibited. When they are set to the same size, large pages are effectively disabled. The maximum size of a large page is 1 GiB.

This freedom of sizing the pages forces the designer to think about the layout of the page table. All configurations are supported by the hardware but some make more sense than others. It is for instance possible to define the page size at 4 KiB and the large page size at 8 KiB. This then would lead to L2 page tables of just 2 entries. In this case it would be better to set them to the same size, removing a level of indirection of the PT. The ranges of the sizes of the L1 and L2 tags and the offset can be found in Figure 4.12.

[3] The last term is the width of a Page Table Entry (PTE) in bytes.

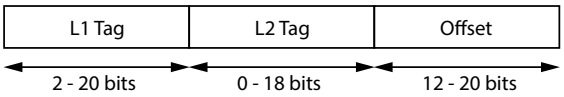


Figure 4.12: The division of addresses into L1 and L2 tags and the page offset is based on the sizes of regular and large pages.

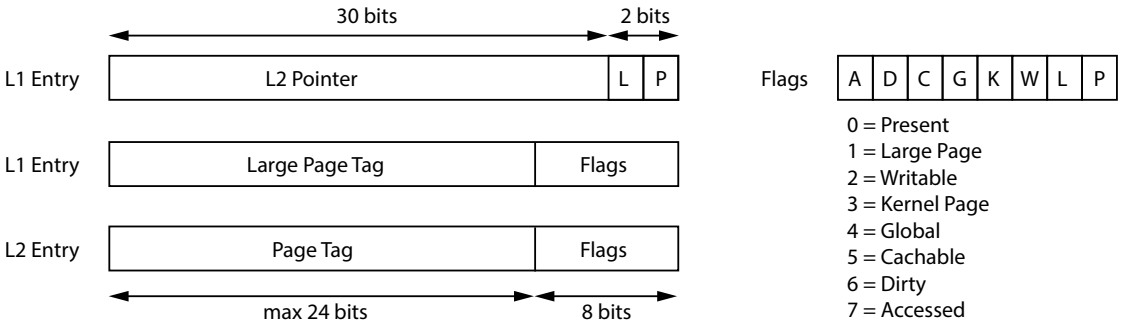


Figure 4.13: Page table entries for the first and second level of the PT.

4.8.1 Page Table Entries

PTEs are different for first and second level tables. In a second level table the entries always hold set of bit flags and a physical tag if the entry is valid. In a first level table, the entry either holds a pointer to a second level table or a physical tag for a large page. The distinction between these two is made based on whether the large page flag is set. The layout of the different PT entries is depicted in Figure 4.13. A listing of the different bit flags is given in Table 4.1.

4.9 Register Interface

The configuration of the ρ-VEX MMU can be changed through a memory mapped register interface. Additionally, the register interface is used to supply the MMU with process specific parameters which are required to map processes to the correct address space. These parameters are the PTP and the ASID. Finally, this interface also provides a fine grained flushing interface for the TLBs. An schematic diagram of all the control registers and their fields is depicted in Figure 4.14. In the rest of this section, each control register's functionality will be explained, grouped by functionality.

Page size register (MMUC)
This read-only register holds the sizes of regular and large pages. It can be read by an OS to determine which page sizes the MMU is configured to implement, so it can adjust accordingly. For the OS it is important to know these parameters because it dictates the layout of the page table and is required information for memory allocation. These registers allow the OS to handle these sizes as parameters which allows the OS to run on differently configured platforms without needing to be recompiled. This is the only

Table 4.1: Overview of the bit flags which are maintained in the page table.

Index	Name	Description
0	Present	Indicates whether the PT entry is valid and the corresponding page or L2 table is present.
1	Writable	If this bit is '0', the page is marked as read only. Writing to this page generates a trap.
2	Protection Level	This bit is set for kernel pages. When access is performed without elevated access rights a trap is generated.
3	Global	Global pages and their virtual to physical mapping are valid for every process (ASID is ignored). These entries are never replaced by the TLB.
4	Cachable	Read and writes to this page bypass the cache.
5	Dirty	This bit is set by the TW when a write is performed to the page. The OS checks this bit when a page is swapped out to see if its contents are changed need to be written back.
6	Accessed	This bit is set by the TW when the page is accessed. The OS can use this bit as an indication which pages are not frequently used when it needs to swap out a page.
7	Large Page	When this bit is set in a L1 PT entry it indicates the entry holds the address mapping of a large page instead of a pointer to a L2 page table.

control register associated with the MMU that is global and not context-specific.

Flushing parameters (MMU_FHI, MMU_FLO & MMU_FID)
This set of registers are used to supply parameters for a fine grained flushing operation. Available flushing operations are: flush all, flush every entry with a specific virtual tag, flush a specific range of virtual tags, flushing all entries with a specific ASID and all combinations of these modes. The MMU_FID register is used to set the ASID which all entries associated with must be flushed. The MMU_FHI and MMU_FLO registers are used to specify the lower and upper boundaries of a range of virtual tags which will be flushed. When a single virtual tag needs to be flushed, the MMU_FLO register is used to supply this tag. The flushing mode and flushing trigger are part of the MMU_CR register. These registers are context-specific so that each active context has its own flushing interface. Recall however from Section 3.6 that, whenever a context issues a flush command, all TLBs, execute the flush ensuring coherency.

MMU Control Register (MMU_CR)
This is the main control register for the MMU. In this section the fields illustrated in Figure 4.14 will be clarified.

- **Mode**, This three bit field is used to specify the flushing mode. The possible values and corresponding modes are listed in Table 4.2.

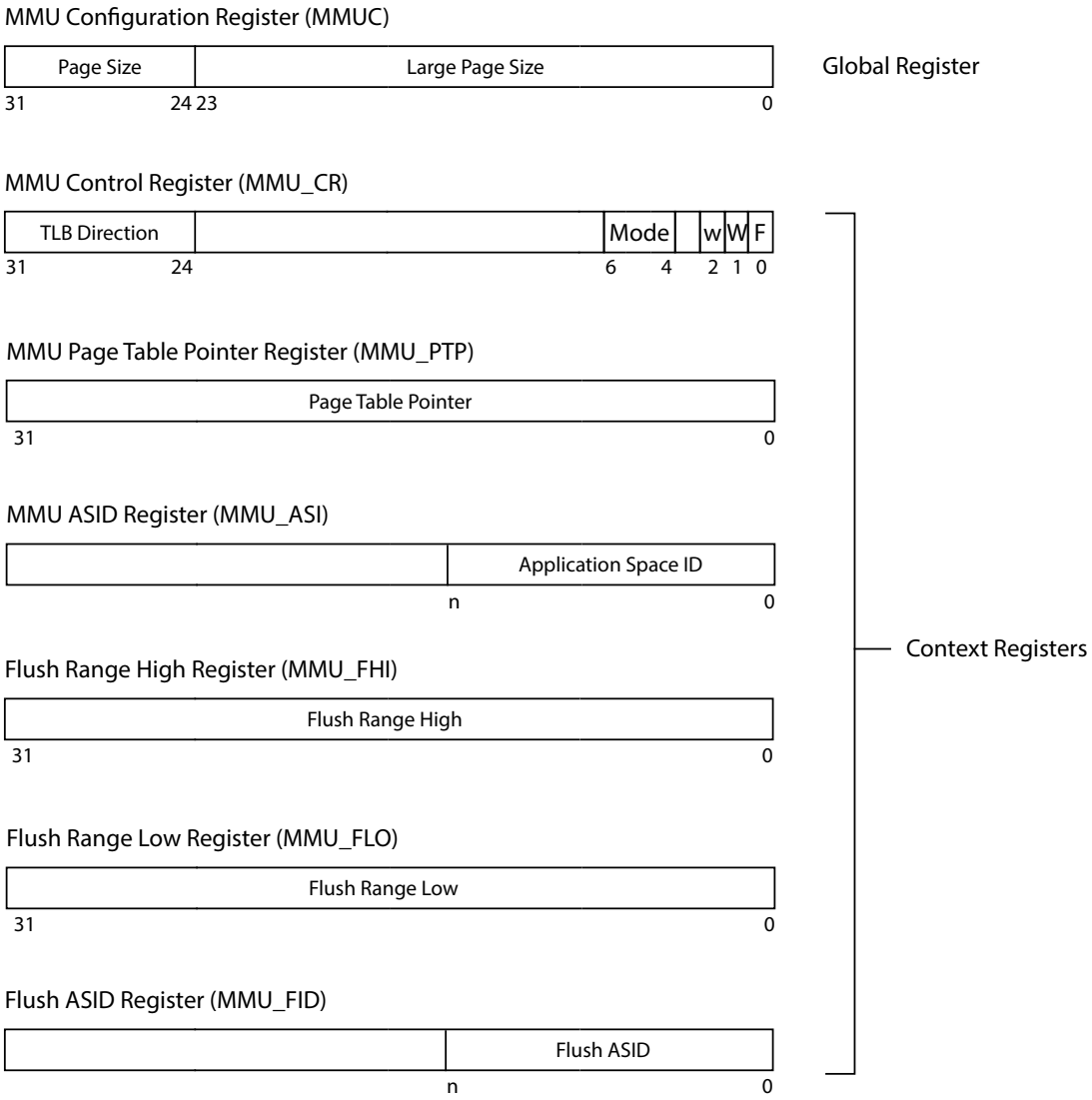


Figure 4.14: Overview of all the control registers added for the MMU and the layout of their fields. An explanation of the fields and their function is given in Section 4.9.

- **F**, Writing a '1' to this bit field issues a flush command. It is located the same byte as the flush mode field so the mode can be selected in the same write operation.
- **W**, Writing a '1' to this bit field enables the write-to-clean-page trap. This trap is useful when the OS implements a copy-on-write scheme.
- **w**, Writing a '1' to this field disables the write-to-clean-page trap.
- **TLB Direction**, This field can be used to direct TLB misses to a specific issue lane. This means that all updates are performed on the lane specified. This can be useful if it is already known that the process will switch to that lane in the

Table 4.2: This table lists all possible values for the flush mode field in the MMU_CR.

Value	Flush mode
"000"	Flush all
"001"	Flush all entries with a certain virtual tag
"010"	Flush a range of virtual tags
"100"	Flush all entries with a certain ASID
"101"	Flush all entries with a certain ASID and a certain virtual tag
"110"	Flsuh all entries with a certain ASID and a range of virtual tags

future. This will only work if the value is set to the index of a lane the process is currently running on. All other possible values for the field will result in the default behaviour where updates are evenly distributed over the coupled lanes.

Process specific parameters (MMU_PTP MMU_ASI)
Differentiation between virtual address spaces is done based on the ASID and PTP. The ASID is used to denote which context a TLB entry belongs to. The PTP is used by the TW to access the correct PT when a TLB miss occurs. These parameters must be updated to execute a process switch.

4.10 Conclusion

This chapter details the implementation of the MMU and its subcomponents in reconfigurable hardware. In chapter 3 the architecture for the memory hierarchy was selected. This architecture places the TLB between the core and the cache. The interface between these three components is an extension of the baseline interface between the core and the cache. The MMU can assert stall signals to halt the operation of the cache and core when it fails to translate an address tag in a single cycle. The MMUs operation is controlled through a memory mapped register interface.

The TLB is the most complex component of the MMU. This is for the most part caused by the CAM, which is is not well supported by the FPGAs hardware resources. The CAM is implemented in BRAM resources to allow for larger TLB depths. One implication of this choice is that the CAM can contain stale entries. Since BRAMs cannot be cleared globally in a quick way, another solution is designed to handle this problem. The implemented solution is a deferred clean up mechanism. This entails that faulty entries are removed from the CAM whenever they are encounters.

The ρ-VEX MMU supports both design-time configuration and run-time reconfiguration. There are multiple parameters to tailor the MMU to a specific application. These include the page size and the TLB depth. A mechanism is implemented to distribute TLB reads and writes over all TLBs in a group of coupled lanes. This allows lanes to use the address mappings present in the TLBs of coupled lanes as well as their own.

5 Functional Verification

The main reason to add an Memory Management Unit (MMU) to the ρ-VEX system is to be able to run an Operating System (OS) on the platform. It is impossible to run an OS which implements virtual addressing on any platform without an MMU and an MMU has little use if it there is no OS running to use it. Unfortunately designing an MMU for the ρ-VEX and porting Linux are two separate projects which are too large to be completed in a single master thesis project. There currently is a Linux version available for the ρ-VEX but because this was developed before the ρ-VEX MMU, a version a Linux was ported which is stripped of the memory management functionality. This work was also done by a master student and is presented in [36]. For this reason, at the completion of this project, it is not possible to test the MMU in synergy with an OS.

To still be able to verify that the system is capable to support an OS, it is necessary to determine the minimum hardware requirements a system needs for this purpose. Since there is already a working port of Linux in existence, this verification is only required for the parts that relate to virtual addressing. In Section 2.6, the minimal functional requirements of the MMU were listed. Based on these core functionalities, verification software was designed. This piece of software can be viewed as the essence of an OS, implementing only the most fundamental tasks such as virtual address spaces for each process, task switching, and memory protection. This software is described in Section 5.1. Evaluation of the verification software will take place in Section 5.2. Finally in Section 5.3, bugs which are still present in the system will be discussed.

5.1 Verification Software Design

The raison d'etre for an OS is to allow multiple processes to share the hardware of a single computing platform. In both uniprocessor and multiprocessor systems this is done by multiplexing the processors resources in time to different processes. Whenever processes cannot terminate within their respective timeslots, the processor preempts these processes and assigns the hardware resources to another task. Additionally, to protect processes running concurrently on the system from each other, the systems memory is also divided between the different tasks. Each process can request a part of the memory for it own use which is protected from access by other processes. Memory protection is enforced by a unique mapping from the processes virtual pages to physical frames which are solely accessible by the specific process. Because the translation of addresses is performed in hardware by the MMU this system can not be circumvented. The OS controls the division of memory resources because it is in charge of creating and maintaining virtual to physical memory mappings which are stored in each processes private Page Table (PT).

The above description of an OS is emulated by the verification software written to test the ρ-VEX MMU. In this software, multiple Powerstone benchmarks are run in their own virtual address space and are preemptively switched at set intervals. The program terminates when each benchmark is complete. These processes are identified by unique Address Space Identifiers (ASIDs), which ensure that they cannot use each others mappings present in the Translation Look-aside Buffers (TLBs). When the program starts, first some initialization code is run in kernel mode which performs tasks such as configuring the core, the MMU, the timer interrupt peripheral, and clearing each process' L1 page table. After the initialization is completed, the code switches to the first benchmark which is run in its own virtual address space. Because the software uses demand paging this immediately leads to an instruction page fault since there are no mappings in the TLB yet. When the MMU issues a pagefault, the softwares Interrupt Service Routine (ISR) is invoked which updates the processes private PT. In the verification software, the virtual to physical mappings are simply offsets[1] which are unique for each process. This results in each benchmark occupying subsequent sections of the physical memory. This virtual to physical mapping is illustrated in Figure 5.1.

Task switching
At a set interval in time during the program external interrupts are triggered by a timer peripheral. Each time the core receives an interrupt the programs ISR performs a task switch. This entails storing the context presently running on the core and restoring the context of the process that is about to run. The MMUs control registers for the Page Table Pointer (PTP) and ASID are also updated so that the MMU will translate incoming virtual addresses to the newly selected address space. In Figure 5.2, part of a modelsim wave viewer window is shown, which illustrates this process.

The task switching mechanism is included in the program for two reasons. Primarily this is a test to check if the MMU is able to separate the different address spaces from each other. More specifically to see if the TLBs are able to hold mappings for different processes at the same time and to differentiate between them correctly. The second reason is to increase the pressure on the TLBs. When multiple programs run concurrently, they have to share the available slots. This enables a better performance comparison between different MMU configurations which will be performed in Section 6.2.

[1]The mapping of virtual pages to physical frames is a high level task in the OS since it is related to the division of memory resources amongst processes. Because this is not relevant for evaluating the hardware, the most simple mapping solution is implemented.

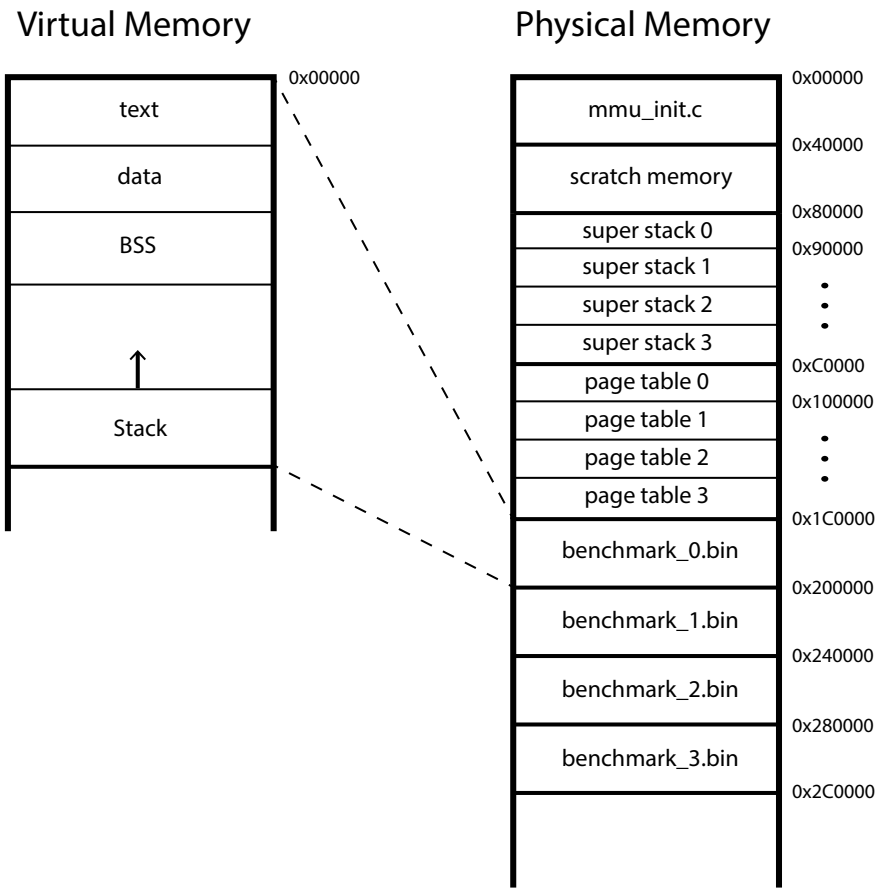


Figure 5.1: This diagram shows the physical memory layout of the verification software. Also shown how a virtual address space of one of the benchmarks is mapped to physical addresses.

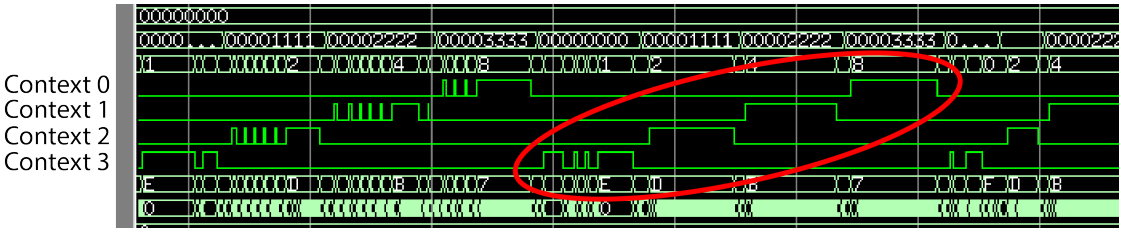


Figure 5.2: This is a waveform of the ρ-VEX running the verification software. Shown is the enable signal for the MMU for the four different hardware contexts. This image visualizes the preemptive task switching and pagefaulting in the software. Only one context is active at a time, occupying the entire core. The notches occur when a pagefault is serviced. These are handles in kernel mode in which the MMU is bypassed. Note that this is taken from an earlier version of the test software which relied on hardware context switches. The version used for this thesis uses software contexts running on a single hardware context. This was done to support eight processes instead of four.

5.2 Verification Software Evaluation

The minimal requirements of the MMU to support an OS, which implements virtual addressing was listed in Section 2.6. To increase ease of reading, these points will be listed again in this section.

1. Autonomous address translation of mappings stored in the TLB.
2. Automatic checking of access rights (read/write and protection level) on every page access.
3. A way to insert entries in the TLB; either a software interface or an Table Walker (TW) unit.
4. In case of a hardware managed TLB, a mechanism to stall the core upon a table walk and a clearly defined PT layout.
5. A mechanism to issue a trap when a page fault or access violation occus.
6. A software interface to flush TLB entries.
7. A mechanism to bypass the MMU for system initialization and kernel code that is executed in physical address space.

A comparison will be made between these points and the verification software to ascertain that the minimum functionality is covered by the implementation of the MMU.

1. The software test this extensively by running each benchmark in virtual mode. This feature is used at every memory access made by the user processes which in this case are the benchmarks. The correct termination of each benchmark is a good test if the mapping is actually done correctly.
2. This feature is not used in the verification software.
3. The software relies on the TW to load entries into the TLB the software is not able to do this since there is no interface for this as the MMU is implemented as hardware managed.
4. The stall mechanism is required for proper functioning of the system. If the OS and TW do not agree on the PT layout, the program would not be able to handle pagefaults since it cannot update the PT in a way such that the TW can find the correct mappings.[2]
5. This feature is used by the software to trigger the page fault handler which is used many times during the software's runs. Without this feature operational the software would not be able to handle pagefaults.

[2]Note that in the ρ-VEX the PT layout is well defined but not fixed. The system has parameters which determine the size of regular and large pages. These sizes also influence and layout and size of the two levels of the PT. Section 4.8 describes the exact layout of the tables and their entries.

6. This feature is used to flush all TLBs at the start of the program. This ensures that each consecutive run of the program has the same initial conditions. The fine grained flushing capabilities of the MMU are not used in this test.

7. When the software boots, the first part of the program which is used for initialization purposes and also the ISR code are both run in physical mode. This shows how the MMU can be bypassed on start-up and if desired inside trap handlers.

This leaves feature two, access right checking to be tested separately. This is done by adapting the software described in this chapter to test this function specifically. The rest of this section describes how this test is performed.

Access rights
Access right checking in the MMU is performed by the TLB on each translation. The access right bits which are stored alongside each address mapping (kernel-page and write-access) are compared to the access rights and operation performed by the active context. When a kernel page is accessed by a process which is running in user mode, a *kernel space violation* trap is generated. When a write is performed to a read-only page, a *write-access-violation* trap is generated.

To verify this mechanism functions correctly, the verification software is adapted to deliberately trigger these kinds of traps. This is done by modifying the page-fault handler to raise the access restrictions for the pages it allocates for one of the user processes. After such a page is allocated, the corresponding user program will try to access it without the proper rights. This leads to trap of one of the types described earlier. That this trap is actually thrown and is recognized by the processor can de gathered from a print message put into the handler for the trap.

5.3 Implementation Bugs

While the core functionality of the MMU is operational, there is still a bug in the implementation. This bug can appear while running larger programs in virtual mode and at this time, inhibit thorough testing of the implementation on the Field-Programmable Gate Array (FPGA). The issue sometimes manifests itself when the core performs an Return from Interrupt (RFI) instruction at the end of executing the trap handler. When this RFI instruction is issued, the core jumps to the point where the trap occurred, freezes and no more instructions are executed afterwards. While the occurrence of the issue is deterministic, it happens only in some cases. Small changes is the software can sometimes avoid this bug to surface. The specific conditions which leads to the appearance of this problem have not yet been identified.

Some time has already been spent in locating the cause of this issue. The insertion of performance counters that keep track of the amount of stalled cycled have excluded a deadlock caused by the MMU. It is possible that the issue is caused by a bug in the core's trap mechanism that has not been encountered before. Before the addition of the MMU, the trap mechanism has never been used extensively. A complicating factor is that this issue never occurs when simulating the system. This makes the debugging

process much more difficult and time-consuming. Unfortunately, there is not enough time in the course of this project to spent on locating and solving this issue.

Running the verification software described in Section 5.1 has been successful, although preemtive task switching had to be disabled. Successful attempts to run software in virtual mode on the FPGA relied on avoiding these bugs to appear by inserting No Operations (NOPs) at certain points in the ISR. Therefore, most of the measurements presented in Chapter 6 are obtained by simulations.

5.4 Conclusion

In this chapter the functionality of the added hardware is verified. This task is complicated by the fact that there is no OS available for testing. Custom software was designed to emulate the function of an OS. The software is designed to rely on hardware support on the points listed in Section 2.6. The rationale is that an MMU able to support the verification software, it would also be able to support a real OS. The last part of the chapter discussed bugs that are still unsolved.

6 Measurements

in Chapter 5, a functional evaluation of the implemented hardware was performed. This chapter will try to evaluate the performance of the new ρ-VEX system with virtual memory support. The performance cost of virtual memory consists of two components. When a pagefault occurs, a software layer usually, located in the Operating System (OS), needs to bring a missing frame into memory and update the Page Table (PT). This software layer falls outside the scope of this project and will not be discussed here. The other mechanism, that increases programs latency, are Translation Look-aside Buffer (TLB) misses. These can be solved by a table walk if the missing memory mapping is present in the PT. The number of entries the TLB can hold and the memory access pattern of the application determine how often misses occur. When multiple programs are run concurrently on a processor, they need to share the available TLB space. The ρ-VEX Memory Management Unit (MMU) includes Address Space Identifiers (ASIDs) in the TLB entries to distinguish between address mapping of different processes.

In Section 6.1, the new system is compared to the baseline version in terms of area increase and operating frequency. Section 6.2 quantifies the performance overhead of supporting virtual memory. The effect of different settings of the static configuration parameters on this overhead is measured and compared. Section 6.3 explores the performance behaviour of the ρ-VEX virtual memory hardware in scenarios where dynamic reconfiguration is performed.

6.1 Area utilization and Operating Frequency

Adding virtual memory hardware support to the ρ-VEX obviously incurs a price. The increase in Field-Programmable Gate Array (FPGA) resource utilization is of course unavoidable. A comparison of the old and new ρ-VEX systems is given in Table 6.1. The results are for a dynamic 8-issue ρ-VEX with a 16 KiB instruction cache and a 4 KiB data cache. The MMU is configured to its base configuration, 32 entry TLBs and a page size of 4 KiB. The largest increase is in utilized Block RAM (BRAM) resources. These are used for the Content Addressable Memory (CAM) which is present in every TLB in the system. One of the reasons for implementing the CAMs in BRAMs was because these are readily available of the target FPGA. There is also a significant increase in Lookup Table (LUT) utilization. This is probably due to the many features that have been added to the MMU in the course of this project.

Unfortunately the operating frequency of the ρ-VEX system has also suffered from the addition of the MMU. When synthesized for a Virtex 7 FPGA, the baseline implementation reaches around 75 MHz. The new ρ-VEX system only reaches 47 MHz. When a lot of effort is put in optimization by the synthesis tool a frequency of 57 MHz can be reached. There might be some room for improvement of these figures if some more time

is spent studying the critical path and optimizing the design.

The performance decrease is probably also partly inherent to the design. The choice of implementing the TLBs CAM in BRAMs was explained in Section 4.4.1. This has a drawback which was not taken into account when the system was designed. Using BRAM resources limits the routing freedom of the synthesis tool. Inside the FPGA the BRAM resources are located in columns at certain physical locations. Using them either fixes the location of the TLBs or requires routing the signals from the TLBs location to the locations of the BRAM blocks. In either case, this will lead to larger routing delays for the data path. Implementing the CAMs in slice resources avoids this penalty.

Another architectural decision was to implement a Virtually Indexed Physically Tagged (VIPT) cache. Refer to Section 3.5 for this discussion. This type of cache theoretically allows a simple memory hierarchy without the added latency of a Physically Indexed Physically Tagged (PIPT) cache. However, with the current increase in operating frequency, the added latency seems preferably. Both these alternatives will be further discussed in Section 7.4 as future work.

6.2 Static Configuration Evaluation

The ρ-VEX MMU is designed to comply with the ρ-VEX philosophy of design-time configurability and run-time reconfigurability. This section will explore how different configuration parameters for the MMU influence the performance of the verification software described in this chapter. These results can loosely be generalized to any multitasking system that employs virtualization. Besides the enable flag, the MMU can be configured in four dimensions; TLB depth, page size, large page size, and ASID bit-width.

The comparisons performed in this section are based on varying the first two. Using and varying the size of large pages has a similar effect as increasing the regular page size, only more fine grained. Therefore, this feature will not be explored. Sizing the ASID can have advantages in a system where a large number of tasks are run concurrently. Increasing this parameter can help avoid reassigning ASIDs between processes. Because this parameter does not affect the performance of the hardware, it will also not be used in this sections evaluation.

In the verification software, multiple programs run concurrency on the ρ-VEX, thereby contending for TLB space. If the TLB is not large enough, this will lead to TLB replacements. When this happens the number of TLB misses the program encounters during its run increases. Inside the MMU, there are performance counters which

Table 6.1: This table compares the resource utilization of the ρ-VEX system that incorporates the memory management hardware to the baseline implementation.

recourse	**old design**	**new design**	**increase**
slices	24051	25604	6.4%
LUTs	61891	70880	14.5%
flip-flops	30488	33877	11.1%
BRAMs	225	273	21.3%

count how many times these events occur.

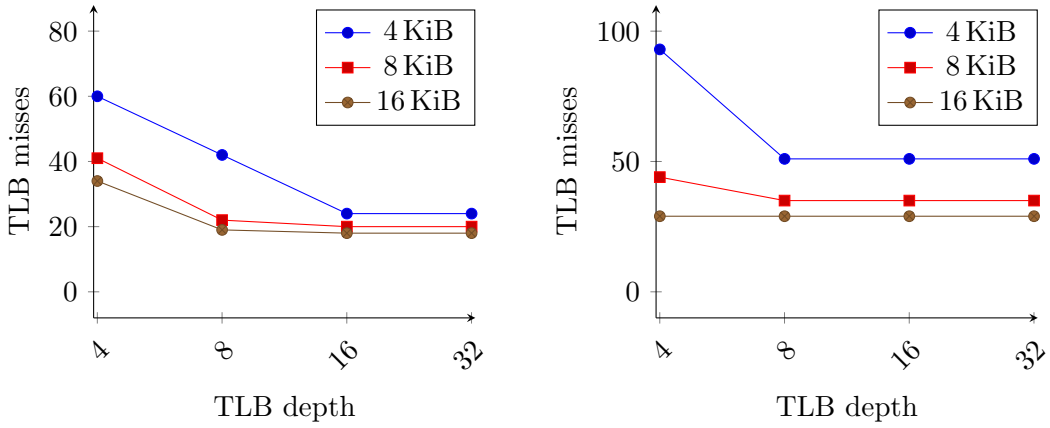
Each TLB miss increases the programs run-time because the Table Walker (TW) needs to perform a table walk. This can obviously be minimized by increasing the number of entries each TLB can hold. Another way to decrease the number of TLB misses is to increase the page size. When the page sizes is increased programs generally require less pages to be allocated for them. This also relieves some of the pressure on the TLBs because each entry spans a larger portion of the memory. The drawback of larger pages sizes is that the memory utilization of programs can increase due to internal fragmentation. In Table 6.2 and 6.3 the results are shown of running the evaluation software on a ρ-VEX processor configured as an 8-issue and a 2-issue respectively.

The graphs in Figure 6.1 visualize the essence of these tables more clearly. The number of TLB misses was plotted because they this relates directly to the amount of stall cycles caused by the MMU. As expected increasing the TLB depth and page size both have a positive effect on the hardware latency overhead of memory virtualization. This is beneficial up to the point where the TLBs are simply large enough to hold all the mappings of each process present in the system. The attentively reader might note that in terms of resource utilization all TLBs which have up to 32 entries use the same amount of BRAMs. Therefore there is no reason to use TLBs which have four, eight, or sixteen entries. These smaller TLB sizes are chosen for this test simply because these eight benchmarks do not use a lot of memory. In a real system where an OS runs hundreds of processes concurrently, TLB sizes of 32, 64 or larger are more appropriate.

Notable in the results is also the difference between the software run on a ρ-VEX configured as an 8-issue or 2-issue system. In the 8-issue system the data TLB of all lanes are available which leads to the amount of space being four times as large compared to the 2-issue system. The effect of this can be clearly seen when comparing Figures 6.1(b) and 6.1(d).

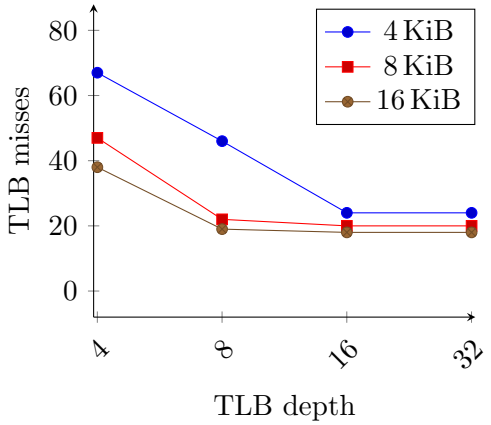
The tables and graph clearly show that increasing the page size can have a positive effect on the latency of programs due to less TLB misses. The trade-off however is an increased memory footprint of the program due to internal fragmentation of the memory. Figure 6.2 shows how this scales for the verification software. Programs usually occupy a contiguous area of virtual memory, in that case only the highest page suffers from internal fragmentation. This means that this problem is more pronounced when running many small programs then when running fewer large ones. Note that because of the choice for a VIPT cache, the page size also limits the maximal size of the data cache. Therefore the choice for a certain page size is a trade-off between less TLB misses and a larger maximal data cache size versus higher memory use.

One last point to notice is that in general increase in cycles caused by the virtualization hardware is generally low. Only in cases where programs access a large number of pages in proportion to the number of TLB entries, which leading to thrashing, is this a serious bottleneck. The larger cost of virtual addressing is in handling pagefaults which is done in software and can cost hundreds to thousands of cycles. This is very implementation dependant and falls outside the scope of this project.

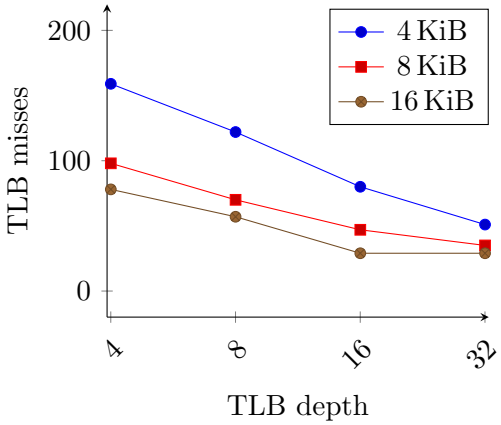


(a) instruction misses on an 8-issue core

(b) data misses on an 8-issue core



(c) instruction misses on a 2-issue core



(d) data misses on a 2-issue core

Figure 6.1: These graphs show the number of TLB misses encountered when running the verification software on different static configurations of the ρ-VEX.

6.3 Dynamic Reconfiguration Evaluation

In Section 4.7, a mechanism was presented that allows the core to broadcast addresses to all data TLBs of coupled issue lanes. This effectively scales TLB size of a hardware context linearly with the number of coupled lanes. Section 4.7.2 described another feature that allows the core to direct TLB updates to a specific TLB in a coupled configuration. This second mechanism allows the the core to anticipate to reconfigurations in the near future. These two features can be used to minimize the amount of TLB misses which occur in the course of running a (set of) application(s). TLB misses will therefore be used as the main performance metric in this chapter. The benchmarks and scenarios used in this chapter are all purely synthetic. They are designed to generate a certain pattern of page accesses. The results of the measurements will be evaluated and conclusions are drawn that can be extended to real scenarios.

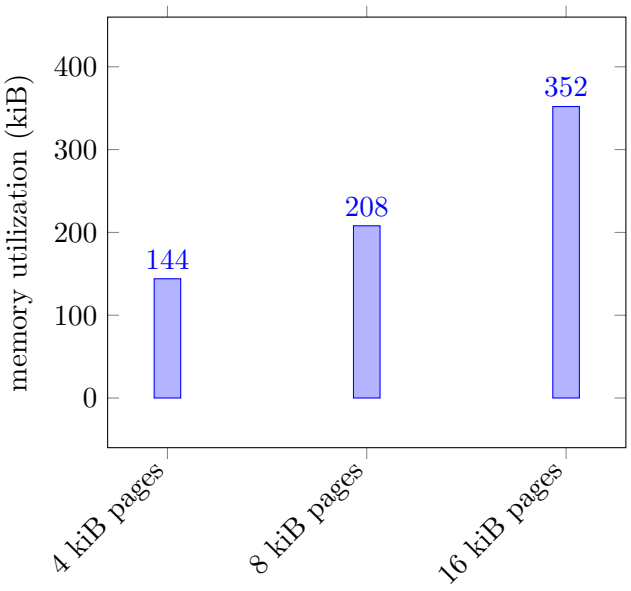


Figure 6.2: This graph shows how the verification software's memory utilization rises with increased page size due to internal fragmentation of the code. Note that the increase is extreme in this case because the included benchmarks only use one or two pages. for larger programs the increase will be less significant.

Table 6.2: Performance results of the MMU for different page sizes and TLB depths. These numbers were obtained by running eight Powerstone benchmarks in the software described in Section 5.1 in 8-issue mode with a task switching period of 10000 cycles.

	TLB Depth:	**4**	**8**	**16**	**32**
Page Size: 4 KiB	instruction TLB misses:	60	42	24	24
instruction page faults: 12	data TLB misses:	89	51	51	51
data page faults: 24	user mode cycles:	200000	199450	198512	198512
memory utilization: 144 KiB	MMU stall cycles:	2839	1804	1534	1534
	MMU stall %:	1.42%	0.90%	0.78%	0.78%
	task switches:	26	26	26	26
Page Size: 8 KiB	instruction TLB misses:	41	22	20	20
instruction page faults: 10	data TLB misses:	44	35	35	35
data page faults: 16	user mode cycles:	197349	196786	196821	196821
memory utilization: 208 KiB	MMU stall cycles:	1641	1176	1146	1146
	MMU stall %:	0.83%	0.60%	0.58%	0.58%
	task switches:	25	24	24	24
Page Size: 16 KiB	instruction TLB misses:	34	19	18	18
instruction page faults: 9	data TLB misses:	29	29	29	29
data page faults: 13	user mode cycles:	197561	197461	197480	197480
memory utilization: 352 KiB	MMU stall cycles:	1230	1005	990	990
	MMU stall %:	0.62%	0.51%	0.50%	0.50%
	task switches:	24	24	24	24

Table 6.3: The same test results as in Table 6.2 but run on a 2-issue ρ-VEX

	TLB Depth:	**4**	**8**	**16**	**32**
Page Size: 4 KiB	instruction TLB misses:	67	46	24	24
instruction page faults: 12	data TLB misses:	159	122	80	51
data page faults: 24	user mode cycles:	271395	270716	269875	269550
memory utilization: 144 KiB	MMU stall cycles:	3790	2937	1876	1365
	MMU stall %:	1.40%	1.08%	0.70%	0.51%
	task switches:	36	36	36	36
Page Size: 8 KiB	instruction TLB misses:	47	22	20	20
instruction page faults: 10	data TLB misses:	98	70	47	35
data page faults: 16	user mode cycles:	270450	270007	269703	269593
memory utilization: 208 KiB	MMU stall cycles:	2473	1701	1231	1033
	MMU stall %:	0.91%	0.63%	0.46%	0.38%
	task switches:	34	34	34	34
Page Size: 16 KiB	instruction TLB misses:	38	19	18	18
instruction page faults: 9	data TLB misses:	78	57	29	29
data page faults: 13	user mode cycles:	271727	271172	270839	270839
memory utilization: 352 KiB	MMU stall cycles:	1997	1392	866	866
	MMU stall %:	0.73%	0.51%	0.32%	0.32%
	task switches:	34	33	33	33

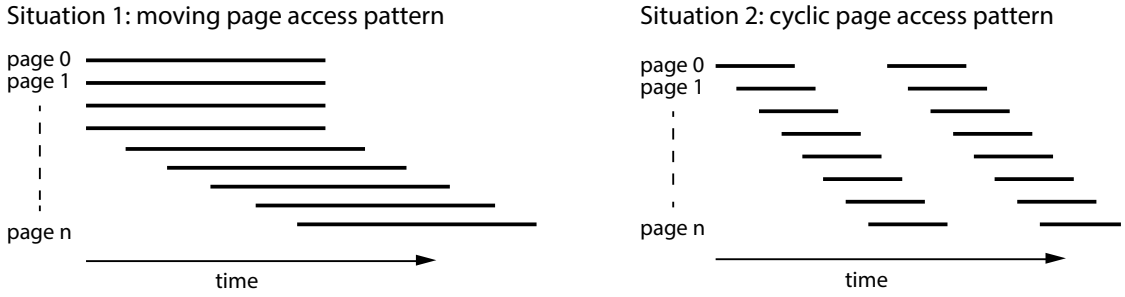


Figure 6.3: These diagrams give a schematic representation of the page access patterns of the two benchmarks designed to measure the the performance effects of dynamic reconfiguration on the MMU.

6.3.1 Synthetic Benchmarks

Besides the hardware configuration, another factor that influences the performance of the virtual memory hardware is the page access pattern of the application. To be able to take this into consideration, two synthetic benchmarks are designed. The page access pattern of these two programs are depicted schematically in Figure 6.3. The first benchmark does an element wise summation of a number of vectors. The program has a moving window of the vectors it operates on. Furthermore, each vector is located on another memory page. This forces the program to access an unusual large number of pages each triggering a pagefault. The second benchmark multiplies a matrix with a transposed version of itself. In this program, each row of the matrix is also located on another page. This program displays a cyclic page access pattern.

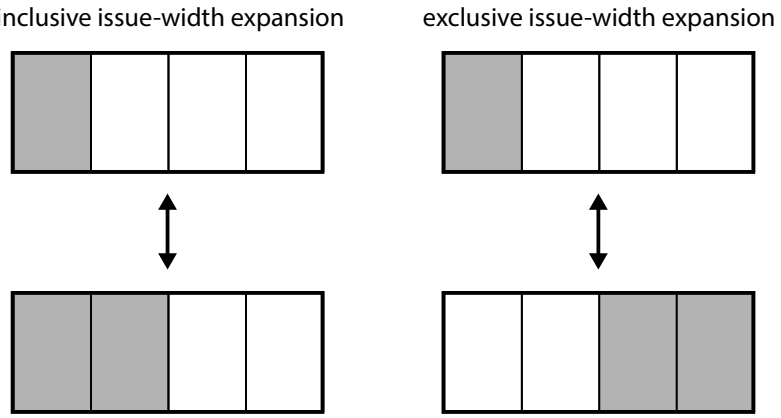


Figure 6.4: The blocks in this diagram depict an 8-issue ρ-VEX core which can be divided into maximum of four 2-issue cores. The diagram depicts the two scenarios used for the measurements presented in Section 6.3.2.

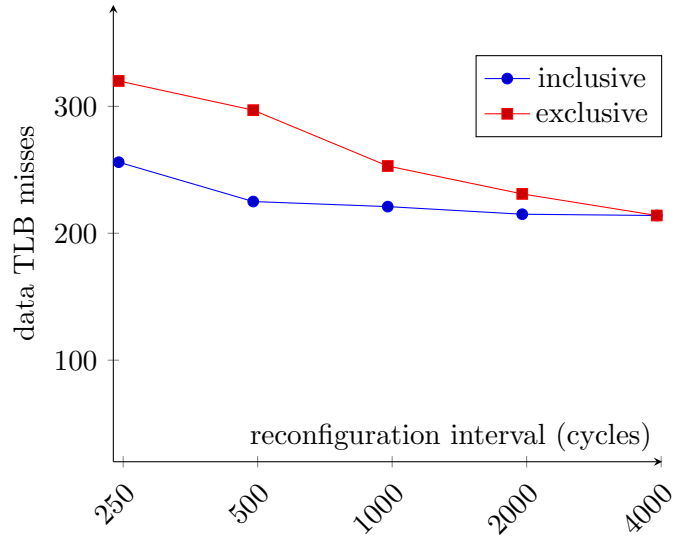
6.3.2 Lane Expansion

In the ρ-VEX system, a process can be migrated to a wider core when it requires more hardware resources. It can be advantageous if the cache blocks and TLBs of the designated lanes are already initialized, avoiding a so called *cold start*. This is the case if the set of lanes the process migrates to includes the lane(s) it has been running on.

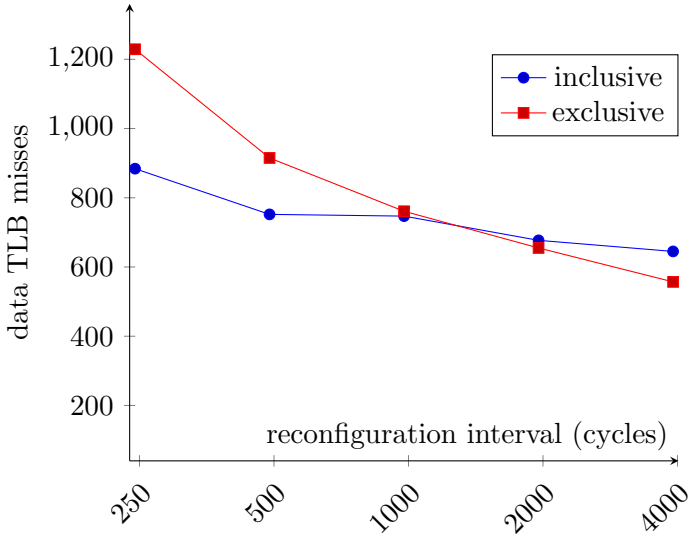
To get an idea of the benefits of this inclusion, a comparison is made between two different scenarios. In both scenarios, a benchmark program is started on a 2-issue ρ-VEX core. During the course of the program the context is repeatedly migrated to a 4-issue ρ-VEX core and back. In one situation the expansion includes the initial lane and in the second situation it does not. In Figure 6.6 these two situations are illustrated graphically. The switching is triggered by an external interrupt, whose interval is counted in user mode cycles. This means that all time spend servicing the page faults and handling the external interrupt does not influence the results.

This experiment is run for both benchmarks outlined in Section 6.3.1. This allows taking the memory access pattern of the application into account as well. Figure 6.5(a) shows the results for both experiments. In case of moving page access, inclusive lane switching can clearly have a positive effect. This effect holds as long as there are active pages in the TLB when switching back to an older configuration. If the switch period is longer than the page window, the positive effect is lost.

Inclusive lane switching can even have a negative effect. This can be seen in the graph in Figure 6.5(b). The cyclic benchmark differs from the other one in that it operates on more pages than the TLB can hold. This leads to thrashing, which results in the higher number of TLB misses than for the other benchmark. The reason for the performance penalty of inclusive switching comes from the fact that the application has access to only two TLBs. In the exclusive expansion scenario, there are three TLBs used during the course of the program. This effect can overpower the positive effect of lane inclusion.



(a) Moving page access pattern.



(b) Cyclic page access pattern.

Figure 6.5: Results of the lane expansion comparison for the two different benchmarks. The experiments are performed on a ρ-VEX configured with a page size of 4 KiB and 8 entry deep TLBs.

6.3.3 Lane Reduction

In this section, an experiment is described that measures the benefits of the data TLB update direction mechanism, presented in Section 4.7.2. The scenario for this experiment is a migration of a hardware context that runs on a 8-issue ρ-VEX core to a 2-issue core. The context alternates between these configurations every 8000 cycles. A varying number

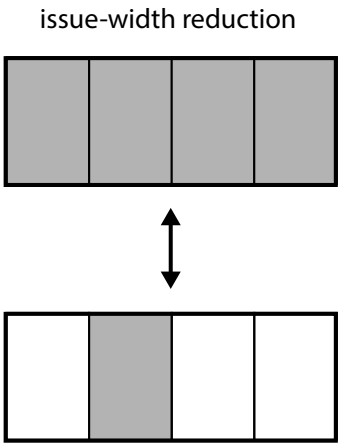


Figure 6.6: This diagram illustrates the scenario of the lane reduction experiments.

of cycles before the lane reduction is scheduled, the context will direct its data TLB updates to a specific TLB. This can avoid misses since all recently added translations are already present in the TLB after the migration. The benchmark with moving page access pattern is used in this experiment. In Figure 6.7 the number of data TLB misses is plotted for different direction periods. The plot shows the number of additional misses due to reconfiguration, normalized to the situation where no TLB direction is performed. The graph shows that the direction strategy can offer a modest benefit, but can also increase the number of misses. This negative effect stems from the fact that during the direction period, only one data TLB is used. The reduction in TLB space makes it more likely that updates replace an entry that is still in use. An optimal direction period is related to the average time that pages are used in the application. Ideally, all updates during the direction period are referenced after the reconfiguration. If this period is too large, the reduction in TLB space overpowers the effect of avoiding cold starts.

6.4 Conclusion

This chapter first makes a comparison between the baseline platform and the extended platform regarding area increase and operating frequency. The new design primarily increases the number of BRAMs and LUT resource by 21.3% and 14.5% respectively. A much more significant cost is the decrease in operating frequency from 75 MHZ down to 47 MHz. It might be possible to alleviate this penalty if some more effort is spent in locating and removing unnecessary critical paths. However, it might also be inherent to the architectural choices made during the course of this project. The decision to implement the CAMs in BRAMs and to implement a VIPT cache might have incurred this cost.

In the following section, the verification software described in Chapter 5 was used to evaluate the performance of the MMU in several static configurations. The effects of varying the TLB size and page sizes on the number of TLB misses was measured. It was shown that increasing these parameters can significantly reduce the number of

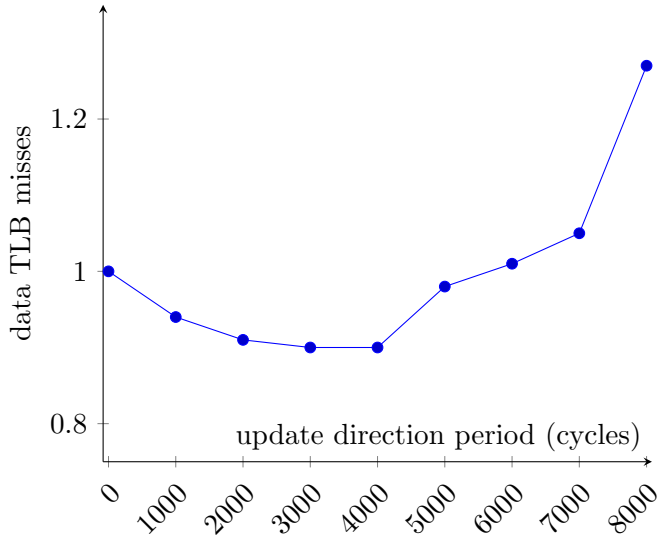


Figure 6.7: Results of the lane reduction experiments. The experiments are performed on a ρ-VEX configured with a page size of 4 KiB and a migration period of 8000 (user mode) cycles, running the benchmark that exhibits the moving page access pattern.

TLB misses. At the same time these measurements have shown the positive effect of the data TLB coalescing mechanism described in Section 4.7.1. Due to this mechanism, the number of data misses is significantly lower on an 8-issue compared to an 2-issue.

Subsequently, two scenarios were explored to measure the performance of the address translation hardware in relation to dynamic reconfiguration. In the first experiment, the benefit of avoiding a cold start when expanding to a wider core was measured. It was shown that the extent of this advantage is dependant on the page access pattern of the application. The second experiment measured the possible benefits of the TLB update direction mechanism described in Section 4.7.2. The results showed that a modest advantage can be obtained if the direction period is chosen carefully.

7 Conclusion

In this chapter the results of the project are discussed. In Section 1.2 the goals of this project were defined. This chapter will evaluate the final result with respect to these goals.

Section 7.1 summarizes the chapters of this thesis. In Section 7.2 the main contributions of this project will be listed. Finally in Section 7.4 recommendations for future work will be discussed.

7.1 Summary

In Chapter 2, first some background information about the implementation platform for the project was discussed. The first section of the chapter explained how Field-Programmable Gate Arrays (FPGAs) work and what their benefits are. Subsequently, it was explained how processors, referred to as softcore, can be implemented in this technology. In the following section, the general concept of Very Long Instruction Word (VLIW) processors and their advantages were explained.

This established the basis that allowed for the introduction of the ρ-VEX processor. The ρ-VEX is a VLIW softcore processor implemented in the reconfigurable fabric of an FPGA. Being a VLIW processor, the ρ-VEX issues instruction in bundles of two or more. This increases the Instruction Level Parallelism (ILP) and thereby the performance of application with sufficient levels of parallelism.

The ρ-VEX can be configured in a number of ways at design-time. This enables tailoring the ρ-VEX system to an specific application, optimizing performance and minimizing area. Besides these static configuration parameters, the ρ-VEX is also dynamically reconfigurable during run-time. This allows the ρ-VEX processor to switch between a single wide VLIW processor or multiple smaller cores. This feature enables running a single process with with high ILP or multiple processes with less parallelism, thereby increasing Thread Level Parallelism (TLP). Being able to switch between these configurations has implications for the memory system. The ρ-VEX system already features a cache that supports the core in each possible configuration. One of the challenges of this thesis project is to design and implement an Memory Management Unit (MMU) with similar properties.

The second part of Chapter 2 explained the key concepts of virtual memory using paging. In a system that supports paging, each process has its own virtual address space. The mapping of this address space to physical memory is controlled by the Operating System (OS) and is transparent to user programs. The translation of addresses is done dynamically by dedicated hardware called the MMU. The MMU contains multiple caches called Translation Look-aside Buffers (TLBs) which are used to cache the virtual to physical mappings. The address mappings created by the OS are stored in main memory

in a process specific data structure called a Page Table (PT). When an address is referenced that is not contained in the TLB, another hardware component called the Table Walker (TW) searches the PT for the missing translation. When the translation is not present in the PT, the TW generates an interrupt to call for OS intervention. The OS then invokes a routine that bring the missing page into memory and updates the PT.

In the chapters last section, a listing was made of the minimum set of functions that virtual address hardware should posses to support an OS. In Chapter 5, this list is revisited and used to evaluate the final implementation.

In Chapter 3, the high level design choices regarding the ρ-VEX MMU were explained and substantiated. First an overview was given of the different memory hierarchies possible in a system with both virtual address support and caches. After weighing the advantages and drawbacks of each of these architectures, the decision for a Virtually Indexed Physically Tagged (VIPT) type cache was explained. This type of cache avoid most of the complexities associated with a Virtually Indexed Virtually Tagged (VIVT) cache, especially in a multicore system like the ρ-VEX. Additionally, because address translation is not required before indexing the cache, the VIPT cache does not increase the length of the pipeline like a Physically Indexed Physically Tagged (PIPT) cache does. The main drawback of a VIPT cache is that its size is restricted by the page size. This can be extended by increasing the set associativity of the cache. Other ways to circumvent this restriction are increasing the page size and page coloring. Subsequently, the implications of the ρ-VEX dynamic reconfigurable nature for the MMU were discussed. The challenge lies in using the instantiated resources efficiently in all configurations. In the chapters last section, a few additional optimizations were proposed. These are relatively easy to implement and do not incur a high price complexity wise. These optimizations are possible in the ρ-VEX because the TLBs are tightly coupled to each other.

Chapter 4 details the implementation of the MMU and its subcomponents in reconfigurable hardware. The selected architecture places the TLB between the core and the cache. The interface between these three components is an extension of the baseline interface between the core and the cache. The MMU can assert stall signals to halt the operation of the cache and core when it fails to translate an address tag in a single cycle. The MMUs operation is controlled through a memory mapped register interface.

The TLB is the most complex component of the MMU. This is for the most part caused by the Content Addressable Memory (CAM), which is is not well supported by the FPGAs hardware resources. The CAM is implemented in Block RAM (BRAM) resources to allow for larger TLB depths. One implication of this choice is that the CAM can contain stale entries. Since BRAMs cannot be cleared globally in a quick way, another solution is designed to handle this problem. The implemented solution is a deferred clean up mechanism. This entails that faulty entries are removed from the CAM whenever they are encounters.

The ρ-VEX MMU supports both design-time configuration and run-time reconfiguration. There are multiple parameters to tailor the MMU to a specific application. These include the page size and the TLB depth. A mechanism is implemented to distribute TLB reads and writes over all TLBs in a group of coupled lanes. This allows lanes to use the address mappings present in the TLBs of coupled lanes as well as their own.

In Chapter 5, the functionality of the design is verified. This task is complicated

by the fact that there is no OS available for testing. Custom software was designed to emulate the function of an OS. The software is designed to rely on hardware support on the points listed in Section 2.6. The rationale is that an MMU able to support the verification software, it would also be able to support a real OS. this chapter also discussed several bugs that are still unsolved.

Chapter 6 first makes a comparison between the baseline platform and the extended platform regarding area increase and operating frequency. The new design primarily increases the number of BRAMs and Lookup Table (LUT) resource by 21.3% and 14.5% respectively. A much more significant cost is the decrease in operating frequency from 75 MHz down to 47 MHz. It might be possible to increase this if some more effort is spent in locating and removing unnecessary critical paths. However, it might also be inherent to the architectural choices made during the course of this project. The decision to implement the CAMs in BRAMs and to implement a VIPT cache might have incurred this cost.

Subsequently, the verification software described in Chapter 5 was used to evaluate the performance of the MMU in several static configurations. The effects of varying the TLB size and page sizes on the number of TLB misses was measured. It was shown that increasing these parameters can significantly reduce the number of TLB misses. At the same time these measurements have shown the positive effect of the data TLB coalescing mechanism described in Section 4.7.1.

In the last section of this chapter, two scenarios were explored to measure the performance of the address translation hardware in relation to dynamic reconfiguration. In the first experiment, the benefit of avoiding a cold start when expanding to a wider core was measured. It was shown that the extent of this advantage depends on the page access patter of the application. The second experiment measured the possible benefits of the TLB update direction mechanism described in Section 4.7.2. The results showed that a modest advantage can be obtained if the transition period is chosen carefully.

7.2 Main Contributions

The problem statement of this thesis project was:

How to implement hardware support for virtual memory on the ρ-VEX platform?

To answer this question, the following three goals were established:

1. Designing and implementing memory translation hardware for the ρ-VEX platform.
2. Proving that the platform is able to support an OS which implements virtual memory.
3. Measuring how the implementation performs in different static configurations and dynamic reconfiguration scenarios

To reach the first goal, virtual memory hardware has been designed that is tailored to the properties and requirements of the ρ-VEX platform. This hardware is encapsulated

in an MMU component and is integrated with the rest of the system. It is designed to require minimal modifications of the components that are already in place. The address translation step is performed between the core and the cache, in a way that allows it to be performed in parallel to the cache lookup. The MMU fully supports the dynamic reconfigurable nature of the platform in an efficient way. Furthermore, the MMU extends the statically configurable nature of the platform with several parameters. Besides these necessities, the MMU implementation also features some unique optimizations related to dynamic reconfiguration. A mechanism is implemented which allows the data TLBs of all coupled lanes to be referenced when one of the lanes performs a data access. This effectively scales the data TLB depth linearly with a cores issue-width. Additionally, it is also possible to direct TLB updates to a specific issue-lane. This allows software to anticipate on future reconfigurations. Furthermore, the fact that TLBs are tightly coupled allows for several optimizations. All TLBs share a single Table Walker (TW), multiple instruction TLBs can be updated simultaneously, and TLB coherency is automated and transparent to the programmer.

To meet the second goal, software has been designed that emulates the core functionality of an OS. This software runs multiple benchmarks in their own virtual address space, thereby relying on the address translation hardware in the same way a real operating system would. The fact that the implemented hardware is able to run this emulation proves that it could also support a real OS. However, there is still a bug in the system that prevents successful execution of virtualized programs in some situations. The time frame of this project does not allow more effort to be spent in solving this last issue. This effectively means that the second goal has not been completely fulfilled.

To reach the third goal, first a static evaluation of the system was performed. This evaluation consisted of running the same software on the platform with different configurations of the MMU. Comparing the results has shown that varying the configuration parameters can reduce the latency overhead of the virtual memory hardware. At the same time these experiments have shown the data TLB coalescing optimization can significantly reduce data TLB misses for programs run on wider issue cores.

For the second part of the third goal, two dynamic reconfiguration scenarios have been investigated. These measured the performance benefits of the two MMU optimizations relating to dynamic reconfiguration, TLB coalescing and update direction. It was shown that both can decrease the number of data TLB misses but they can also have a negative effect in some situations.

The rest of this section lists the features of the implemented hardware, divided into three categories. The first category holds the bare essential features of an MMU. These correspond to the ones established in Section 2.6. The second category holds features which increase the control the OS has over the hardware and allow for many optimizations. The final category hold features that are related to the unique properties of the ρ-VEX system.

Minimal functional requirements of the MMU:

- Autonomous address translation of mappings stored in the TLB.
- Automatic checking of access rights (read/write and protection level) on every page access.
- A TW unit to update the TLB when a miss occurs.
- A stall mechanism to halt the core and cache when a table walk is performed.
- A mechanism to issue a trap when a page fault or access violation occurs.
- A software interface to flush TLB entries.
- A mechanism to bypass the MMU for system initialization and kernel code that is executed in physical address space.

Additional features implemented in the ρ-VEX MMU:

- TLB entries are tagged with Address Space Identifiers (ASIDs). This avoids flushing TLBs on context switches.
- Support for large and global pages.
- Fine grained flushing control.
- Pages can be marked as non-cachable.
- PT bits for *accessed* and *dirty*, which are automatically maintained by the MMU.
- Transparent TLB coherence maintenance.
- Maskable write-to-clean trap. This allows implementing copy-on-write schemes.

ρ-VEX-specific features of the MMU:

- A large number of parameters to statically configure the MMU.
  - An MMU enable switch.
  - TLB depth.
  - page size.
  - large page size.
  - ASID bit-width.
- Support for run-time reconfiguration.
- Data TLB coalescing when merging cores.
- TLB update direction specifiable by software.

7.3 Additional Work

In the course of this project some time has been spent on additional work related to the ρ-VEX processor. This side project eventually resulted in publication of a paper at the Reconfig 2015 conference. This paper presents an idea which allows more efficient use of the register file. The ρ-VEX's register file is implemented using BRAM resources. These allow to store up to 18 KiB of data, while the register file only uses 256 32-bit entries. The paper discusses a method that allows some of the remaining space to be used to support multiple hardware contexts in the processor. This allows hardware task switching for a number of processes. These are much faster than software context switches, which involve spilling registers to the stack. Writing this paper was done in collaboration with multiple other students. This paper can be found in Appendix A.

7.4 Future Work

The addition of virtual memory hardware is primarily a step towards running a full-fledged version of Linux on the ρ-VEX platform. On a shorter term, there are still some bugs in the hardware that need to be sorted out. Furthermore, there are also some design choices which were taken during the course of this project which might have to be revised. These relate to the drop in operating frequency that occurred because of the addition of the new hardware. Some time has already been spent on eliminating some unnecessary long paths which has resulted in some increase in operating frequency. It might be possible to get the current implementation to a satisfying performance level if more effort spent on this. Finally there are also some recommendations for parts of the ρ-VEX, which are indirectly related to the MMU.

Future work relating to the MMU:

- Locating and solving the bug that sometimes surfaces when running software in virtual address mode on the FPGA. In Section 5.3 the circumstances and possible causes of this bug were discussed. Some effort has already been spent in solving the issue. Unfortunately the time frame of this project does not allow to complete this effort.
- Examination of the timing reports to remove any remaining unnecessary long paths.
- Implementing the TLBs CAM in slice resources instead of BRAMs to possible decrease routing delays.
- If the above options fail, implementing a PIPT cache. This means moving the TLB upstream in the memory hierarchy, thereby increasing the length of the pipeline.

Future work relating to other parts of the ρ-VEX:

- A Linux port which implements virtual memory. The availability of an OS would allow to implement a software layer which can control the dynamic reconfiguration

of the ρ-VEX to adapt to different tasks. This software layer could base its decision on compiler directives combined with performance counters in the hardware. These possibilities have already been proposed in [36]. The ρ-VEX platform now features a cache and MMU which both support dynamic reconfiguration and offer opportunities for additional optimization. These additions offer more variables for the control layer to take into account. It would be interesting to measure the effects the cache and virtual memory hardware on real applications where dynamic reconfiguration is controlled by the OS.

- The TLB update direction mechanism described in Section 4.7.2 and evaluated in Section 6.3.3 has not shown spectacular results. There is already work in progress within the Computer Engineering (CE) group which focusses on a similar system inside the cache. These two mechanism could be combined, possibly leading to more significant performance gains.

- In section 3.5 it was explained that the maximal size restriction of a VIPT cache can be extended by increasing the set associativity of the cache. If the cache blocks are combined as sets when lanes are coupled, this would allow each block to be sized to the page size. In the current implementation, the sets are combined to a direct-mapped cache. This means that the size of the entire cache is limited to the page size. This modification of the cache would allow a four-fold increase in the maximal cache size for an 8-issue ρ-VEX.

- When FPGA technology allows combining multiple ρ-VEXes on an FPGA, the technique discussed in 3.4.4 could be implemented to support easy level 1 VIVT caches. While this requires a second level of cache to be implemented, this architecture is the most favourable from the comparison performed in

Bibliography

[1] M. Cekleov and M. Dubois, "Virtual-address caches. part 2. multiprocessor issues," *Micro, IEEE*, vol. 17, no. 6, pp. 69–74, Nov 1997.

[2] R. Bryant and D. Hallaron, *OComputer Systems: A Programmer's Perspective 2nd Edition*. Pearson, 2010.

[3] S. Wong, T. van As, and G. Brown, "r-vex: A reconfigurable and extensible softcore vliw processor," in *Proc. International Conference on Field-Programmable Technology*, Taipei, Taiwan, Dec 2008.

[4] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The molen polymorphic processor," *Computers, IEEE Transactions on*, vol. 53, no. 11, pp. 1363–1375, Nov 2004.

[5] H. Wong, V. Betz, and J. Rose, "Comparing fpga vs. custom cmos and the impact on processor microarchitecture," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950419

[6] Maheshwari and Smid, *Introduction to Theory of Computation*. Ottowa, Canada: Carleton University, 2014.

[7] C. Iseli and E. Sanchez, "Spyder: a reconfigurable vliw processor using fpgas," in *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, Apr 1993, pp. 17–24.

[8] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An fpga-based vliw processor with custom hardware execution," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, ser. FPGA '05. New York, NY, USA: ACM, 2005, pp. 107–117. [Online]. Available: http://doi.acm.org/10.1145/1046192.1046207

[9] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A vliw processor with reconfigurable instruction set for embedded applications," *Solid-State Circuits, IEEE Journal of*, vol. 38, no. 11, pp. 1876–1886, Nov 2003.

[10] V. Brost, F. Yang, and M. Paindavoine, "A modular vliw processor," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, May 2007, pp. 3968–3971.

[11] C. Gaisler, *GRLIB IP Library Users Manual*, ser. User Guides, November 2015.

[12] J. Fisher, P. Faraboschi, and C. Young, *Embedded Computing*. San Francisco, CA: Morgan Kaufman, 2005.

[13] P. Faraboschi, G. Brown, J. Fisher, G. Desoll, and F. Homewood, "Lx: a technology platform for customizable vliw embedded processing," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, June 2000, pp. 203–213.

[14] HP. Hewlett-packard laboratories. vex toolchain. [Online]. Available: http://www.hpl.hp.com/downloads/vex/

[15] F. Anjam, L. Carro, S. Wong, G. Nazar, and M. Rutzig, "Simultaneous reconfiguration of issue-width and instruction cache for a vliw processor," in *Proc. International Conference on Embedded Computer Systems: Architecture Modeling and Simulation*, Samos, Greece, July 2012.

[16] A. Brandon and S. Wong, "Support for dynamic issue width in vliw processors using generic binaries," in *Proc. Design, Automation & Test in Europe Conference & Exhibition*, Grenoble, France, March 2013, pp. 827 – 832.

[17] Silberschatz, Galvin, and Gagne, *Operating System Concepts 8th Edition.* John Wiley and Sons, Inc., 2009.

[18] STMicroelectrics, *ST231 Core and Instruction Set Architecture Manual*, ser. ST200 VLIW Series, March 2004.

[19] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[20] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach 5th Edition.* Morgan Kaufmann, 2012.

[21] M. Cekleov and M. Dubois, "Virtual-address caches. part 1: Problems and solutions in uniprocessors," *Micro, IEEE*, vol. 17, no. 5, pp. 64–71, Sep 1997.

[22] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, Oct 2011, pp. 340–349.

[23] D. Black, R. Rashid, D. Golub, and C. Hill, "Translation lookaside buffer consistency: A software approach," *SIGARCH Comput. Archit. News*, vol. 17, no. 2, pp. 113–122, apr 1989. [Online]. Available: http://doi.acm.org/10.1145/68182.68193

[24] B. Jacob and T. Mudge, "Virtual memory: Issues of implementation," *Computer*, vol. 31, no. 6, pp. 33–43, jun 1998. [Online]. Available: http://dx.doi.org/10.1109/2.683005

[25] D. Bovet and M. Cesati, *Understanding the Linux Kernel, 3rd Edition.* O'Reilly, 2005.

[26] J. R. Goodman, "Coherency for multiprocessor virtual address caches," *SIGPLAN Not.*, vol. 22, no. 10, pp. 72–81, oct 1987.

[27] M. Chekleov, M. Dubois, J. Wang, and F. Briggs, "Virtual-address caches," Los Angeles, California, Tech. Rep., 1990.

[28] W.-H. Wang, J.-L. Baer, and H. Levy, "Organization and performance of a two-level virtual-real cache hierarchy," in *Computer Architecture, 1989. The 16th Annual International Symposium on*, May 1989, pp. 140–148.

[29] ARM. Page colouring on armv6 (and a bit on armv7). [Online]. Available: https://web.archive.org/web/20160201144045/https://community.arm.com/groups/processors/blog/2012/05/14/page-colouring-on-armv6-and-a-bit-on-armv7

[30] P. Weisberg and Y. Wiseman, "Using 4kb page size for virtual memory is obsolete," in *Information Reuse Integration, 2009. IRI '09. IEEE International Conference on*, Aug 2009, pp. 262–265.

[31] B. Jacob and T. Mudge, "Virtual memory in contemporary microprocessors," *Micro, IEEE*, vol. 18, no. 4, pp. 60–75, Jul 1998.

[32] Xilinx, *ML605 Hardware User Guide*, ser. User Guides UG534, October 2012.

[33] K. McLaughlin, N. O'Connor, and S. Sezer, "Exploring cam design for network processing using fpga technology," in *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, Feb 2006, pp. 84–84.

[34] K. Locke, *Parameterizable Content-Addressable Memory*, ser. Application Note XAPP1151, March 2011.

[35] J. van Straten, *r-VEX reference manual*, December 2015.

[36] J. Hoozemans, "Porting linux to the rvex reconfigurable vliw softcore," Master's thesis, TU Delft, Delft, 2014.

Reconfig 2015 Paper

A

Multiple Contexts in a Multi-ported VLIW Register File Implementation

Joost Hoozemans, Jens Johansen, Jeroen van Straten, Anthony Brandon, Stephan Wong

Computer Engineering Lab, Delft University of Technology, The Netherlands
Email:{j.j.hoozemans, a.a.c.brandon, j.s.s.m.wong}@tudelft.nl
{j.johansen, j.vanstraten}@student.tudelft.nl

***Abstract*—The register file is an expensive component in the design of any processor, especially, when considering the additional ports that are needed to support multiple datapaths within a wide-issue VLIW processor. In a recent work, these additional resources were used to dynamically reconfigure the register file to support a dynamically reconfigurable VLIW core. The design can be perceived as a single 8-issue, two 4-issue, or four 2-issue VLIW cores. Consequently, the multi-ported design can operate in different modes, namely as *one*, *two*, or *four* register files, respectively, corresponding to the active number of cores. The implementation of the register file design on FPGAs using Block RAMs still results in unused resources due to the coarseness of the Block RAMs.**

In this paper, we propose to re-purpose these unused BRAM resources to additionally support multiple contexts next to earlier-mentioned modes. In this manner, the 8-issue, 4-issue, and 2-issue cores have access to 4, 2, and 1 contexts, respectively. Consequently, we can avoid saving and restoring of the task states in a multi-task environment, turning context switching from a traditionally time-consuming event to an almost instantaneous event. The advantage of this is the reduction of interrupt latency and task switching latency, which are important in real-time and embedded systems.

Our results show that our technique can improve interrupt latency by a factor of 17.4× compared to using a software register spill routine, depending on the behavior of the memory system. Likewise, the task switching time can be improved by 6.7×.

I. Introduction

The ρ-VEX processor [1] is a dynamically reconfigurable VLIW processor that can adapt its organization to the requirements of different workloads. One of its most important run-time parameters is the issue-width that allows for adaptation towards the ILP of the task(s) at hand. The design can be configured as a single 8-way (1×8-way), two 4-ways (2×4-way), four 2-way (4×2-way) VLIW processor core(s), or combinations of those: e.g., two 2-ways and one 4-way. This capability requires the design of an extensive register file to support these different modes. In the worst case, the register file must provide:

- 8 write ports and 16 read ports when running in the 1×8-way mode
- 4 architecturally separate register files when running in the 4×2-way mode

This work has been supported by the Almarvi European Artemis project nr. 621439.

To design a register file that satisfies these requirements we use techniques such as Block RAM (BRAM) duplication and a Live Value Table (LVT), which we will discuss in Section II.

A major drawback of the current design is the large resource utilization. The BRAMs used to implement the register file on the FPGA need to be duplicated multiple times to provide the necessary amount of read and write ports. Every BRAM has a capacity of 512 32-bit words (2KiB); however, the architecture only requires 64 32-bit registers. Because of this, the resulting design has an enormous storage capacity of which at most an eighth is used by the processor in any particular configuration.

The design presented in this paper aims to convert the drawback of the high BRAM usage of the register file for wide-issue VLIW softcore processors into an advantage by using the overcapacity to store different execution contexts. The actual utilization of the BRAM storage capacity will increase from $\frac{1}{8}$ to $\frac{1}{2}$. Support for multiple contexts in hardware relieves the core from having to spill and restore its entire register file contents to and from memory in the event of a task switch or interrupt. In a multi-tasking environment, this concept changes task switches, which are traditionally very time-consuming, into a virtually instantaneous event. Faster context switching has advantages in numerous computing scenarios, as it will increase responsiveness for interactive workloads and improve interrupt latency and task switching speeds in real-time systems. In the following, we illustrate several cases in which our work can improve performance:

- Frequently used threads: Kernel threads, like schedulers, must be frequently executed. In a traditional core implementation, timers interrupt the core and trigger context switching in order to execute such threads. In our work, these threads can be maintained within the core and thereby remove the need for context switching. For example, an application is executing in the 8-issue mode using 1 out of 4 contexts. When the scheduler needs to execute, the current thread can be scheduled to run on a 4-issue core - this mode switch only takes several cycles when using generic binaries [2]. In the remaining 4-issue core, the execution of the scheduler can be resumed by using its own context that remained "dormant" within the core.
- Dynamic switching of execution by different cores: When threads require more resources, e.g., when their ILP

978-1-4673-9406-2/15/$31.00

increases, our processor design allows for it to claim additional datapaths to execute the code more efficiently. This does mean that another thread must be stalled for a while. However, in our case, the context of the second thread does not need to be saved into the memory and can remain within the core until it is resumed. In the latter, another context switching operation is saved.

- Context-cycling after cache misses: When our processor is running in the 8-issue (4-issue) mode, it can have up 4 (2) contexts stored within each core. This means that when one thread is encountering a cache miss, thus execution is stalled, the core can easily switch to another thread (context) and continue execution, i.e., Switch-on-Event Multi-Threading SoEMT.
- Embedded real-time systems with multiple tasks that require stringent real-time constraints (e.g., control loops with sensors and actuators). A single core can process more events using multiple contexts [3]. Therefore, a softcore can be used as microcontroller on an FPGA which would save the designer from having to design hardware circuits to handle some events or having to resort to a multi-core system where distinct events are handled by a dedicated core.

The register file of our ρ-VEX is a complex topic, as it is also instrumental in supporting the core's dynamic reconfigurability [4]. We limit the scope of this paper to evaluating the benefits from multiple hardware contexts. It must therefore be noted that the costs of this design (see Table I) are paid not only for multiple contexts, but also to support the dynamic reconfigurability. Our approach in this paper gives us a $17.4\times$ reduction in interrupt latency and $6.7\times$ reduction in context switching time.

II. Background

The multi-ported register file is a challenging component in the design of softcore VLIW processors. Wide-issue VLIW processors like the ρ-VEX need register files with a large number of read and write ports. The VEX instruction set architecture (ISA) supports operations that use two source registers and one destination register. Because of this, the number of write ports required is equal to the issue-width, and the number of read ports is equal to twice the issue-width. Creating such complex register files using FPGA LUT resources is very expensive and scales very poorly with the number of ports. The reconfigurable ρ-VEX design and the implementation of its multi-ported register file are introduced in [5]. Moreover, in [6], the idea of using a Live Value Table (LVT) is discussed that enables the use of banked memories with duplication to create multi-ported BRAM memories. The ideas presented in this paper are built upon a register file design that is implemented using this technique. We will discuss the concepts and challenges briefly in this section.

Creating RAM memories that have more read ports is straightforward and achieved by duplicating the BRAM and writing data into each block simultaneously. In this way, each BRAM contains the same data, and their read ports can be used independently of each other. Increasing the number of write ports, however, is more difficult. Several solutions exist in literature. The simplest solution is to divide the register file into banks, each connected to one of the write ports [7]. This solution restricts the range of registers each write port can write to and thus reduces the freedom the compiler has to schedule instructions. Another solution introduced in [8] increases the size of each bank to the original register file size and renames the registers in between the compiler and assembler. This solution enables a banked design with the same scheduling freedom as an actual multi-ported register file but utilizes a multiple of the number of registers. Note that this technique does not necessarily require more BRAMs since their size is a lot larger than the 64 registers specified in the VEX ISA. It does, however, increase the number of bits required to specify the source and destination registers in instructions.

The register file used in the ρ-VEX uses the technique introduced by [6]. This scheme also duplicates the register file for each write port. However, instead of uniquely naming the registers in each bank, a Live Value Table (LVT) keeps track of which bank holds the most recent value of each register. It uses this information to multiplex the right bank to the read ports, as shown in Figure 1. The LVT needs to be implemented as a multi-ported LUT based RAM because it still needs one write port per register file write port. However, since it only needs to hold a bank address, it is much narrower than the original register file that the scheme seeks to replace. While this technique enables the register file to be implemented mostly with BRAMs instead of LUTs, it still scales poorly with the number of ports. The number of BRAMs required is equal to the product of the number of read and write ports. The depth of the LVT scales linearly with the number of registers in the register file while the width scales logarithmically with the number of write ports. The number of ports required for the LVT is equal to the number of ports on the register file.

III. Related work

In [9] the authors analyzed the high requirements that wide-issue VLIW processors pose on the register file. They discuss hypothetical FPGA primitives similar to existing BRAMs but featuring many more read and write ports. These primitives do not exist in current FPGAs, therefore, the use of large BRAM or LUT-based structures is required to emulate this behavior [6].

In [10], it is stated that "the context switch time is one of the most significant overhead factors in any operating system" and shows that high timer interrupt handling latency can impede schedulability of real-time tasks. In [3], it is measured that using a multi-threaded architecture with 4 register sets allows an autonomous guided vehicle to run at a 28% higher velocity. In [11], measurements were performed to quantify the interrupt latency of several embedded Linux distributions running on a Xilinx Microblaze.

There are numerous examples of processors which use the concept of multiple register files to enhance the context

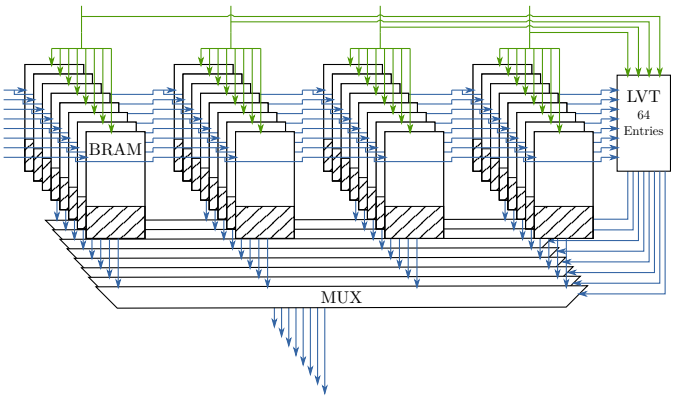


Figure 1. Block diagram of register file implementation using multiple banks of BRAMs. The green arrows indicate write ports, while the blue arrows indicate read ports. The shaded area represents the portion of the BRAM used for storing a single context.

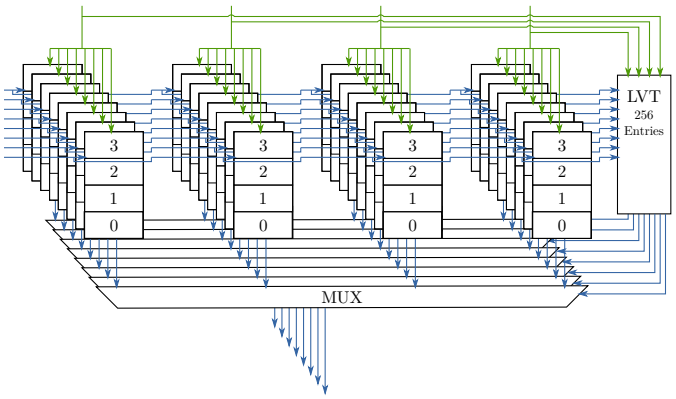


Figure 2. Block diagram of register file implementation supporting multiple contexts. Here the number of BRAMs is the same, but the LVT is larger.

switching time and interrupt latency in hardware. In [12], comparisons are made (by means of simulations) between increasing the number of cores and increasing the number of register sets in terms of increasing performance for a parallel workload. In [13], the MIPS architecture is extended by duplicating the register file multiple times and adding special instructions to switch between them when a context switch is required. In [14], the authors propose a novel architecture, which also supports holding multiple contexts in hardware simultaneously, and extend it with a dedicated cache to hold contexts to prevent spilling to main memory. Among other things the effects of the additional contexts on interrupt latency is investigated. Storing multiple contexts is also a requisite for (Simultaneous) Multi-Threading (SMT) [15]. An example of a VLIW processor with SMT support is the Itanium [16]. These technologies target high-end ASIC processors while this work targets the embedded (FPGA) domain.

The synthesizable ARPA-MT [17] and RTBlaze [18] processors also use SMT to improve schedulability and performance for embedded real-time systems. However, all the resource investments in this core are only used for SMT. The ARPA-MT core has a single execution pipeline. The fetch and decode circuits as well as the register file need to be duplicated for each thread slot.

In contrast, the ρ-VEX uses the additional resources to support: 1) a very wide VLIW to exploit ILP, 2) multiple hardware contexts and 3) a multi-core configuration (in other words, all contexts can be active and executing at the same time). Therefore, it uses the additional resources in a more efficient way compared to the previous work.

IV. Implementation

Figure 1 shows the implementation of a register file with four write ports and eight read ports ($4W \times 8R$), using BRAMs and an LVT. The $8W \times 16R$ version would be 4 times as large. The hatched area represents the part of the BRAM that is actually used to store the 64 registers used by the ρ-VEX. The figure shows that a large part of the BRAMs is unused. Because the ρ-VEX can be configured as four independent processors, it also needs four separate register files. However, the total number of read and write ports is the same for one large 8-issue processor or four separate 2-issue processors. Because of this characteristic, the same multi-ported register file can be used in each configuration. The number of registers, however, needs to be quadrupled, for a total of 256 registers, since each core needs a separate register file of 64 registers. The BRAM resources on contemporary FPGA boards provide more than sufficient storage capacity to accommodate this, so there is no added cost in BRAM resources. However, the LVT does need to increase in size, to keep track of the most recent location of all 256 registers.

Figure 2 shows how the multiple contexts can be stored in the previously unused space of the BRAMs. Creating four separate register spaces is a necessary cost to enable the ρ-VEX to be split into four separate processors. However, not all of the register spaces are used when the core is configured as a single 8-issue processor or two 4-issue processors. This creates the opportunity to re-purpose these unused register spaces as alternative register windows, which can be used to store the register context of inactive processes. Since the four register windows are implemented as a larger continuous address space, the uppermost bits can be used to select one of the four register windows.

The ρ-VEX utilizes more registers than just the 64 general purpose registers. It also has the following registers, that must be stored for a context switch:

1) A special 32-bit register used to store the return address for a function call (the link register).
2) Eight 1-bit registers used for conditional branching.
3) The program counter.
4) Various control registers, used for example for interrupt handling.

These registers cannot easily be stored in BRAMs, as the control logic needs to be able to access all these registers at once. Therefore, these registers are implemented in LUTs. To support running as 4×2-issue processors, all these registers need to be duplicated as well, and can thus be used as part of

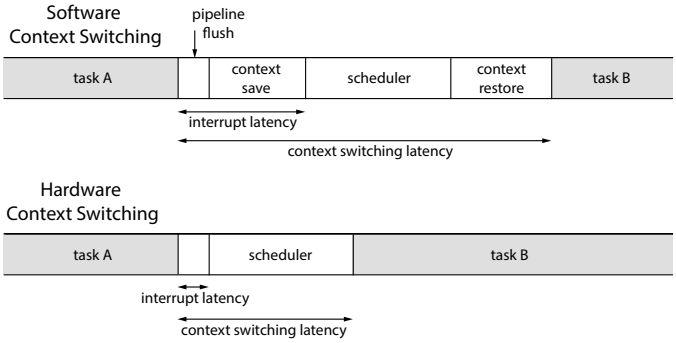


Figure 3. Context switching and interrupt latency definition.

the hardware contexts. Some additional hardware is required to use these registers for context switching, as not every lane would necessarily need access to all duplicates of the registers for reconfiguration only, while this is necessary for context switching. However, when this is done, the only registers which need to be spilled and restored are those registers which are used by the context switching routine, or scheduler itself. Because the additional hardware cost is small, our context switching design incorporates this feature.

A hardware context switch is not entirely free in terms of cycles in the current ρ-VEX design. To avoid complicating the forwarding logic, context switches are only possible when the pipeline is empty. Because the ρ-VEX has a five stage pipeline, five cycles are needed to flush the pipeline before a context switch can occur. In addition, the context switches are currently controlled by the dynamic reconfiguration controller, which takes three additional cycles to decode and commit a new configuration. Two of these are spent still executing instructions in the old context.

V. Experimental Setup

Our measurements are carried out using the ρ-VEX VLIW softcore processor clocked at 37.5 MHz running on a Xilinx ML605 development board, which incorporates an XC6VLX240T Virtex 6 FPGA. We use a timer connected to the interrupt request input of the processor to generate interrupts at different rates to measure the impact of our approach on the performance of the system.

We quantify the impact on performance by measuring two different values, namely:

1) *Interrupt Latency*: The number of cycles elapsed between the moment an interrupt request is received by the core, and the first instruction of the interrupt handler being executed.
2) *Context switching latency*: The number of cycles elapsed between the moment a context switch is requested (due to an interrupt), and the first instruction being executed in the new context.

Figure 3 shows what these latencies are made up of, namely: pipeline flushing, saving context registers, running the interrupt service routine (in our case the task scheduler), and finally restoring the context registers. By using hardware contexts the latency of saving and restoring registers can be eliminated. We measured these quantities by creating a workload of four programs. At every timer interrupt a scheduler selects a different program to execute, and performs the context switch to that program. The programs themselves have no impact on the measurements, since they are purely dependent on the time it takes to save and restore all context registers.

In order to measure the difference between hardware and software context switching, we wrote a software and a hardware context switching routine. The software version saves the complete context to the stack of the currently running task, stores the stack pointer to a predefined memory location, and starts executing the interrupt handler. The interrupt handler then calls the scheduler in order to schedule the next task. The current stack pointer is then replaced with the stack pointer of the new task. Next, the application context of the newly selected task is restored from the stack, after which control is handed back to the application. The hardware switch routine does not need to save or restore all registers. Instead it only has to do so for the registers used by the interrupt routine, in this case the scheduler.

The scheduler utilizes a linked list in memory to determine which task to switch to; each entry representing a task, with a mapping to another task. When a task completes, the linked list is rebuilt such that the context switching code does not switch back to the completed task, and a context switch is requested immediately using a software trap instruction. When the last task completes, it signals completion to the platform.

Because cache behavior will impact the latencies for saving and restoring the contexts we perform the measurements for different memory access latencies. We measure using latencies from 0 (single cycle memory access) to 30 cycle memory access on cache miss. The cache itself consists of a separate instruction and data cache, respectively 32KiB and 8KiB in size. The size has intentionally been kept small, because the programs under test had to be small as well for the entire memory to fit on the FPGA; it is assumed that, under normal circumstances, larger caches will be used, but the running programs will also use wider regions of more memory. Both caches have single-cycle hit latency for reads. The data cache has a two-cycle latency for writes for both hits and misses, as long as one of the four write buffers is vacant.

To evaluate the context switching overhead in multi-process time-sharing systems, overall performance of the multi-task system is tested on hardware using the cached system. The timer is used to generate an interrupt at a fixed frequency, often referred to as the system "tick," in which a context switch is performed. Clearly, the context switching overhead is directly related to the frequency of the system tick [10]. The frequency of the tick is usually in the order of 50 to 1000 Hz. A lower frequency will lead to lower switching overhead, but higher frequencies will result in a more responsive system. Systems that require more responsiveness will therefore have a higher tick frequency. For example, the Linux kernel uses a system tick of 1000 Hz for desktop systems, but this can be reduced to 100 Hz for server systems to reduce overhead. On the other

Table I
RESOURCE USAGE OF REGISTER FILE WITH AND WITHOUT SUPPORT FOR MULTIPLE CONTEXTS.

	Register File		Core	Increase over Core
	1 Context	4 Contexts		
Slice Registers	806	1392	8529	6.9%
Slice LUTs	10764	15591	35148	13.7%
RAMB18E1	128	128	147	0%
RAMB36E1	0	0	128	0%

hand, the Windows kernel uses 66 Hz. The frequency is varied between tests to evaluate its effect. In addition, the system is evaluated with varying bus latencies. The latencies used are estimates of what the average latency would be for a real off-chip memory system.

A cycle counter available within the ρ-VEX processor is used to measure the time from system reset to the program completion signal, which is given by the task switching implementation when all tasks have completed. For each timer and memory system configuration, both context switching implementations are evaluated. Because all other factors are kept constant, the difference in total execution time is only dependent on the context switching overhead. The speedup between the baseline and hardware context switching implementations is then determined to quantify this overhead.

VI. RESULTS

In Table I we show the increase in resource utilization of the register file when adding support for four contexts. As expected the number of BRAMs used does not increase. Only the number of registers and LUTs increases, since these are used to implement the LVT. While these increases seems large, when compared to the total usage of the core they are less significant. Additionally, note that this increase in resources in the register file is required to support the dynamic reconfigurability of the processor.

As we can observe in Table II, the interrupt latency is 87 cycles for software context switching. The interrupt latency when using hardware contexts is only 5 cycles, solely due to the pipeline flush performed by the trap handling logic. A full context switch, i.e., the time between a tick interrupt request and the execution of the first instruction in the new context, takes 174 cycles using the software implementation, compared to 26 cycles using the hardware contexts.

Table II
INTERRUPT AND CONTEXT SWITCHING LATENCY WITH SINGLE-CYCLE MEMORIES IN CYCLES.

	Software	Hardware	Reduction
Interrupt Latency	87	5	17.4×
Context Switch Latency	174	26	6.7×

In Table III, we can observe the results of the same experiments run using a cached memory system, with a bus latency of 20 cycles. We observe that the improvement due to the hardware context switching is greater in this system, with the improvement in interrupt latency increasing from 17.4 to 23.5×, and the improvement of context switching time increasing from 6.7 to 14.8×.

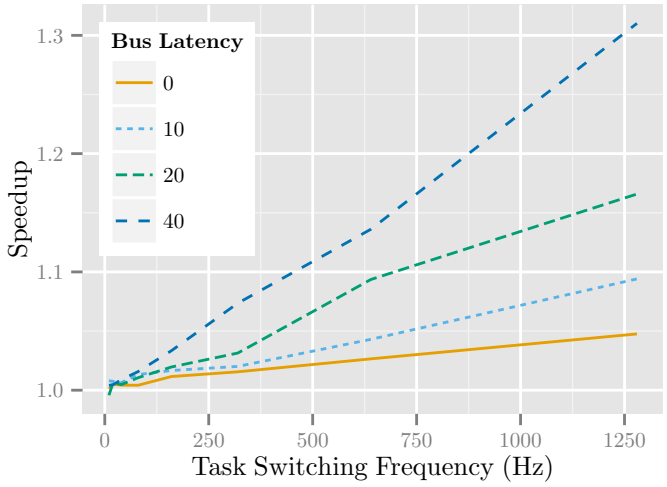


Figure 4. Speedup of the multi-task system due to the hardware context switching implementation.

Table III
INTERRUPT AND CONTEXT SWITCHING LATENCY WITH CACHE AND 20 CYCLES BUS LATENCY IN CYCLES.

	Software	Hardware	Reduction
Interrupt Latency	16798	713	23.5×
Context Switch Latency	31861	2148	14.8×

Figure 4 shows the speedup for different frequencies of the timer tick parameterized for different memory latencies, as measured on hardware using the cached system. It can be seen that in the region of higher task switching frequencies the difference between hardware and software context switching can be quite substantial depending on the memory system. A speedup of over 1.3× can be achieved for a bus latency of 40 cycles at a switching frequency of 1280 Hz.

VII. CONCLUSIONS

The concept of using additional register files to speed up multi-threading performance has been applied in numerous designs in the past. In this paper, we apply the concept to an existing design, exploiting the overcapacity of the BRAMs in the existing implementation of the multi-ported register file and the additional logic required by the parameterized reconfigurability of the ρ-VEX softcore. We have demonstrated that the proposed design can decrease the interrupt latency by a factor of over 20 times in a realistic environment. Likewise, the total context switching time can be decreased by a factor of over 10 times. In a simple multi-task system the effect of this is apparent as the decrease in overhead results in a speedup of 1.3× in the most extreme case evaluated. For applications with few real-time requirements, where the

system tick frequency would be relatively low, the speedup is negligible, as the task switching code would not be executed as often. However, embedded real-time systems that need to process large numbers of events will benefit most from the improvements.

REFERENCES

[1] S. Wong and F. Anjam, "The Delft Reconfigurable VLIW Processor," in *17th International Conference on Advanced Computing and Communications*, 12 2009, pp. 244–250.

[2] A. Brandon and S. Wong, "Support for dynamic issue width in VLIW processors using generic binaries," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 827–832.

[3] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, "Interrupt service threads-a new approach to handle multiple hard real-time events on a multithreaded microcontroller," *RTss WIP sessions, Phoenix*, pp. 11–15, 1999.

[4] F. Anjam, M. Nadeem, and S. Wong, "Targeting code diversity with run-time adjustable issue-slots in a chip multiprocessor," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[5] S. Wong, F. Anjam, and F. Nadeem, "Dynamically Reconfigurable Register File for a Softcore VLIW Processor," in *Design, Automation Test in Europe Conference Exhibition*, March 2010, pp. 969–972.

[6] C. LaForest and J. Steffan, "Efficient Multi-ported Memories for FPGAs," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. ACM, 2010, pp. 41–50.

[7] M. Saghir and R. Naous, "A Configurable Multi-ported Register File Architecture for Soft Processor Cores," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, vol. 4419, 2007, pp. 14–25.

[8] F. Anjam, S. Wong, and F. Nadeem, "A Multiported Register File with Register Renaming for Configurable Softcore VLIW Processors," in *International Conference on Field-Programmable Technology (FPT), 2010*, Dec 2010, pp. 403–408.

[9] M. Purnaprajna and P. Ienne, "Making Wide-issue VLIW Processors Viable on FPGAs," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 33:1–33:16, Jan. 2012.

[10] G. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011, vol. 24.

[11] A. Ronnholm, "Evaluation of Real-Time Operating Systems for Xilinx MicroBlaze CPU," Master's thesis, Malardalens University, 6 2006.

[12] R. Thekkath and S. Eggers, "The Effectiveness of Multiple Hardware Contexts," *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 5, pp. 328–337, Nov. 1994.

[13] N. Rafla and D. Gauba, "Hardware implementation of context switching for hard real-time operating systems," in *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS),* , Aug 2011, pp. 1–4.

[14] K. Tanaka, "PRESTOR-1: a Processor Extending Multithreaded Architecture," in *Innovative Architecture for Future Generation High-Performance Processors and Systems, 2005*, Jan 2005.

[15] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 392–403.

[16] R. Riedlinger, R. Bhatia, L. Biro, B. Bowhill, E. Fetzer, P. Gronowski, and T. Grutkowski, "A 32nm 3.1 billion transistor 12-wide-issue itanium processor for mission-critical servers," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, Feb 2011, pp. 84–86.

[17] A. Oliveira, L. Almeida, and A. de Brito Ferrari, "The arpa-mt embedded smt processor and its rtos hardware accelerator," *Industrial Electronics, IEEE Transactions on*, vol. 58, no. 3, pp. 890–904, March 2011.

[18] T. P. Wijesinghe, "Design and implementation of a multithreaded softcore processor with tightly coupled hardware real-time operating system," Master's thesis, 2008. [Online]. Available: http://search.proquest.com/docview/250936948?accountid=27026