NTNU – Trondheim
Norwegian University of
Science and Technology

Extending Amber with Virtual Memory

Jakob Dagsland Knutsen

Master of Science in Computer Science
Submission date: Januar 2015
Supervisor: Donn Morrison, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Since the birth of modern computer architecture, computer performance has witnessed an exponential growth, driven mainly by increased transistor density. In the last decade, performance advancement has become increasingly difficult as computers are limited by power budgets due to problems with heat dissipation. As transistor technology continues to advance, full utilization of the available silicon is becoming ever challenging. As a result, new avenues of design exploration have emerged to track down alternative routes for continued performance enhancement. One such research field concerns heterogeneous computer architectures, where cores that excel at different areas of computation are combined on a single chip in order to provide performance scaling and the assignment of process workloads to optimally suited hardware.

The SHMAC project applies heterogeneous research in a single ISA environment, where various hardware is deployed in a matrix of interchangeable tiles, with the only general purpose processor tile being an ARMv4T compliant Amber processor core.

While programs and operating systems have been proven to successfully run on the processor core, their practical use is limited by the available memory space as the core does not feature a memory management system.

This thesis introduces a memory management unit into the Amber processor, enabling virtual memory support at the hardware level. The final contribution is Vilma; an ARMv4T compliant core featuring a memory management unit, verified using hardware simulation and an assembly test suite.

Sammendrag

Siden moderne dataarkitektur først ble til har ytelsen til datamaskiner overvært en eksponensiell vekst. Denne fremgangen er først og fremst drevet av en økt transistortetthet. Gjennom det siste tiåret har forbedring av ytelse jevnt blitt vanskeligere å realisere på grunn av begrensede strømbudsjetter forårsaket av problemer med varmeutslipp. Ettersom transistorteknologien fortsetter å avansere begynner det å bli enda mer utfordrende å realisere full utnyttelse av tilgjengelig silikon. Som en følge av dette trer det frem nye veier i utforskningen av systemdesign, med det målet å finne alternative løsninger for å jobbe videre med ytelsesforbedring. Ett av forsknigsområdene innen ytelse omhandler heterogen dataarkitektur hvor kjerner som utmerker seg på forskjellige områder innen databehandling er forbundet på en enkel chip. Dette skal sørge for ytelsesskalering og tildeling av prosessoppgaver til optimalt tilpasset hardware.

SHMAC-prosjektet bruker heterogen forskning i et enkelt ISA-miljø hvor ulike typer hardware er plassert i en matrise som utskiftsbare brikker hvor den eneste prosessorbrikken er en ARMv4T ettergivende Amber prosessorkjerne.

Selv om programmer og operativsystemer har vist seg å kjøre uten problemer på prosessorkjernen, så er den praktiske bruken begrenset av tilgjengelig plass på minnet siden kjernen ikke har et fungerende minnehåndteringssystem.

Denne avhandlingen introduserer en minnehåndteringsenhet inn i Amberprosessoren og muliggjør støtte for virtuelt minne på hardwarenivå. Det endelige produktet er Vilma; en ARMc4T ettergivende kjerne som inneholder en minnehåndteringsenhet. Enheten har blitt verifisert ved bruk av hardwaresimulering og et assembly testrammeverk.

Problem Description

Task: Improving the AMBER-based Processor Core for the Heterogeneous SHMAC Multi-Core Prototype Current multi-core processors are constrained by energy. Consequently, it is not possible to improve performance further without increasing energy efficiency. A promising option for making increasingly energy efficient CMPs is to include processors with different capabilities. This improvement in energy efficiency can then be used to increase performance or lower energy consumption.

Currently, it is unclear how system software should be developed for heterogeneous multi-core processors. A main challenge is that little heterogeneous hardware exists. It is possible to use simulators, but their performance overhead is a significant limitation. An alternative strategy that offers to achieve the best of both worlds is to leverage reconfigurable logic to instantiate various heterogeneous computer architectures. These architectures are fast enough to be useful for investigating systems software implementation strategies. At the same time, the reconfigurable logic offers the flexibility to explore a large part of the heterogeneous processor design space.

The Single-ISA Heterogeneous MAny-core Computer (SHMAC) project aims to develop an infrastructure for instantiating diverse heterogeneous architectures on FPGAs. A prototype has already been developed. The current processor model is a 5-stage in-order pipelined AMBER processor. To ease software development and increase heterogeneity, this implementation should be extended.

The absence of virtual memory (VM) support is currently a major limitation to SHMAC software development.

The main task in this assignment is the development and implementation of virtual memory support for Amber. Specifically, the student shall design and implement a memory management unit (MMU) in hardware and ensure that the software component (e.g., operating system) manages the virtual address space and assignments to virtual memory from real memory. An important part of a virtual memory system is a mechanism for paging such that programs can make use of secondary storage. This has until now been a bottleneck in SHMAC, which has limited main memory. It is expected that the student will conduct a review on VM implementations, including in heterogeneous settings.

Preface

This thesis is submitted to the Norwegian University of Science and Technology.

Acknowledgements

I would like to thank my supervisor Donn Morrison, and co-supervisor Antonio Garcia Guirado, for their guidance. I would also like to give thanks to Benjamin Bjørnseth for keeping up with my questions and always striving to give thorough answers.

I would also like to give special thank Isa Agnete Halmøy Fredriksen for her support throughout this semester. I am truly grateful.

J.D.K.

Contents

List of Figures

List of Tables

Listings

Chapter 1
Introduction

As processor designs have undergone vast improvements in terms of performance over the last decades, the common areas exploited for this performance increase are becoming exhausted. Driven mainly by transistor technology, as predicted by Moore's law [18] and Dennard scaling [11], advancement has been achieved through increased clock frequencies and shortening of the critical path. Moore's law was predicted in 1965, and is the observation that the number of transistors that fit in a dense circuit double approximately every two years. Coupled with Dennard scaling, which states that power requirements are proportional to area, it has driven an exponential performance growth for decades. However, with the breakdown of Dennard scaling [8] in the first decade of the 21st century while Moore's law continues to hold, increased power demands challenges transistor technology, as insufficient heat dissipation is causing chips to heat up.

1.1 The Dark Silicon Effect

As cutting edge processor designs reached the point where it was impossible to power the whole core without damaging the chip due to the thermal increase, multicore processors allowed designers to maintain performance advancement without increasing the core clock frequency. However, the exploitable parallelism of multicore systems is reaching its limits, and with the continuation of Moore's law transistor count keeps increasing. Thus, processor design is once again faced with the problems of overheating and limitations of heat dissipation leading to wasted potential in unused transistors, otherwise known as the dark silicon effect [14].

1.2 SHMAC

Dark silicon is motivating the emergence of new fields of study, where different avenues of transistor utilization are explored. One such field is the area of heterogeneous processor design, where multicore processors are composed of heterogeneous cores

that have different performance and power attributes. By issuing workloads to the best suited hardware available, heterogeneous cores feature a potential to maximize performance for a given power budget. In addition, heterogeneous designs have greater potential to regulate power consumption by scaling performance according to demand, leading to a potential increase in overall energy efficiency. A use case where energy preservation is desirable in addition to periods with high performance demands, like in the mobile industry, is an applicable field for heterogeneous cores. ARMs big.LITTLE core is an example of a recent heterogeneous system that has entered the consumer market.

The EECS[1] group at NTNU contributes to heterogeneous research with the Single-ISA Heterogeneous MAny-core Computer (SHMAC) [4] research project. SHMAC provides an environment for heterogeneous computing research with regards to both hardware and software, and research focuses on exploring the challenges related to heterogeneous architectures.

Heterogeneous research is enabled through a tiled matrix layout, where each tile can be a different processing element, and each element can talk with its nearest enighbors. Figure 1.1 shows an example of a SHMAC layout.

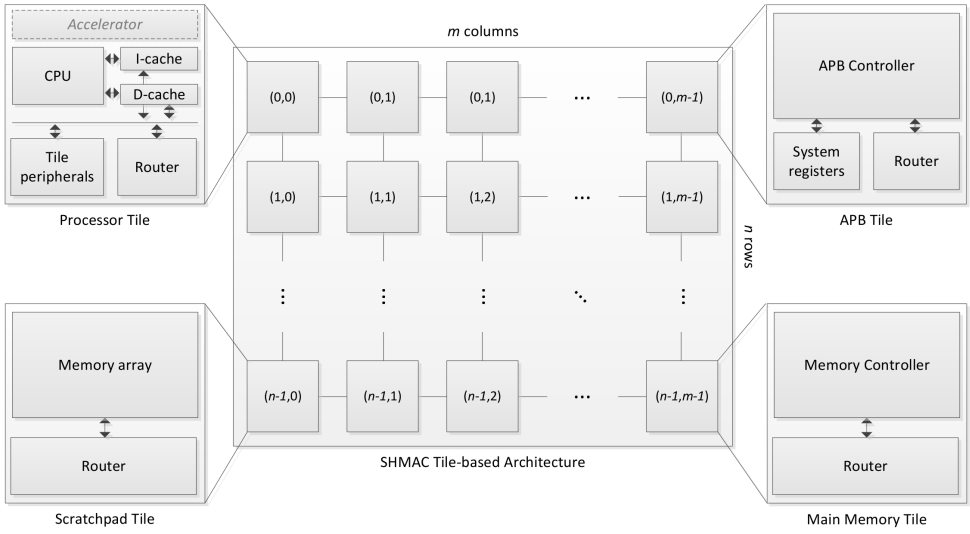

Figure 1.1: SHMAC High-level Architecture

The single-ISA vision in SHMAC ensures that every general purpose processing tile is able to handle any workload, regardless of differing characteristics in performance

[1] Energy Efficient Computing Systems

and power consumption.

1.3 Requirements

Requirements are used to systematically evaluate the success of the thesis. In this section I present a list of requirements as I have interpreted them from the assignment text, and explain the reasoning behind each point. Some requirements are taken from the assignment text, while others have been worked out in dialog with my supervisor.

Address translation	Vilma should be able to perform address translation in hardware.
Resource balanced	FPGA resource usage of Vilma should be balanced and not not greatly exceed that of Amber.
Implementable	The mechanism for address translation in Vilma should be possible to implement into the original Amber core and other variants of it.
Verified	Vilma should be extensively tested to verify the design.

Table 1.1: Listing of the thesis requirements.

1.3.1 Address translation

As specified in the assignment text, Vilma should be able to translate from virtual to physical addresses. Since the basis of Vilma, Amber, is an ARMv4T compliant processor, translation must proceed in accordance with expected behavior by software written for this architecture. This requirement is important because it determines the reusability of software that manage the virtual address space on ARM processors.

1.3.2 Resource balanced

As SHMAC research is performed on FPGA-chips the number of tiles in a setup is limited by the available on-chip resources. As the addition of translation hardware removes the memory bottleneck that has so far limited SHMAC software development, it would be counter productive to introduce new limitations due to resource exhaustion. FPGA-resources are measured in logic slices which inlude LUTs, and block RAM.

1.3.3 Portable

Introducing virtual memory to the SHMAC environment implies the use of virtual addresses across a complete tile setup. The introduced mechanism for address translation should thus be portable to Amber and variants thereof, such as the Turbo Amber core [5]. The single-ISA vision of SHMAC underlines the importance of this

point, as code that is compiled for a virtual address space must be runnable on any tile setup. This requirement demands engineering of a system with minimal implications on existing hardware.

1.3.4 Verified

Although not explicitly specified by the assignment text, verification of the design is implied, and is considered the most difficult part of the project. Considerable effort should be made to ensure the functional correctness of the design.

1.4 Contributions

This thesis contributes to the SHMAC project with the digital design of a memory management unit, written in Verilog and implemented into the Amber core. The MMU supplies the necessary hardware support to employ virtual memory as the system memory management technique, allowing main memory to be virtually extended into secondary storage. The most significant benefit of being able to use virtual memory is the elimination of the memory bottleneck that has previously limited SHMAC software research. In addition, access control hardware has been introduced to supply software with the necessary framework for applying memory protection.

1.5 Report Outline

The rest of the report is structured as follows:

Chapter 2: Background gives a historical overview of virtual memory and supplies the information necessary to understand how it works and the role it plays in a modern computer. It explains the benefits involved, and shows how performance penalties can be mitigated. It also looks closely at the literature used for development, the ARM architecture reference manual [6].

Chapter 3: Vilma details the design and implementation of the thesis contributions.

Chapter 4: Evaluation & Discussion looks at design verification and discusses the developed unit, how it turned out in terms of performance and size, and talks about problems that were encountered during development.

Chapter 5: Conclusions and Further Work lists the results in light of the assignment requirements, and discusses some relevant areas for further development.

Chapter 2
Background

The emergence of modern computer science in the 20th century gave rise to a field that has seen significant changes over the seven decades since its birth.[1] Technological revolutions have transformed digital computers from constructs taking up whole floors[2] into tiny devices of just a few grams.[3] Even so, the basic principle that a computer consists of a processing element and some type of memory still remains, as one has no function without the other. This inherent dependency between processing element and memory implies a limitation where not only the speed at which the processing element can process data, but also the amount of data it can process, relies on the memory's ability to serve a request.

The availability of high-speed computer memory has a direct impact on performance, and large amounts have generally always been expensive. Insufficient amounts of main memory due to program size and lack of hardware motivated programmers to introduce systems for memory management as early as in the 1940's and 1950's. Denning [12] describes how programming at that point was performed at a very basic level and required a thorough understanding of the underlaying hardware. As such, the programmer would identify the independent blocks of code in a program, and instruct which block would be in memory at different points in time during execution. This technique, called overlaying, was sufficient for the time as there was little demand for resource sharing between programs and the programmer usually had direct hardware access.

After higher level programming languages were developed in the mid 1950's, layers of abstraction supplied greater tools for development and focus began to move away from machine details and over to software. As the complexity of programs increased, so did the the development overhead involved with overlaying.

[1] The Z3 of 1941 was the first working programmable, fully automatic digital computer [21].
[2] The ENIAC of 1946 was roughly 2.4m by 0.9m by 30m and weighed about 27 tons [**?**].
[3] Microcontrollers can typically weigh just a few hundred grams and have more processor power than the ENIAC [17].

Significant performance penalties would ensue under the employment of inadequate overlaying strategies, and as development continued the burden of constructing a sufficient memory management system increased. This led to demands for systems with large amounts of main memory [2]. As demands in turn increased for resource sharing between processes, security issues became a problem due to the fact that software with direct memory access may interfere with the process state of other programs.

In 1961, research conducted at the Manchester University during the development of the Atlas computer [13][16] introduced a new technique which efficiently solved the issues related to memory management. The techinque is called virtual memory, and has had profound influence on computer architecture since its invention.

In this Chapter I will provide an overview of virtual memory as a memory management technique and show how the ARM memory system is built. The Amber core is discussed in light of how the ARM Architecture Reference Manual for ARMv4T employs the memory management unit, and methods of fast address translation are also introduced.

2.1 Virtual Memory

In the following section I will introduce the components necessary to understand what virtual memory is and the role it plays in a computer system. The discussion will start by looking closer at the motivation behind memory management, and introduce all hardware and software systems that play a part in the enabling and execution of virtual memory as a memory management system.

2.1.1 Memory Hierarchy

In order to explain virtual memory it is first necessary to understand the memory hierarchy as well as how the computer handles a running program. The chosen computing system model is as depicted in Figure 2.1, showing the relevant hardware as the processor, volatile main memory, e.g., RAM blocks, and non-volatile secondary storage, e.g., hard disk or flash memory. In this document, *main memory* will also be referenced as *physical memory*, or simply *memory*. The software pieces involved are the operating system and any number of running programs present in main memory.

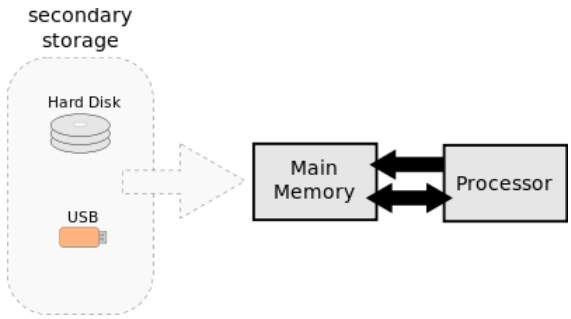

Figure 2.1: Processor, main memory, and secondary memory in a computing system.

When power is off, all information resides on secondary storage. As the processor boots up it loads data from secondary storage into main memory, the first program being the operating system. The operating system can then initiate a running instance of another program stored on secondary storage, by moving it into main memory. A running program is called a process.

2.1.2 Memory Management

This subsection is introduced to provide a more detailed explanation of the motivation behind memory management, and the challenges it faces.

To run a process on a computer there are two main criteria. One, all program code must be accessible or the processor will not have instructions to execute. Two, all existing program data has to be accessible, including any new data that may be generated during runtime, or the process state will be inaccurate and cause the program to malfunction. For both code and data to be accessible, it must exist in main memory.

Now consider a 32-bit system with 2GiB of physical memory installed, with no operating system and only one running process.[4] Since the byte-addressable address space of 2^{32} is 4GiB, the lack of physical memory is cause for an unused potential of 2GiB of the address space. Still, the program can be coded to start at address zero with an address space of 2GiB and all will be fine. Problems start to arise however, when a second process is introduced to the system. At which address should the new process start, how much space should it be given, and how is the first process notified of the newly introduced process. What if the first process generates data that enters the memory area of the second, overwriting data or instructions. And

[4]For the purpose of the example assume that the process can load without the help of an OS.

how can a program ensure the integrity of its own process state when other software may interfere with its data.

In summation, memory management is challenges in three areas;

1. Unused address space
2. Resource sharing
3. Protection

2.1.3 Introducing Virtual Memory

Virtual memory uses a combination of hardware and software to create an abstraction of main memory that solves the challenges with memory management. The software component is part of the operating system and manages a file structure, called a page table, for each process, containing virtual to physical address mappings. The hardware component uses the contents of the page table to perform fast address translation during runtime.

By splitting physical memory pages and adding the layer of translation, a virtual address space is created [20] where it is possible to move pages in and out of memory without breaking process execution. Figure 2.2 shows how a process may have addresses mapped to pages that reside in both memory and secondary storage. A result of this is that processes can use the whole 32-bit address space, given there is enough space on disk.[5] In addition, it does not matter where in memory a block is mapped to, which means a program can run from any location in main memory. This feature is called *relocation* and simplifies loading programs for execution.

To elaborate, consider again what happens when a program is instantiated on a computer. The operating system moves the executable file into memory and the processor starts executing the instructions within. Some of these instructions may be used to prepare the initial environment for the process, and once executed will not be referenced again for the duration of the process. The observation that memory may contain inactive code, and that the active part of a program differs during process execution, gives ground for a memory management system where only the active parts of a program are kept main memory. Inactive parts may then be moved to secondary storage, freeing memory space.

To employ a virtual address space a system needs to be able to perform the necessary translations, as well as manage the mappings. Address translation and management

[5] The virtual address space is generally larger than the physical address space, although it does not have to be and exceptions can occur for systems where external storage is limited.

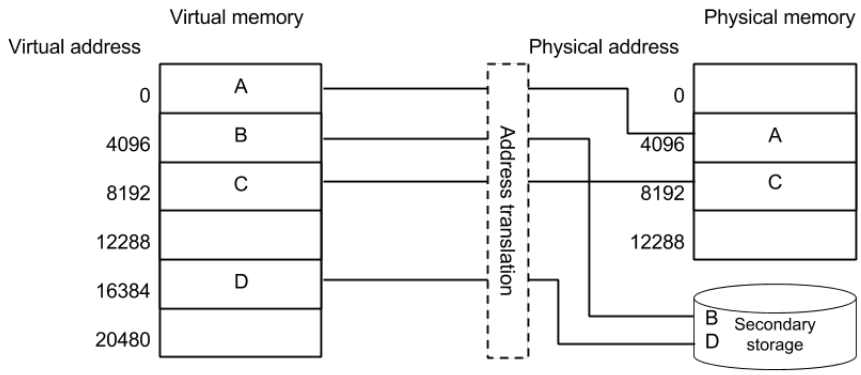

Figure 2.2: Pages residing in memory and secondary storage

is performed automatically by the Memory Management Unit *(MMU)*,described below.

2.1.4 Memory Management Unit

The MMU can be thought of as a unit that extends into two places, with one piece in hardware and another in software, however only the hardware is referenced as the MMU. The MMU is the logic that performs the actual address translation, while the software builds and manages page tables. The software is usually contained within the operating system. Together, these two pieces make up the system that applies the necessary translation layer required for virtual memory.

To keep track of process data, the OS keeps a data structure for every active program where it maps virtual page numbers to a physical page number or disk location. This map is called a *page table*, and the blocks of data are *pages*. A page location recorded in the page table is called a *page table entry*. When a reference occurs to a page that is not present in main memory the hardware will notify the OS. The OS is in turn responsible for bringing that page into memory, and updating the corresponding physical page number of the page table entry.

Figure 2.3 shows an example of how address translation works. The virtual page number is looked up in the page table and the corresponding physical page number is used for the memory access. The page table, indexed by the virtual page number, is sized at the number of pages in the virtual address space. With 32-bit addresses, 4KiB pages, and 4 bytes per page table entry, this adds up to $(2^{32}/2^{12}) * 2^2 = 2^{22}$. Consequently, the OS keeps a 4MiB table for every process.

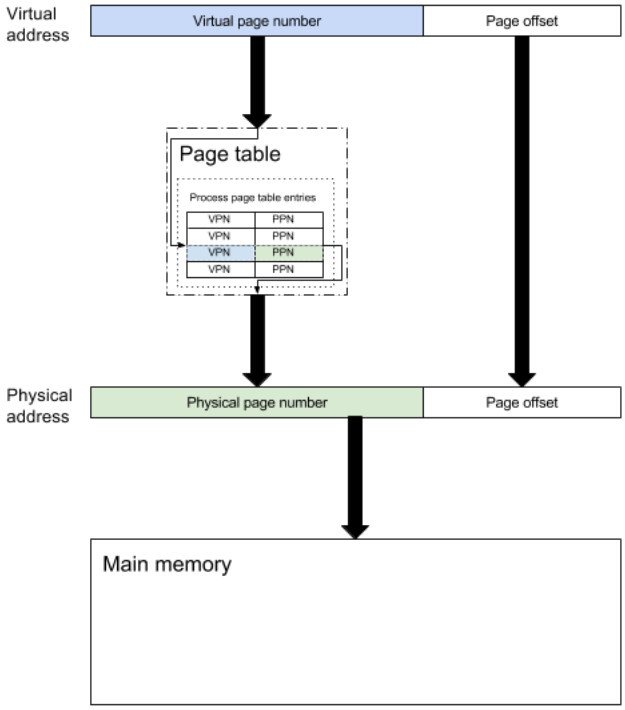

Figure 2.3: Address translation

2.1.5 Virtual Memory in Operating Systems

In theory, the maximum addressable area is limited only by the processor architecture *e.g.* 4GiB for a 32-bit system. However, kernel data vital for system operation must always be present in memory, which excludes portions of the address space from paging. The exact method used to achieve this behavior may vary between OSes, but for Linux and Windows the addressable area is divided into kernel and user space. A common split is 1GiB/3GiB for 32-bit systems, but Windows will in some cases split it 2GiB/2GiB. A process may only use addresses of the user space, while the OS may access the address space in its entirety.

The OS maps the virtual address space for each process through the page table, and allocates page entries on demand. On page faults, the page fault handler is activated by the processor as it branches to a predefined address where this handler starts, and it is up to this software to determine what to do next.

2.2 Fast Address Translation

A system using virtual memory may take a significant performance hit as the amount of memory accesses are effectively doubled; one access to translate, another to perform the operation. This penalty grows even larger in systems where the page table itself is paged. It can, however, be significantly reduced by adding a buffer for the translations. This buffer exploits the high spacial locality associated with address translation [20], and is called a translation lookaside buffer, or TLB.

2.2.1 Translation Lookaside Buffer

The TLB functions as a translation cache, tagged by the virtual address, and holds the corresponding translation in the data field. Each translation is accompanied by a reference bit, valid bit, dirty bit, and a protection field.

The reference bit is used by the replacement algorithm of the OS, while the dirty bit signals whether the corresponding page has been modified (not the TLB entry itself). The vaild bit is used to determine if an entry is active, and the protection field holds access permission bits associated with the page. The operating system manages these bits by updating the page table and invalidating the corresponding entries.

2.3 ARM MMU

The ARM architecture used in this thesis is ARMv4T, which is the current version of the Amber core used in SHMAC [15]. More on Amber can be found in Section 2.5. ARM DDI 0100E [6] is the ARM Architecture Reference Manual used in the development. While this manual covers versions up to ARMv5TE, the Preface on *Architecture versions and variants* states no difference in MMU architecture from v4 to v5, and is assumed safe to use.

2.3.1 About the MMU Architecture and System Overview

ARM is a Harvard Architecture and consequently employs separate instruction and data memory, which means there must be separate MMU and TLB for each memory interface [6]. Figure 2.4 shows how the memory system is organized. The green areas are part of the MMU design while the rest is present in the Amber core.

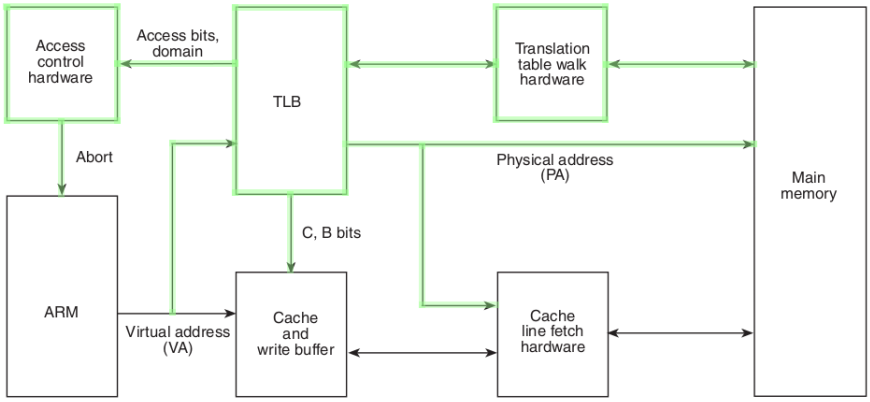

Figure 2.4: Cached MMU memory system overview, from [6] p. B3-4

A memory access issued by the ARM core starts with a virtual address passed from the core to both cache and TLB. If the requested data is cached we have a hit and normal operation follows the next cycle. On a miss, the TLB will walk the page table and return the physical address to the cache. Each memory access also passes through access control hardware that will raise an MMU fault for invalid accesses, invoking the OS access permission handler.

2.3.2 MMU Control

Control of the MMU is done via a coprocessor. The ARM core offers support for up to 16 coprocessors, using 4 bits to address them.[6] Some coprocessors are reserved for specific functions, like coprocessor number 15 which is the designated system control coprocessor.

The MMU is controlled by coprocessor number 15, the system coprocessor, using coprocessor instructions *mrc* and *mcr* on registers 2, 3, 4, 5, 6, 8 and 10, and some bits of register 1 [6].

Writing to a coprocessor register triggers a hardware response, but does not necessarily mean that data is written to a particular register. Only data that may be required more than once cycle is registered, like the MMU enable bit. Table 2.1 describes the different registers, the data field bits, and their functions. A complete description can be found in [6] B3.7.

[6]There are 16 coprocessors where the lowest is number 0 and the highest is number 15.

Reg	Bits	Function
c1	[0]	MMU enable/disable
		0 = disabled
		1 = enabled
	[1]	Alignment fault checking enable/disable
		0 = disabled
		1 = enabled
	[8]	System protection bit, used in Access permissions
	[9]	ROM protection bit, used in Access permissions
c2	[31:14]	Translation table base
c3	[31:0]	Domain access control
c4	[31:0]	Reserved
c5	[3:0]	Fault status: Indicate the type of access being attempted
	[7:4]	Fault status: Specify which domain was being accessed
	[8]	Fault status: Returns zero
c6	[31:0]	Fault address
c8		TLB invalidate functions (shootdown)
c10		TLB lockdown

Table 2.1: Listing of MMU control via system coprocessor registers

2.3.3 Translation

The MMU supports the use of pages and sections. Pages can be tiny, small, or large, with a block size of 1KiB, 4KiB, and 64KiB, respectively. Sections are comprised of 1MiB blocks of data. Sections and large pages are supported for mapping large regions of memory while using only a single entry in the TLB. For pages, the translation table is split into two levels that hold first- and second-level descriptors.[7] The translation process runs in hardware on every TLB miss occurrence, and is depicted as the *translation table walk hardware* box of Figure 2.4.

The first step retrieves the first-level descriptor by combining the contents of the translation base register (system coprocessor register 2) with the first-level table index of the virtual address. The first-level descriptor contains the base pointer to the second-level descriptor in addition to domain access control bits. The domain bits are used for setting access permission to large areas at a time in conjunction with access permission bits of the second level. The address for the second-level descriptor is formed by the page table base address of the first-level descriptor and the

[7] The process for a small page table is presented due to its use in the Amber system.

second-level table index of the virtual address. The second-level descriptor consists of the page base address (the physical page number) and access permission bits.

2.3.4 Page Faults and Access Control

Access control is performed by the access control hardware depicted in Figure 2.4, which notifies the processor of errors that occur during address translation. This is done through the use of an abort signal coupled with status bits that detail the fault specifics, and are acted upon differently depending on the source of the error. For a data access, a Data Abort signal is passed to the processor and the Fault Status Register and Fault Address Register (registers 5 and 6 in Table 2.1) are updated with information detailing the exception. For instruction fetch, a Prefetch Abort signal is issued to the processor. No additional information is collected in the case of a Prefetch Abort, and the exception is only handled if the instruction executes. This is done to avoid handling an exception for an instruction that does not execute, for example, due to a branch. By not setting the Fault Status Register and Fault Address Register for a Prefetch Abort, data integrity for these registers is ensured in the case where a Prefetch Abort would corrupt the data before being branched past. On any abort, the processor will branch to a defined address where the appropriate abort handler resides, allowing the OS to apply the appropriate actions.

Figure 2.5 shows the fault checking sequence of the ARM processor. The points of interest are the descriptor fault checks after each descriptor fetch, which checks bits[1:0] of the descriptor. The value 'b00 determines an invalid descriptor, where the first descriptor check will signal a section fault and the second a page fault. Domain and access permission checks verify user access rights.

External aborts are caused by errors in the memory system rather than those caught by the MMU. These are expected to be extremely rare and likely fatal to the running process.

2.3.5 Walking the Page Table

On a TLB miss the ARM TLB will automatically look up the missed entry in the page table. To look up a page table entry, memory accesses are placed on the memory bus twice; once to retrieve the first-level descriptor, and once again to retrieve the second-level descriptor. Due to the way page table entries are retrieved, the action is referenced as a *page table walk*. A high-level overview of the hardware is shown in Figure 2.4.

Figure 2.6 details the process of translation for a small page. Bits [31:14] of the translation table base register in the coprocessor are concatenated with bits [31:20] of the virtual address along with two zeros, forming the 32-bit address of the fld.

Figure 2.5: Fault checking sequence, as seen in [6] p. B3-20

The two zeros ensure the beginning of each word is addressed in the byte-addressable address space. The fld address is then placed on the memory bus until the fld is retrieved.

Looking closer at the fld, bits [1:0] are used to determine page presence in memory and the value 'b00 will trigger a page fault. The value 'b01 indicates a valid first-level descriptor. The address for the sld is formed by fld bits [31:10], concatenated with bits [19:12] of the virtual address, and two zeros. In reference to bits [9] and [4:2], SBZ stands for *should be zero* and the meaning of IMP is *implementation defined*. The domain bits are used by the access control hardware.

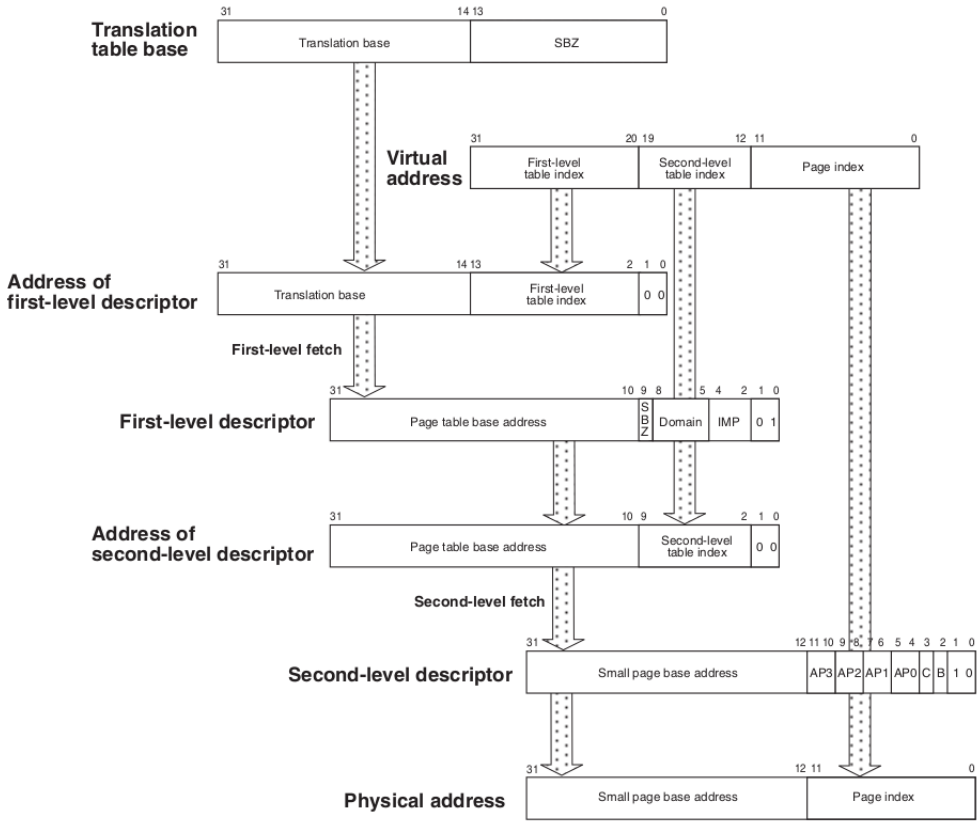

Figure 2.6: Small page translation, as seen in [6] p. B3-14

Looking at the second-level descriptor, bits[1:0] determine page validity in the same way as for the fld, where the value 'b00 will trigger a page fault. Bits [11:4] are used by the access control hardware, and the C (cacheable) and B (bufferable) control the cache and write buffer. Bits [31:12] represent the physical page number, and concatenation with the page index forms the physical address.

2.3.6 TLB and Caches

PIPT caches are simple and avoids problems with aliasing as described by [9][10], but suffer a performance penalty as the virtual address must be translated before it can be sent to cache.

There are two ways to solve this problem, where one is more efficient than the other at a cost of increased complexity. The simplest way is to stall the core while the

TLB reads, adding a delay of one cycle for every instruction. The more advanced way is to pipeline the MMU, increasing the penalty of pipeline flushes. Once again, simplicity has been favored over performance, and the former solution was chosen.

2.4 Virtual Memory in Multicore Systems

Early recognition of TLB influence on system performance steered uniprocessor designs into adapting the multilevel hierarchies used for caching in TLBs. As chip multiprocessors become more mainstream, TLBs have been exposed to coherency issues with implications on performance. Various strategies have been explored to increase performance and simplify design. A common design is to keep an MMU and the TLB hierarchy for each core, (and perform TLB shootdowns when pages are invalidated?)

Research done by [7] suggests shared last-level TLBs perform better than private per-core L2 TLBs in addition to offering a simpler design, while [22] demonstrates the benefits of a different coherence protocol.

2.5 Amber

The Amber processor core is the processor into which the MMU detailed in this thesis has been implemented. Amber is an ARM-compatible 32-bit RISC processor that supports the ARMv2 ISA, created by Conor Santifort [1], and comes in two versions; a 3-stage pipeline named Amber23, and a 5-stage pipeline named Amber25. Amber25 is the processor this thesis focuses on. The system features an external communications interface named Wishbone, an opensource system-on-chip interconnect architecture [3] that is used for memory communication and peripheral control. The five-stage pipeline featured in Amber25 is depicted in Figure 2.7. The five stages are fetch, decode, execute, memory, and write-back, and there are separate L1 instruction and data caches. There is only one level of cache.

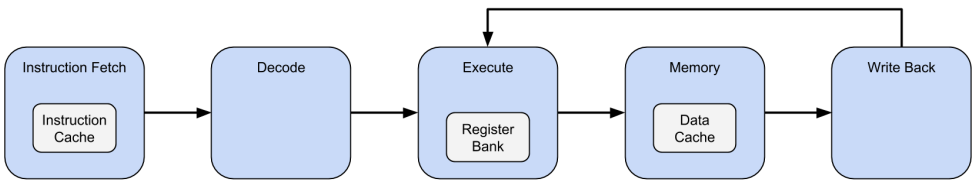

Figure 2.7: The Amber core

The program counter is driven by the execute stage and supplies fetch with the next instruction address. An unregistered version of the address is sent directly to the cache, and in the following cycle the cache tag output is compared to the registered version of the address that entered the fetch stage. If the cache misses, the pipeline is stalled until data is retrieved over the wishbone bus. On a cache hit, the data is passed on to the decode stage where the instruction is decoded and sent on to the execute stage in the following cycle, and so on. Performance of the two remaining pipeline stages, memory and write-back, differ somewhat from traditional five-stage pipline implementations, like the MIPS pipeline shown in [19]. Here, the register bank is updated by the write-back stage, whereas in Amber registers are updated in the execute stage. To avoid data hazards, Amber stalls the pipeline during load and store operations.

The SHMAC project has continued development of the Amber core to support the more recent ARMv4T ISA, done by Andersson and Amundsen in the spring of 2014 [15]. The most significant improvement in relation to this project was the addition of the *CPSR* and *SPSR* registers, used to save program status during exception handling. Amber has previously been verified by booting Linux, although without virtual memory enabled due to the lacking hardware [15].

The Amber project also comes with a test suite that run assembly code on the core which is used to verify correct behavior. The test suite can be extended to verify the integrity of new designs, and will notify the user should any functionality be broken. At the point of writing it consists of 89 tests written in assembly.

2.5.1 Wishbone Bus

The wishbone interface supplies a communications bus that serves as the memory interface between the core and external SRAM. It features an interface with four inputs and two outputs. The write signal and input data are only involved in the case of writes, and for modules that only do reads these ports are grounded.

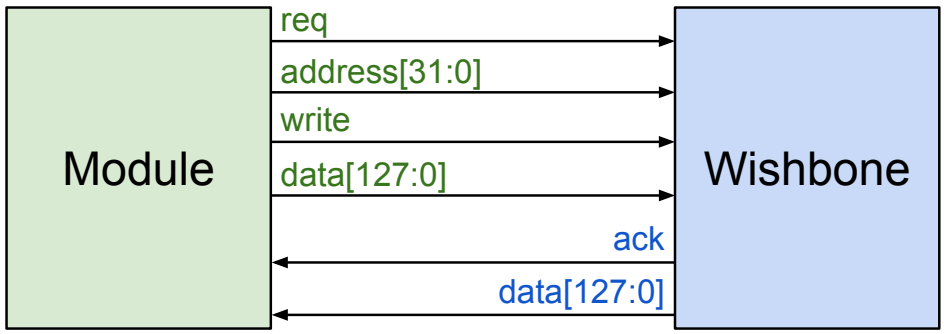

Figure 2.8: Wishbone interface

Figure 2.8 shows how the wishbone interface looks from the perspective of any module in the processor. Asserting *req* places a memory request for the accompanied address. The wishbone module will then handle the memory access, and pulse *ready* to notify when data is ready on the data bus. As the data bus is 128 bits wide it can carry up to four words.

The wishbone module will also store simultaneous requests and give priority in a predefined order. Since the pipeline stalls during memory reads, modules that feature a wishbone connection must have a state machine that is not blocked by the core stall signal, otherwise the core could get stuck in a deadlock. Data is assured to be available for one cycle before before any new request is served.

Chapter 3
Vilma

This Chapter describes the changes made to Amber in order to make it support virtual memory. The core is named Vilma - VIrtuaL Memory on Amber. The chapter starts by laying out some of the design decisions before giving an architectural overview of the implementation. Next, the MMU and TLB implementations are shown, followed by the access control and table walk hardware. The edits made to the coprocessor are shown last.

3.1 Design Decisions

Except bits that can be implementation defined, the ARM Architecture Reference Manual [6] leave little room for alternative design choices on an ARM core. However, certain shortcuts can be made possible by simplifying Vilma where performance is would otherwise be gained through increased complexity. Two main shortcuts were made in the project. First, Vilma supports only small pages. Tiny and large pages as well as sections have to be enabled by software, and since 4KiB is a widespread default small pages were deemed sufficient. Second, the TLB is direct mapped. TLBs are commonly designed as fully associative to provide storage flexibility in a buffer where balancing size versus speed is critical. By comparing the outputs of each register, a fully associative TLB gains speed through complexity. Dropping the comparators meant a substantial simplification of the design, and consequently reduced development time. Since the main focus of this thesis has been to get a working MMU implemented into the core, simplicity has been favored at the cost of speed.

Further, caches have been left untouched and are considered virtually indexed, physically tagged (VIPT). This was possible because the caches are sized by the address offset.[1]

[1] bits [11:4] of the address

3.2 Architectural Overview

With the addition of this thesis the Amber core has gained the three functional units shown in Figure 3.1. While the coprocessor was already present, it has undergone significant changes to the point where only fractions of the previous design remains and is consequently presented as a new unit.

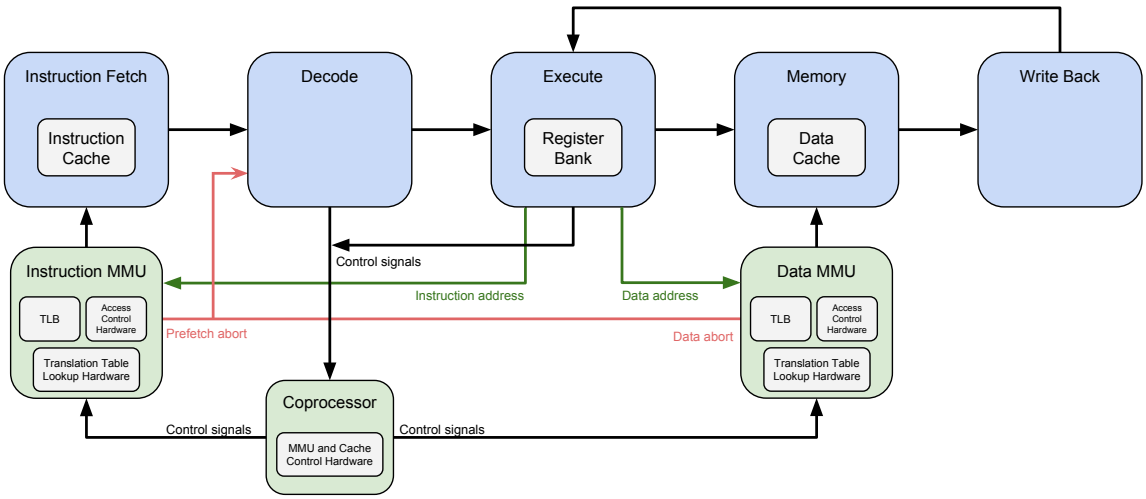

Figure 3.1: Architectural overview of Vilma

Due to the ARM core featuring split memory interfaces, translation is required in two places. While a single MMU could do the job, it would consist mainly of duplicate logic, and separate TLBs are needed regardless. Consequently, there is one MMU for each memory interface and each MMU manages its own TLB. The main difference between the two units is that the data MMU deals with both load and store instructions and needs to check access permissions accordingly, while the instruction MMU will handle nothing but loads and consequently need only check read permissions.

Instead of going directly out from Execute to Memory and Fetch, addresses are wired via the respective MMU for translation. Should the MMU be disabled, addresses are forwarded instantly.

3.3 MMU Implementation

To help the reader understand how the MMU fits into the pipeline I will first explain the flow of instruction addresses in an MMU-less pipeline. Figure 3.2 shows the involved pipeline stages and hardware. The oblong rectangles depict pipeline registers, where computed data is stored on every positive clock edge. The instruction cache is really a module inside the Fetch stage, but to simplify this figure it is placed on the Execute pipeline as there is no functional difference in regard to this example since the signal is forwarded.

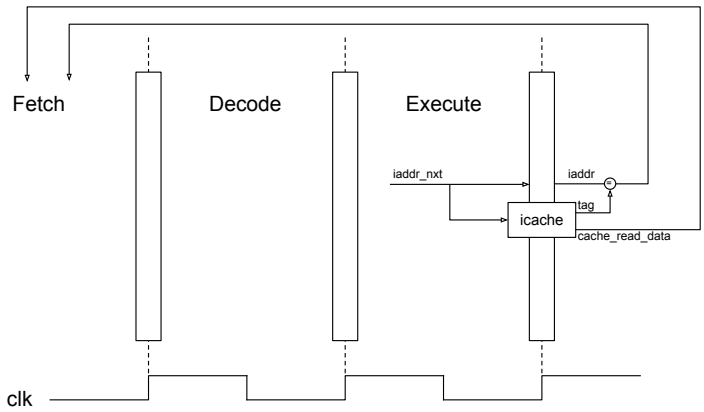

Figure 3.2: Instruction address flow in Amber

In between clock signals, Execute will produce the next instruction address. On the next clock cycle, the address is written into a pipeline register and subsequently read by Fetch. In the case where caches are disabled, Fetch will proceed to stall the core and read the instruction from memory, releasing the stall when the data is ready on the memory bus to be written into a pipeline register in the Fetch module. When caches are enabled the procedure differs. Cache needs a cycle to index its registers, so in order to provide cache with a clock signal without stalling the pipeline the address is forwarded from execute. In the following cycle, if the cache tag matches the address from execute, cache hits and Fetch will receive the data instantly and there will be no stalling involved. On a cache miss the procedure is similar to what happens without caches enabled, only that once fetched the instruction is also written to cache.

As mentioned in Section 3.1, the caches are physically indexed meaning that address translation is only necessary at tag comparison when virtual memory is enabled.

Thus, cache and TLB can be accessed concurrently.

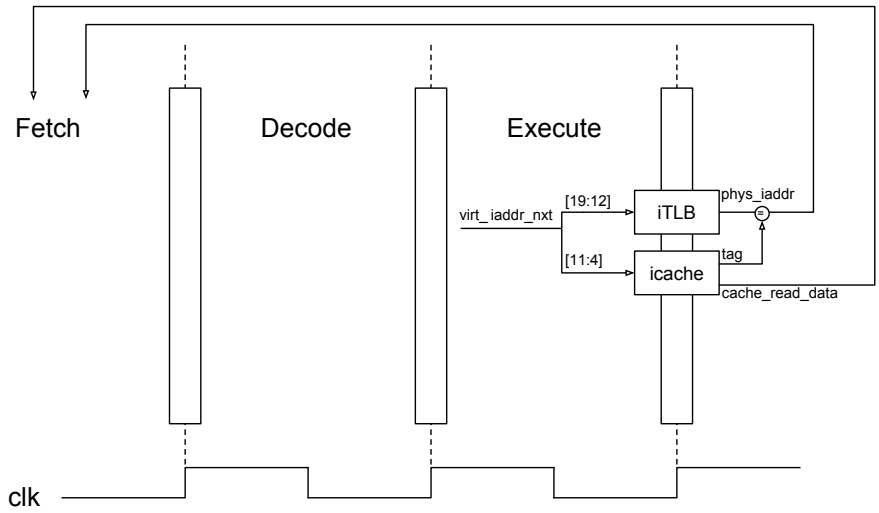

Figure 3.3: Instruction address flow in Vilma

Figure 3.3 shows what happens in Vilma. The pipeline remains the same, only with the addition of translation being performed concurrently with cache. The cache tag is compared with the physical address from the TLB to determine a hit. The prerequisite for determining a hit is that the physical address is available. As the TLB may also miss, cache can only hit if the TLB hits first. The TLB will also notice when the cache stalls the core and hold advancement for the duration of the stall, keeping the physical address intact. As long as the TLB hits, execution proceeds the same way as in Amber. However, when the TLB misses there is no physical address to compare the cache tag with. The TLB will then proceed to stall the core while performing a table walk, and release the stall once done. Because the stall preserves the processor state, nothing except for the TLB output will have changed by the time the stall is released. Next, tag comparison with the physical address determines if the cache hit, and normal execution follows.

There is one issue that arises in the case of a TLB miss. Because of the cache state machine being unaffected by stalls, it will read that the tag comparison is false and think that it missed. This triggers a memory access for a virtual address.

```
1 assign abort      = page_fault || domain_fault || permission_fault;
2 assign o_tlb_hit  = !i_mmu_enable ? 1'b1 : tlb_hit && !abort;
```

Listing 3.1: MMU abort conditional in tlb hit signal

```
1 assign idle_hit   = !i_tlb_hit ? 1'b1 : |data_hit_way;
2 assign read_miss  = enable && !idle_hit && !invalid_read;
```

Listing 3.2: Cache way hit conditional in read miss signal

Listing 3.1 and 3.2[2] shows the verilog code that prevents the caching of not only virtual addresses, but also addresses that cause page faults and access violations. The TLB hit output from the MMU is logically ANDed with the negated abort signal. Should either change, *o_tlb_hit* goes low. Also, when the MMU is not enabled, the effect of the signal is disabled by always being asserted. Cache will never think it missed a read as long as *idle_hit*[3] is asserted. Thus, by asserting *idle_hit* when *i_tlb_hit* goes low, cache is told to always assume true tag comparison for the duration of a TLB miss.

3.3.1 MMU Schematic

A simplified schematic of the MMU is shown in Figure 3.4. Trapezoid shapes represent multiplexers, where the control signal enters from the side. The Some details have been left out to save space and increase readability.

3.4 TLB Implementation

The TLB is driven by the state machine shown in Figure 3.5. Possible states are *idle*, *shootdown*, *read fld*, and *read sld*, where default state is idle. While idling the TLB serves buffered address translations in a single cycle. When a virtual address appears for which no valid translation exists in the buffer, the state machine activates translation table walk hardware to perform a table walk by entering the *read fld* state. The state machine moves to *read sld* when the fld is ready on the memory bus, or back to *idle* should the descriptor be invalid.

At any point an instruction to invalidate the TLB may be executed. Invalidating the entire TLB is commonly referred to as performing a TLB shootdown. Due to the MMU stalling the core whenever the TLB is not idling, the state machine will always be in *idle* when a shootdown is triggered. The shootdown procedure simply loops through every translation register and resets the valid bit to zero.

The actual buffer is a set of registers that store virtual to physical address mappings along with a valid bit and the corresponding page access permission bits. Two sets of registers called match registers and translation registers are used to store the information. Figure 3.6 shows the contents of each register.

[2]The signals are simplified for clarity and while they provide a correct interpretation they do not match exactly with the source.

[3]*idle_hit* is a local cache signal, not to be confused with *tlb_hit*.

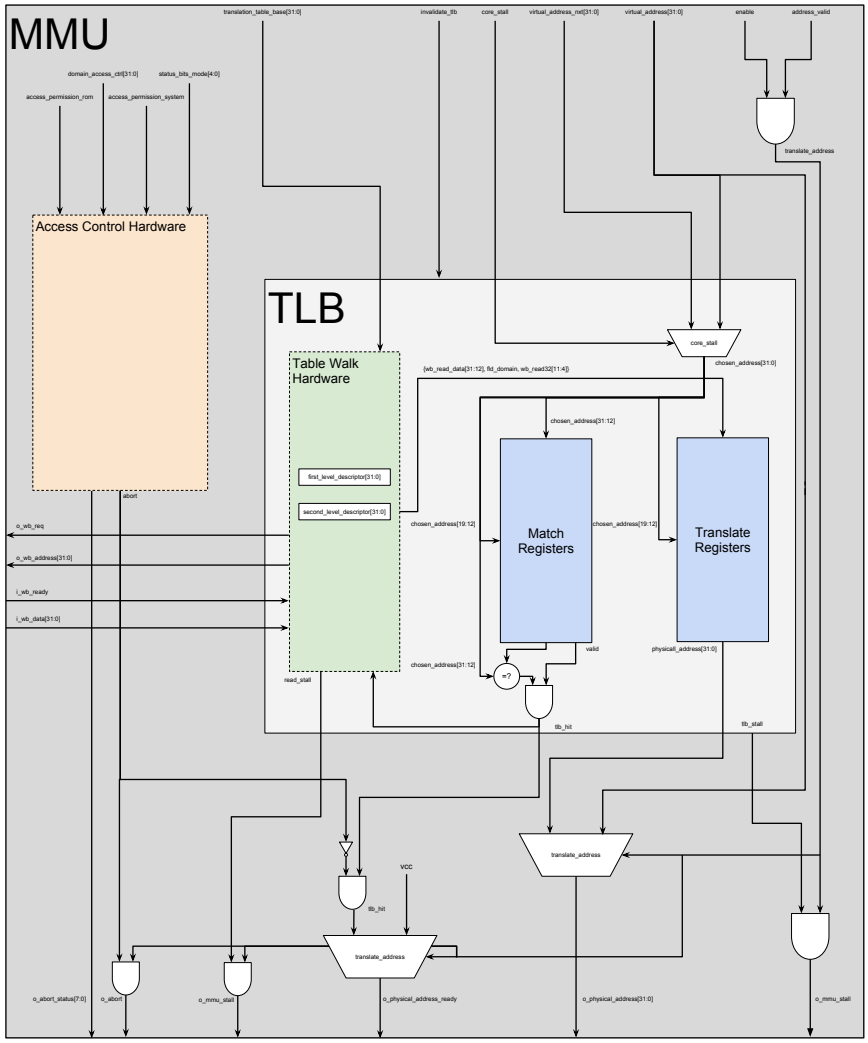

Figure 3.4: MMU schematic

The virtual page number (vpn) of a virtual address is formed by the 20 most significant bits of the word. Since Vilma indexes the translation and match registers by the 8 least significant bits of the virtual page number, the number of entries is $2^8 = 256$. Figure 3.7 shows how a TLB read is performed. The vpn in the match register is compared against the input vpn, where the result combined with the valid bit (v in the figure) determines the TLB hit signal. On a hit, the output of the translation register contains the physical page number (ppn). On a miss, the translation entry is

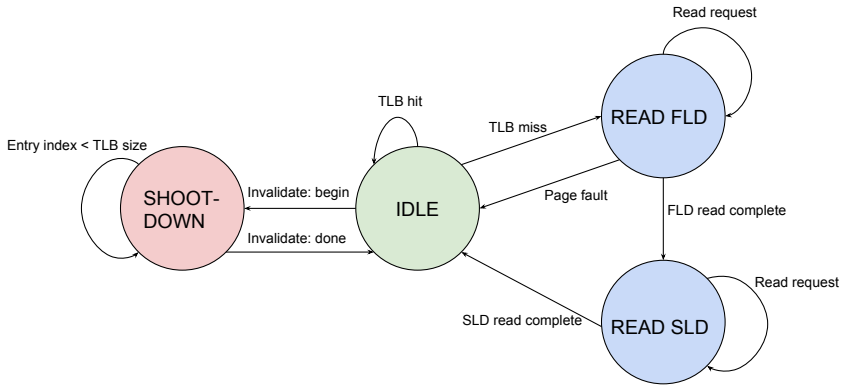

Figure 3.5: TLB state machine

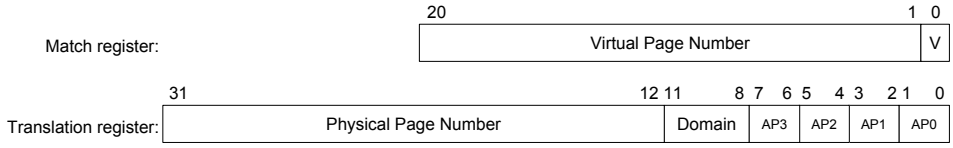

Figure 3.6: Buffer registers

looked up in the page table and written to the registers. A TLB write is indexed the same way, the only difference being that entries are written instead of read.

This way of indexing is simple and fast, but falls short on utilization. Whereas a fully associative TLB will fill up the whole buffer before replacing entries, a direct mapped TLB may have completely unused entries. In addition, a fully associative TLB is able to apply replacement strategies[4] to further increase performance. Such strategies are impossible to employ in a direct mapped buffer.

Because of the extra hardware used in fully associative TLBs, the buffers may contain only as few as 16 entries (see [19], page 503). Vilma attempts to make up for this shortcoming by increasing the number of entries to 256, and may go even higher. However, the number of entries should be measured against the frequency of TLB shootdowns, as shootdowns stall the pipeline for as many cycles as there are TLB

[4] A common TLB replacement strategy is least recently used.

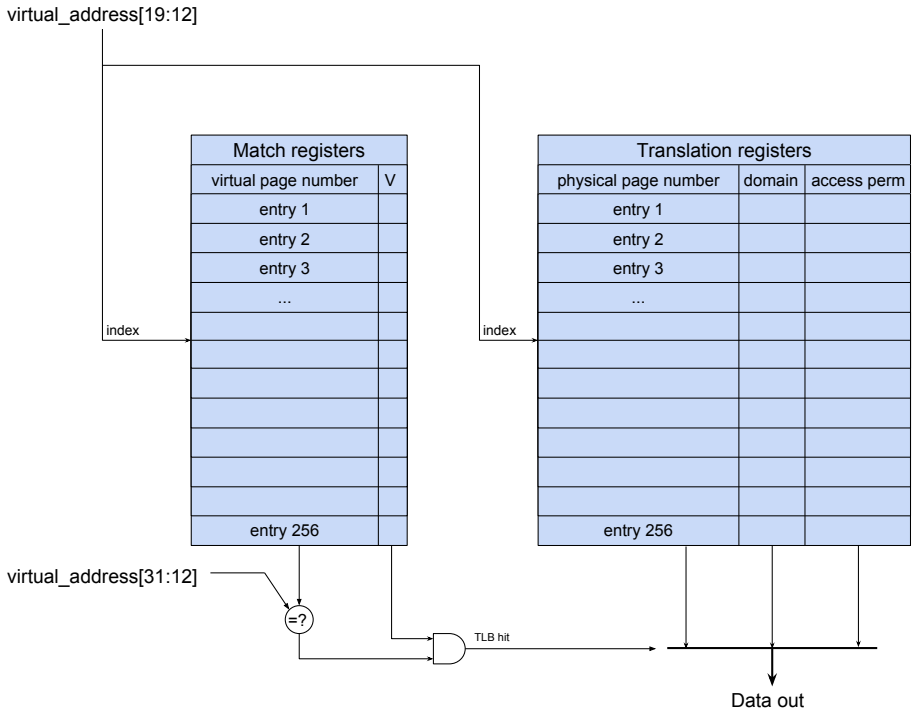

Figure 3.7: TLB read process

entries.

3.5 Translation Table Walk Hardware

When invoked, the translation table walk hardware executes the procedure described in Section 2.3.5. The hardware is contained within the TLB module, and controlled by the TLB state machine in states READ FLD and READ SLD shown in Figure 3.5. Vilma only supports small pages, and the use of any other data structure is undefined and may cause the system to malfunction.

3.6 Access Control Hardware

The access control hardware is kept within the MMU and determines whether a user process is allowed a requested operation on a given page. Vilma access control support is shown in table 3.1.

Priority	Sources		Supported
Highest	Terminal Exception		No
	Vector Exception		No
	Alignment		No
	External Abort on Translation	First level	No
		Second level	No
	Translation	Section	Yes
		Page	Yes
	Domain	Section	Yes
		Page	Yes
	Permission	Section	Yes
		Page	Yes
	External Abort on Linefetch	Section	No
		Page	No
Lowest	External Abort on Non-linefetch	Section	No
		Page	No

Table 3.1: Priority encoding of supported access control and faults in Vilma

3.7 Coprocessor

The only coprocessor present in Vilma is the system control coprocessor. The coprocessor[5] in Amber had only cache control, and was set up with a cache enable signal, a cache flush signal, and a set cacheable area range. The signals and their function have been preserved, but the target of activation has changed. This was done because the original implementation did not follow the arm specification for cache control and was using the System Control register erroneously as well as laying claim to MMU control registers. An example can be seen in Listing 3.3.

```
1  module a25_coprocessor
2  (
3  ...
4  output                      o_cache_flush,
5  );
6
7  ...
8  assign o_cache_flush      = copro15_reg1_write;
9  ...
10 // Flush the cache
11 assign copro15_reg1_write = !i_core_stall             &&
12                             i_copro_operation == 2'd2 &&
```

[5]Numeration is not used consequently, but *coprocessor* should always be read as *coprocessor number 15, or system control coprocessor.*

```
13                                  i_copro_crn       == 4'd1 ;
```

Listing 3.3: Cache flush control in Amber

If an *mcr* instruction occurred to coprocessor register 4'd1, the cache flush signal would be asserted.[6] Coprocessor 15 register 1 is the system control register. Flushing the cache should be done by register 7, and was changed appropriately. Although not irreparable, an unfortunate but necessary effect was that changing the cache control broke the Amber test suite wherever cache control was used.

Vilma has been equipped with a fully functioning system coprocessor that supports both MMU and cache control and functions in accordance with [6], a prerequisite for running an operating system that makes use of virtual memory on the core.

[6] There is also a bug in this line, as the signal is not sensitive to coprocessor 15 but rather any coprocessor write.

Chapter 4
Evaluation & Discussion

In this Chapter I present the results derived from performance testing and discuss how they verify core functionality. Specifically, I have measured FPGA-resource usage to evaluate the feasibility of the increased tile area occupied by the MMU, and core functionality has been measured using a test framework to prove that the core behaves correctly.

4.1 FPGA Resource Usage

With the introduction of new hardware, Vilma has increased resource usage of the Amber core. The larges components are the TLBs, and while the size of the TLB can be subject to change, the chosen size for synthesis has been 256. TLB size optimization for use in a tiled SHMAC environment is not part of this thesis and has been given little attention beyond verifying an acceptable level in comparison to the original Amber core.

The FPGA resources measured are logic slices and block RAM at the tile level, since tiles are the utilized units in a SHMAC environment. LUT usage is not listed separately as it is included in logic slices. The following tables provide an idea of how much space Vilma takes on an FPGA, and the scalability of the core with regard to the SHMAC tile layout.

Resource	Amber	Vilma	Increase
Block RAM	108 KB	252 KB	133 %
Logic slices	3709	4007	8 %

Table 4.1: Resource usage increase relative to Amber

Table 4.1 displays the resource usage of Vilma with respect to Amber. Table 4.2 shows how much space Vilma occupies on the Xilinx Virtex-5 XC5VLX330-ff1760-1 FPGA, the FPGA element used in SHMAC development. All values are collected

Resource	Vilma	Virtex5	Increase
Block RAM	252 KB	19,368 KB	1,3 %
Logic slices	4007	51840	7,7 %

Table 4.2: Vilma's resource utilization of a Virtex5

from design synthesis using Xilinx ISE 14.5, and the results in their entirety are listed in Appendix A.

Table 4.1 shows that Vilma uses a larger amount of block RAM than Amber, which is explained by the introduction of two TLBs. However, in light of available block RAM on the Virtex-5, the increase is less significant compared to the use of logic slices. Still, with an increase of only 8% in Logic slice usage, Vilma can be considered only marginally larger than Amber.

4.2 Verification

In this Section I present the methods used for design verification in the project, and the reasoning behind them.

The use of testbenches is common practice for digital design verification of hardware description languages like Verilog. By giving a set of inputs to the circuitry and comparing the output to the expected values, testbenches can be used to directly simulate signal values as they progress with regard to inputs and clocks. While offering great insight into system behavior, testbenches need to be defined at the most detailed level by directly setting bit values, and their usefulness depend solely on their design. As the set of possible input combinations may increase exponentially with system size, so may the development overhead for designing testbenches. Considering the size of the Amber core, time spent developing testbenches was considered to be rather counterproductive.

Alternatively, processor design may be verified by running real applications on the system. The Amber project comes with an extensive framework for simulating inputs from real assembly applications, and supplies a test suite containing 89 assembly programs. Its application in resent research [15][5] adds to the conclusion that this framework is able to sufficiently test and verify the correctness of the processor, especially considering there exists no alternative publicly available test suite.

In addition to running custom tests, booting an operating system on the processor would be a considerable achievement, and can be viewed as the ultimate testimony of functional correctness. Booting linux has been a goal throughout the development,

but as the project neared completion it became apparent there were bugs in the design that originated from other parts of the pipeline.

The entire test suite has been modified to run with MMU enabled, and simulation successfully completed on every test. The output can be found in Appendix **??**.

4.2.1 MMU table walk

Figure 4.1 shows the instruction MMU performing a table walk. The MMU stalls the core the moment it is enabled because of the TLB missing. Two wishbone requests follow to perform the table walk, and retrieves data in accordance with the data structure viewed in Listing 4.1.

```
1  init_mem:
2      ldr      r0, translation_table_base
3      nop
4      mcr      15, 0, r0, c2, c0, 0   @ set translation table base
5      mov      r0,  #4
6
7      ldr      r0, fld0_addr
8      ldr      r1, fld0_data
9      str      r1, [r0]                  @ put fld0 in memory
10     ldr      r0, sld0_addr
11     ldr      r1, sld0_data
12     str      r1, [r0]                  @ put sld0 in memory
13
14 /* Page table values */
15 translation_table_base:     .word 0x40200000
16 fld0_addr:                  .word 0x40200000
17 fld0_data:                  .word 0x40300001
18 sld0_addr:                  .word 0x40300000
19 sld0_data:                  .word 0x00000002
```

Listing 4.1: Address and data for fld and sld

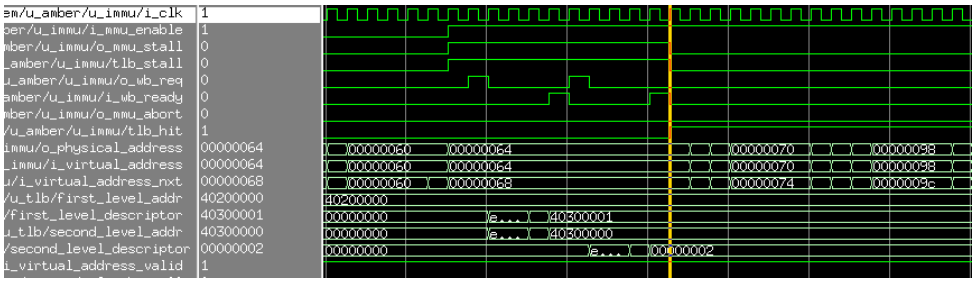

Figure 4.1: Instruction MMU performing a table walk

4.3 Bugs in the pipeline

Specifically, the bugs are associated with coprocessor instructions and their control of the MMU, and stem from load and store hazards. Two issues were discovered and are explained in their respective subsections. I believe the main reason these bugs have gone unnoticed so far is due to the lack of coprocessor usage. Since the coprocessor was only used to enable caching in the old Amber design, these hazards had no complication on system performance.

4.3.1 Load hazard

A load instruction (ldr) to a specific register, followed by a coprocessor store (mcr) of the same register will fail unless a cycle of no operation (nop) appears in between. This happens because the load takes two cycles to complete, which is not accounted for followed by a coprocessor store, the effect of which is that the coprocessor receives the wrong contents.

```
1 init_mem:
2     mov     r0,  #4                     @ r0 = 4
3     ldr     r0, translation_table_base  @ load base into r0
4     mcr     15, 0, r0, c2, c0, 0        @ set cp15 table base
```

Listing 4.2: load r0, store r0 to coprocessor

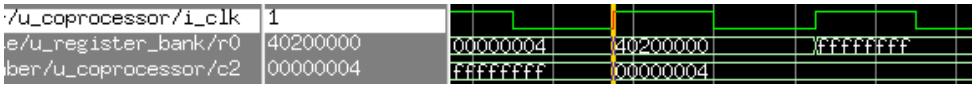

Figure 4.2: ldr hazard

The load hazards was discovered while setting the translation table base register. Listing 4.2 shows the code and Figure 4.2 displays what happens in simulation. The first line of code sets the contents of register r0 to 4. Next, the *ldr* operation sets r0 to the value of *tlb_l1_base* (0x40200000), which is the table base to be loaded into coprocessor register c2. However, by the time r0 is updated the mcr instruction is already being executed, and so the value of the translation table register becomes 0x4 instead of 0x40200000 like it should be. An operating system will thus be unable to update the translation table base correctly without modifying the source code for the memory handling routine.

4.3.2 Store hazard

In the Amber pipeline it takes three cycles from an instruction is created until it is executed; fetch, decode, execute. This means that during normal operation we

should expect to count three cycles from an address is created until it is executed. However, by the time a move from register to coprocessor (mcr) instruction has been issued to enable the MMU, to the point where the enable signal is asserted, the subsequent instruction has already been executed.

```
1 enable_mmu:
2     orr     r0, r0, #0x1            @ set MMU enable bit
3     orr     r0, r0, #0x100          @ set SYSTEM access permission
4     mcr     p15, 0, r0, c1, c0, 0   @ turn on
```

Listing 4.3: load r0, store r0 to coprocessor

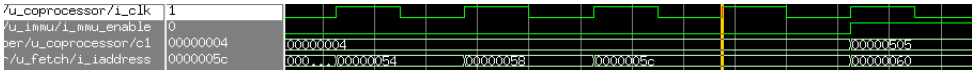

Figure 4.3: mcr hazard

The store hazards was discovered while enabling the MMU. Listing 4.3 shows the code and Figure 4.3 displays what happens in simulation. Due to stalls in the pipeline from asserting the MMU enable signal, it is impossible to get the whole picture into the waveform. As such it is challenging to properly explain the issue so I will proceed to state the expected behavior. The MMU enable signal should be asserted one cycle ealier, by the yellow marker if Figure 4.3. Current behavior results in the subsequent address (0x00000060) not being translated, as an operating system would expect. The result is the execution of a virtual address and will most likely cause the system to misbehave, if not crash.

4.4 Discussion

Due to bugs in the pipeline, testing address translation was only done by building page tables in assembly and running the test suite. As the programs in the test suite are not sufficiently large to exceed a page, and since addresses start at 0x0 with consecutive distribution not exceeding 0x000000FF, the instruction MMU needed only one page table entry mapping 0x0 to 0x0. This allowed simulation of the instruction MMU, but proved more challenging for the data MMU.

As opposed to instructions, data will be spaced at seemingly random places in the address space by the compiler. Thus, building page tables in assembly that contain accurate mappings would have been guess work at best, and, since compilation would change it up regardless, was concluded impossible without a proper memory handler, like that present in an OS. Without proper translations, the data MMU caused all tests to fail. Concluding that digging through OS source code would not yield much

other than wasted time, and that building one in assembly was similarly infeasible, the data MMU output was unhooked during testing. As such, only the instruction MMU was enabled to produce test results. Still, it should be noted that while the Data MMU did not provide translations to the hardware, it attempted to and in turn performed table walks that had an impact on performance. Consequently, the effect of the data MMU is visible on performance output.

Due to these circumstances, only the instruction MMU has been tested, and since the mapping was essentially physical to physical (0x0 to 0x0), a real virtual address space has not been in place. However, I would like to argue that without properly working hardware, the TLB entry for 0x0 would never been accessed. The MMU in Vilma does a complete page table walk in compliance with the ARM Architecture Reference Manual [6], and successfully locates the mapping and loads it into the TLB. The TLB in turn supplies this translation without delay, resulting in the entire design to be transparent to performance except for the initial table walk. In addition, the access control hardware have also been tested with success, addressing page faults and access permissions. Moreover, because both MMUs are instances of the same module, verifying one also verifies the other, as behavior in terms of page table walks and address translation is exactly the same.

Complete verification will not be available until the necessary debugging has been done on the rest of the system to enable running an OS.

Chapter 5
Conclusions and Further Work

In this Chapter I discuss the solution I have presented of the assignment. The first section recaps at the list of requirements presented in Chapter 1 and elaborates on the challenges faced and how the requirement was met. The second section discusses areas for further work.

5.1 Assignment Requirements

The following is a re-listing of the requirements from Section 1.3.

Address translation Vilma features a hardware MMU capable of doing concurrent address translation.

Resource balanced Vilma has been verified to occupy close to the same area as that of the original Amber, and does not excessively drain any particular FPGA-resources.

Portable The MMU in Vilma is in large part contained within one module and only directly impacts caches.

Verified Vilma has successfully passed verification testing.

5.1.1 Address translation

Vilma should integrate a seamless address translation in hardware and work as expected according to the ARM Architecture Reference Manual. Correct functionality is vital for reusing architecture specific code, like that of an OS.

To satisfy this requirement I have written a test that enables the MMU and performs some computations, and then tests the output. If the translations are incorrect, the program will deviate from expected behavior and fail.

Since Vilma successfully passes the test, and because the page table in the test is built in accordance with the ARM Architecture Reference Manual [6], I conclude that both aspects of the requirement is met and that Vilma applies correct translation. The only weakness with the test is that a proper virtual address space is lacking, but I would like to underline that this is only possible to if the rest of the core behaves as expected.

5.1.2 Resource balanced

Vilma should not excessively drain any particular FPGA-resources. The importance of this requirement is its use in the SHMAC environment, where the number of cores in a setup is limited by the exhaustion of the most critical resource. As Vilma drains close to the same area as Amber when compared to the available resources on the Virtex-5 FPGA, I conclude that this requirement has been fulfilled.

5.1.3 Portable

To introducing virtual memory to the SHMAC environment, the virtual address space has to be used across a complete tile setup. Thus, when creating an MMU for SHMAC processing tiles, care should be taken to avoid any significant dependencies on the surrounding hardware.

For this requirement I would like to argue that the simplicity of the design with few inputs and outputs and overall structuring, suffices to meet this requirement for most ARMv4T compliant architectures.

5.1.4 Verified

Vilma should be verified to ensure the correctness correctness of the design. The presence of an MMU that changes or breaks expected system behavior is worse than having none at all, and so correct behavior is vital for success.

Completely meeting this requirement has proved impossible due to the hazards discussed in Section 4.3. Without having tested running an Operating System on the core, verification suffers. In addition, due to how the compiler spaces data in the address space, the data MMU does not have correct page tables. This is the unfortunate consequence of bugs appearing outside of the area of focus. Still, the test suite that comes with Amber serves to give a strong indication that the core functions as expected for those applications.

5.2 Further work

In this section I suggest what can be done to improve Vilma.

5.2.1 Investigate coprocessor bugs

The bugs discussed in Section 4.3 inhibit running an OS on Vilma, and must be resolved before virtual memory can be utilized. Significant effort should be made to evaluate how coprocessor instructions are executed in order to assure correct system behavior.

5.2.2 Increase TLB associativity

FPGA resources can be utilized more efficiently by increasing the associativity of the TLB. Preferably a fully associative implementation should be implemented, which will reduce miss percentage by maximizing the use of TLB entries, while at the same time allowing the size to be reduced with minimal penalty.

5.2.3 Support entry invalidation

Vilma only supports complete TBL shootdowns. Supporting entry invalidation can significantly decrease the performance penalty by invalidating single entries.

5.2.4 Support other data structures

As stated in Section 3.1, Vilma supports only small pages. If the use of other data structures is desirable it will need hardware support. In its current state, only the TLB module of Vilma will have to be changed to reach compliance, and behavior must be in accordance with the ARM Architecture Reference Manual [6].

5.2.5 Make Caches Virtually Indexed, Physically Tagged

If cache size is increased, cache indexing will begin to use bits of the virtual page number. This has implications on the design as explained in [9] [10], and must be addressed accordingly for virtual memory to function.

Bibliography

[1] Amber ARM-compatible core, OpenCores. http://opencores.org/project,amber. Accessed December 2014.

[2] Long Range Computation Study Group, MIT. https://stacks.stanford.edu/file/druid:th919jh6519/sc0524_1995-247_b27_f30.pdf. Accessed Jan 2015.

[3] SoC Interconnection: Wishbone, OpenCores. http://opencores.org/opencores,wishbone. Accessed December 2014.

[4] The Single-ISA Heterogeneous MAny-core Computer (SHMAC). http://www.ntnu.edu/ime/eecs/shmac. Accessed May 2014.

[5] Sebastian Akre, Anders Tvetmarken; Bøe. Trubo amber: A high-performance processor core for shmac. Master's thesis, Norwegian University of Science and Technology, 2014.

[6] ARM. *ARM Architecture Reference Manual*, 2000. ARM DDI 0100E, Accessed December 2014.

[7] Lustig D. Martonosi M. Bhattacharjee, A. Shared last-level tlbs for chip multiprocessors. *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 62–63, February 2011.

[8] Mark Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. January 2007. Solid-State Circuits Society.

[9] Dubois M. Chekleov M. Virtual-address caches. part 1: problems and solutions in uniprocessors. *Micro, IEEE*, 17(5):64–71, Sep/Oct 1997.

[10] Dubois M. Chekleov M. Virtual-address caches.2. multiprocessor issues. *Micro, IEEE*, 17(6):69–74, Nov/Dec 1997.

[11] Yu H. Rideout L. Bassous E. Leblanc A. Dennard R., Gaensslen F. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. October 1974. IEEE Journal of Solid State Circuits SC–9 (5).

[12] Peter J. Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2:153–189, September 1970.

[13] J. Fotheringham. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *ACM*, 4:435–436, October 1961.

[14] Renée St. Amantx Karthikeyan Sankaralingamz Hadi Esmaeilzadehy, Emily Blemz and Doug Burger. Dark Silicon and the End of Multicore Scaling. 2011. Appears in the Proceedings of the 38th International Symposium on Computer Architecture (ISCA '11).

[15] Håkon Amundsen Joakim Andersson. Linux for shmac. Master's thesis, Norwegian University of Science and Technology, 2014.

[16] Edwards D.B.G. Lanigan M.J. Sumner F.H. Kilburn, T. One-level storage system. *Electronic Computers*, EC-11:223–235, April 1962.

[17] Y.K. ; Verma N. ; Chandrakasan A.P. Kwong, J.; Ramadass. A 65 nm sub- $v_t microcontroller with integrated sram and switched capacitor dc - dc converter. Solid - State Circuits, 44 : 115 - -126, Jan 2009$.

[18] Gordon E. Moore. Cramming more components onto integrated circuits. 1965. Electronics Magazine. p. 4. Retrieved 2006-11-11.

[19] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware/Software Interface*. Morgan Kaufmann, 4 edition, November 2011.

[20] David A. Patterson and John L. Hennessy. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 5 edition, 2012. Appendix B.4 - Virtual Memory.

[21] R. Rojas. Konrad zuse's legacy: the architecture of the z1 and z3. *Annals of the History of Computing, IEEE*, 19:5–16, April 1997.

[22] Lebeck Alvin R. Sorin Daniel J. Bracy A. Romanescu, Bogdan F. Unified instruction/-translation/data (unitd) coherence: One protocol to rule them all. *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, January 2010.

Chapter A
Synthesis Report

A.1 Vilma Xilinx Mapping Report File

```
Release 14.5 Map P.58f (lin64)
Xilinx Mapping Report File for Design 'vilma_wrapper'

Design Information
------------------
Command Line   : map -intstyle ise -p xc5vlx330-ff1760-1 -w -logic_opt off -ol
high -t 1 -register_duplication off -global_opt off -mt off -cm area -ir off -pr
off -lc off -power off -o a25_core_map.ncd a25_core.ngd a25_core.pcf
Target Device  : xc5vlx330
Target Package : ff1760
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.55 $
Mapped Date    : Mon Jan 19 15:20:44 2015

Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                 3,842 out of 207,360    1%
    Number used as Flip Flops:               3,841
    Number used as Latch-thrus:                  1
  Number of Slice LUTs:                     11,058 out of 207,360    5%
    Number used as logic:                    9,000 out of 207,360    4%
      Number using O6 output only:           8,284
      Number using O5 output only:              88
      Number using O5 and O6:                  628
```

```
    Number used as Memory:                  2,048 out of  54,720    3%
      Number used as Single Port RAM:       2,048
        Number using O6 output only:        2,048
    Number used as exclusive route-thru:       10
  Number of route-thrus:                      125
    Number using O6 output only:               94
    Number using O5 output only:               30
    Number using O5 and O6:                     1

Slice Logic Distribution:
  Number of occupied Slices:                4,007 out of  51,840    7%
  Number of LUT Flip Flop pairs used:      11,807
    Number with an unused Flip Flop:        7,965 out of  11,807   67%
    Number with an unused LUT:                749 out of  11,807    6%
    Number of fully used LUT-FF pairs:      3,093 out of  11,807   26%
    Number of unique control sets:            142
    Number of slice register sites lost
      to control set restrictions:            215 out of 207,360    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                      314 out of   1,200   26%
    IOB Flip Flops:                           182

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                    7 out of     288    2%
    Number using BlockRAM only:               7
    Total primitives used:
      Number of 36k BlockRAM used:            7
    Total Memory used (KB):                 252 out of  10,368    2%
  Number of BUFG/BUFGCTRLs:                   2 out of      32    6%
    Number used as BUFGs:                     2

Average Fanout of Non-Clock Nets:             5.68
```

A.2 Amber Xilinx Mapping Report File

```
Release 14.5 Map P.58f (lin64)
Xilinx Mapping Report File for Design 'amber_wrapper'

Design Information
------------------
Command Line   : map -intstyle ise -p xc5vlx330-ff1760-1 -w -logic_opt off -ol
high -t 1 -register_duplication off -global_opt off -mt off -cm area -ir off -pr
off -lc off -power off -o a25_core_map.ncd a25_core.ngd a25_core.pcf
Target Device  : xc5vlx330
Target Package : ff1760
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.55 $
Mapped Date    : Mon Jan 19 15:01:17 2015

Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                 3,617 out of 207,360    1%
    Number used as Flip Flops:               3,616
    Number used as Latch-thrus:                  1
  Number of Slice LUTs:                     10,637 out of 207,360    5%
    Number used as logic:                    8,580 out of 207,360    4%
      Number using O6 output only:           7,891
      Number using O5 output only:              74
      Number using O5 and O6:                  615
    Number used as Memory:                   2,048 out of  54,720    3%
      Number used as Single Port RAM:        2,048
        Number using O6 output only:         2,048
    Number used as exclusive route-thru:         9
  Number of route-thrus:                        92
    Number using O6 output only:                79
    Number using O5 output only:                12
    Number using O5 and O6:                      1

Slice Logic Distribution:
  Number of occupied Slices:                 3,709 out of  51,840    7%
  Number of LUT Flip Flop pairs used:       11,386
    Number with an unused Flip Flop:         7,769 out of  11,386   68%
```

```
    Number with an unused LUT:                749 out of  11,386    6%
    Number of fully used LUT-FF pairs:      2,868 out of  11,386   25%
    Number of unique control sets:            127
    Number of slice register sites lost
      to control set restrictions:            192 out of 207,360    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                    314 out of   1,200   26%
    IOB Flip Flops:                         182

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                    3 out of     288    1%
    Number using BlockRAM only:               3
    Total primitives used:
      Number of 36k BlockRAM used:            3
    Total Memory used (KB):                 108 out of  10,368    1%
  Number of BUFG/BUFGCTRLs:                   1 out of      32    3%
    Number used as BUFGs:                     1

Average Fanout of Non-Clock Nets:             5.80
```