NTNU
Innovation and Creativity

FPGA Framework for CMP

Kenneth Østby

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Morten Hartmann, IDI
Co-supervisor: Marius Grannæs, IDI

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

Problem Description

The project's main goal is to develop a platform supporting experiments of Chip Multiprocessors on a Field Programmable Gate Array. The NCAR Group performs research on the simulation of architectural variations in Chip Multiprocessors. It's desirable to extend the experiments to run on parametrizable architectures designed for Field Programmable Gate Arrays. The solution must account for several processors and their interconnection, cache structures, and if time permits operating systems and the ability to communicate with existing simulators. Knowledge about FPGA, VHDL and the C programming language is required.

The subtasks are (to the extent time allows them to be addressed):

Background research on related projects
- Multicore solutions in FPGAs, with emphasis on cache solutions
- potential operating systems

Research on cache solutions on an FPGA
- assess the option of customizable solutions and automated synthesis of cache systems

Make several cores work on existing lab equipment in a simple testing environment

Research and identify suitable cores for the system and possibly an operating system

Cache design with emphasis on customizability and suitable cache hierarchy (L1/L2) on available FPGA cards (Nalle)

Investigate possibilities towards integration with simulation environments (M5 and/or SimpleScalar) and experiments at NCAR

Run suitable experiments and analyse results

Assignment given: 20. January 2007
Supervisor: Morten Hartmann, IDI

Abstract

The single core processor stagnated due to four major factors. (1) The lack of instruction level parallelism to exploit, (2) increased power consumption, (3) complexity involved in designing a modern processor, and (4) the performance gap between memory and the processor. As the gate size has decreased, a natural solution has been to introduce several cores on the same die, creating a chip multicore processor.

However, the introduction of chip multicore processors has brought a new set of new challenges such as power consumptions and cache strategies. Although throughly researched in context of super computers, the chip multiprocessor has decreased in physical size, and thus some of the old paradigms should be reevaluated, and new paradigms found.

To be able to research, simulate and experiment on new multicore architectures, simulators and methods of prototyping are needed by the community, and has traditionally been done by software simulators. To help decrease the time between results, and increase the productivity a hardware based method of prototyping is needed.

This thesis contributes by presenting a novel multicore architecture with interchangeable and easily customizable units allowing the developers to extend the architecture, rewriting only the subsystem in question. The architecture is implemented in VHDL and has been tested on a Virtex FPGA, utilizing the MicroBlaze microcontroller. Based upon FPGA technologies, the platform is close in nature to that of a chip multiprocessor. The thesis also shows that a hardware based environment will significantly decrease the time to results.

Preface

This Master's Thesis was written as a part of the degree as "Sivilingeniør" in Computer Engineering. The Master's Thesis is founded in an earlier project in the subject TDT4720 – Computer Design and Architecture. The goal of TDT4720 was to give the student an introduction to the current state of Computer Architecture, while introducing him to the tools used in hardware construction.

The work is done for the Norwegian University of Science and Technology (NTNU) at the Faculty of Information Technology, Mathematics and Electrical Engineering, and the Department of Computer and Information Science. The group which hosted the project was the NTNU Computer Architecture and Design Group under the supervision of Associate Professor Morten Hartmann.

The advisor for this thesis was Associate Professor Morten Hartmann. Co-advisor was Research Fellow Marius Grannæs.

Kenneth Østby
June 17, 2007

Acknowledgments

I would like to thank the following persons for their support and input throughout this project. Associate Professor Morten Hartmann for being my advisor, and thus allowing me to partake in this journey through Chip Multiprocessors. Marius Grannæs for being my co-advisor, reading through my thesis several times and always available for technical discussions.

I would also like to thank the people at NTNU Computer Architecture Research Group for including me in their much interesting meetings. Also important are the people at my study room, ITV-458, May Linda Martinsen, Knut Imar Hagen, Christian Larsen, Idar Borlaug, Rolf Anders Syvertsen and Jan Peder David-Andersen.

Finally I would like to send my thanks and thoughts to all of my friends and loved ones not mentioned by name.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

During the last couple of years, the traditional single core monolithic CPU has failed to further increase its performance proportionally to the decrease in transistor size, as the popular although not correct interpretation of Moore's Law[34] implies. This is, according to Dr. Patterson in his President's Letter[30], due to 3 main factors. (1) Power dissipation, (2) lack of Instruction Level Parallelism to exploit and (3) the long known memory gap which has steadily grown since the beginning of computer science. A fourth factor mentioned by Spracklen et al.[38] is the inherent complexity of designing a processor. A high performance single core CPU requires vast amounts of chip resources to implement the control logic, ensuring that operations don't interferes with each others. This leaves less room for implementing the computational logic, which in turn influences the overall performance of the processor. The Chip multiprocessor (CMP) tries to solve these problems by utilizing several cores inside a single processor. Having several cores on a single chip introduces several new problems, some which have been encountered before in the world of super computing and others new. This includes handling cache in an attempt to reduce the off-chip access, different topologies to allow inter-communication between on-chip processing elements and finally the power dissipation.

The memory gap, as shown in figure 1.1, has long troubled the computer engineers and has been the subject of several research projects. This is also one of the major focal points when researching CMP architectures. The reasons why the memory gap has appeared is a product of several factors. First there is the sheer distance the data signals must travel between the processor and the memory. Data on-chip have a shorter path to travel, and thus have a noticeable decrease in latency compared to off-chip access. Second, the technology and the larger size of memory that exists outside the processor adds to the latency by requiring more time to deliver the requested data onto the bus itself. All of these factors makes it important to limit redundant memory accesses, storing the frequently used

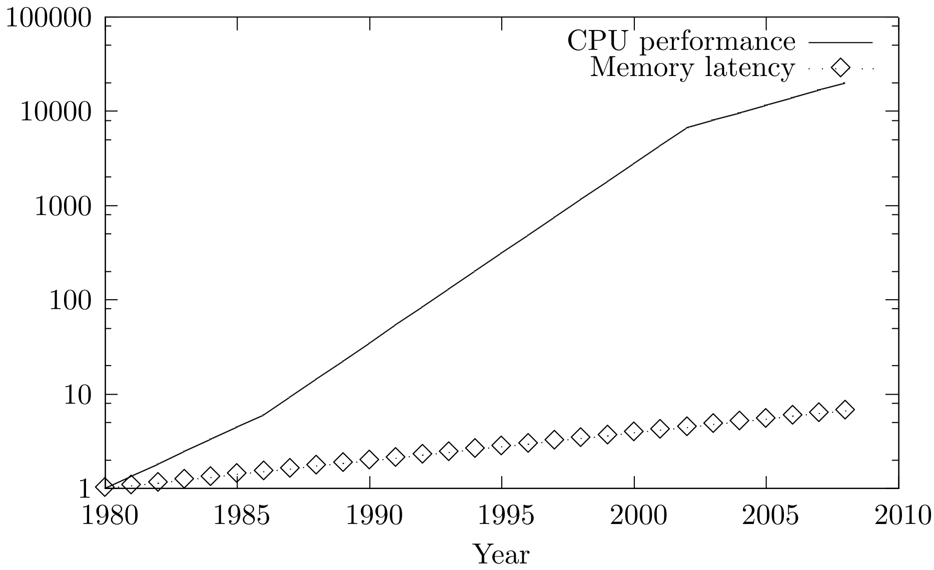

Figure 1.1: CPU/Memory performance[30].
Logarithmic plot for readability.

data in cache banks on the processor, keeping recently used data close in locality. This is done by employing different strategies which forces the processor and the cache banks to cooperate, trying not to evict the frequently used lines of data off the chip. This in an effort to reduce memory access outside the processor itself. Besides reducing the off-chip access much of the cache research studies ways of keeping data requested by the different cores near in locality close to the core which is most likely to request it. This might involve duplication and spreading cache lines around internally on the chip.

Another reason which forced the change of paradigm from single core processors to multi-core processors is the increased difficulty of exploiting Instruction Level Parallelism(ILP). Exploiting ILP is to exploit the fact that certain combinations of instructions can be re-arranged without changing the outcome. This is done to avoid stalls in the processor's pipeline, and hence reduce the time spent by the CPU idling. The main challenge exploiting ILP is that it gets exponentially harder per fraction of parallelism level, and thus stagnating at a certain level[15]. To achieve a perfect level of instruction parallelism, unrealisable hardware much be constructed. This to account for perfect branch prediction, unlimited access of registers and all the memory locations must be known beforehand in order to rearrange load and store operations[15]. All needed to achieve perfect ILP. Seeing how the work load on most of the commercial servers are not focused on computationally intensive tasks, but supporting several concurrent requests have forced forth a change where Thread Level Parallelism(TLP) has gained focus. By adding several cores true TLP can be achieved by running the different threads on different cores.

The power consumption of the computer processor has always been the subject of inquiries, although it has not been the major point of focus in desktop computers. Here performance has been the main focal point. However, as the frequency using traditional CMOS technology reaches its practical limit, ways of reducing the power dissipation have gained popularity. The CMP contributes mainly in two ways to help solve this problem. The first is due to the nature of cubic dependency between the operating frequency of the processor and its power dissipation as presented by Jerraya et al.[18] and Gochman et al.[13]. The practical implication of this means that by halving the frequency of a single core, the power consumption of a single core will be reduced to a mere quarter of its original use. Then by doubling the existing cores on a chip with half the frequency it is possible to retain the same performance with less dissipation of power. This is though a simplified version of the power usage. When calculating the overall consumption on a chip, the mechanisms for communication between different processing elements, cache and etcetera, must be taken into consideration. As shown by Kumar et al.[21] , pending on the surrounding infrastructure, it is not only the cores themselves that dissipates power on a modern processor. Another interesting capability that comes with the inherit modularity of a CMP is the ease of resource management. If the load on a CMP is low, cores that are idling can be disabled to reduce the total power consumption. The ability to scale down the number of operational cores, and halving the total power dissipation are both important steps in battling the consumption of power in modern processors.

The final point which is mentioned by Spracklen et al.[38] and Olukotun and Hammond[27] is the inherit complexity of designing a modern processor and its cores. A high performance single core processor requires serious amount of effort into ensuring safe computations and a coherent environment. Since the cores employed by CMP's can afford a reduction in performance per each individual core and still have the same overall throughput, a single CMP core can be simplified compared to a core in a traditional single core processor. Also by allowing a scaled down version of the cores simplifies the development process, and thus reducing the time to marked and hence decreasing the production cost. Beside the purely economic reason, a simplified model decreases the chance of bugs in the core itself. Hence when gate size decreases, its performance can be improved by increasing the number of cores on-chip.

To help developers address new architectural challenges, software based simulator such as SimpleScalar[4] and M5[3] have been used. Here the developer specifies the different components on the processor and their behaviour. The problems with software based simulators are that they are not completely accurate in the sense that it has to sacrifice accuracy in some field to gain in another[16]. E.g., a simulator which perspires to be instruction set accurate will will not attempt to claim timing accuracy[16]. Another problem is that simulating an entire processor is a slow process, taking quite some time to produce results. The time it takes to produce results could be reduced by decreasing the level of details as proposed by Hines et al.[16], but this would lead to less accurate simulations. Also, by having to manually specify the different levels of detail in the model leaves room for inconsistencies between the different run levels. This might produce different results depending on the detail level when simulating.

The Field Programmable Gate Array(FPGA) is a natural evolution of the Programmable Logic Arrays which allows the developer to program the behavior of the logic hardware.

The extension which made the FPGA popular to prototyping hardware is its Field-Programmable attribute. Unlike many implementations of the Programmable Logic Array, the behaviour of the FPGA is fully reprogrammable. In modern FPGA's this is solved by connecting each programmable unit, i.e., its logic elements and routing resources, to the corresponding bit in memory. Then by changing the bit-stream the behaviour of the FPGA changes. An example would be a mathematical unit, the Arithmetic Logic Unit(ALU). Pending on its corresponding bit, the ALU might work as either an multiplier, divisor or a different mathematical operation. Then by connecting several logic elements it's possible to get the desired behaviour. The major drawback of FPGA's is that due to their flexible nature, they cannot be have the same high clock frequency as a dedicated chip, Application Specific Integrated Circuit. The advantage is that the calculations which the software based simulator must calculate serially per simulated cycle, will happen in true parallel on a FPGA. This means that the reprogrammable hardware will have an increase in performance compared to an software based simulator.

To be able to prototype new multicore architectures and architectural parameters trying to improve CMP performance, a platform which allows for rapid changes is needed. It is also important that the time to produce results is reduced to increase productivity. To help decrease the period between a new architecture or parameter is decided upon, and until the result is known, a flexible FPGA platform is used. To achieve the flexibility in hardware, an architecture must be modular to allow for interchangeable elements. This to prevent time demanding and challenging rewrites of the entire system.

1.2 Problem Description

One of the new trends in computer architecture to battle the performance wall by introducing several cores on the same chip. By doing so, the computer engineers were put in front of old challenges in a new setting. Techniques which were meant for large cluster suddenly had to be scaled into a single die and novel ideas was needed. Seeing how the cost in terms of money and time imprinting the first chip is to high to allow prototyping, simulators and logic analyzers have been the mainstream technology of prototyping processors. However software based simulators such as SimpleScalar[4] are notoriously slow, and in nature not able to fully accurate simulate the inner workings of a CPU[16].

The goal of the Master's Thesis is to investigate and develop a hardware based platform for testing Chip Multiprocessors using Field Programmable Gate Arrays, figure 1.2, as a alternative to the software based simulator. The platform should allow for change of architectural parameters and components.

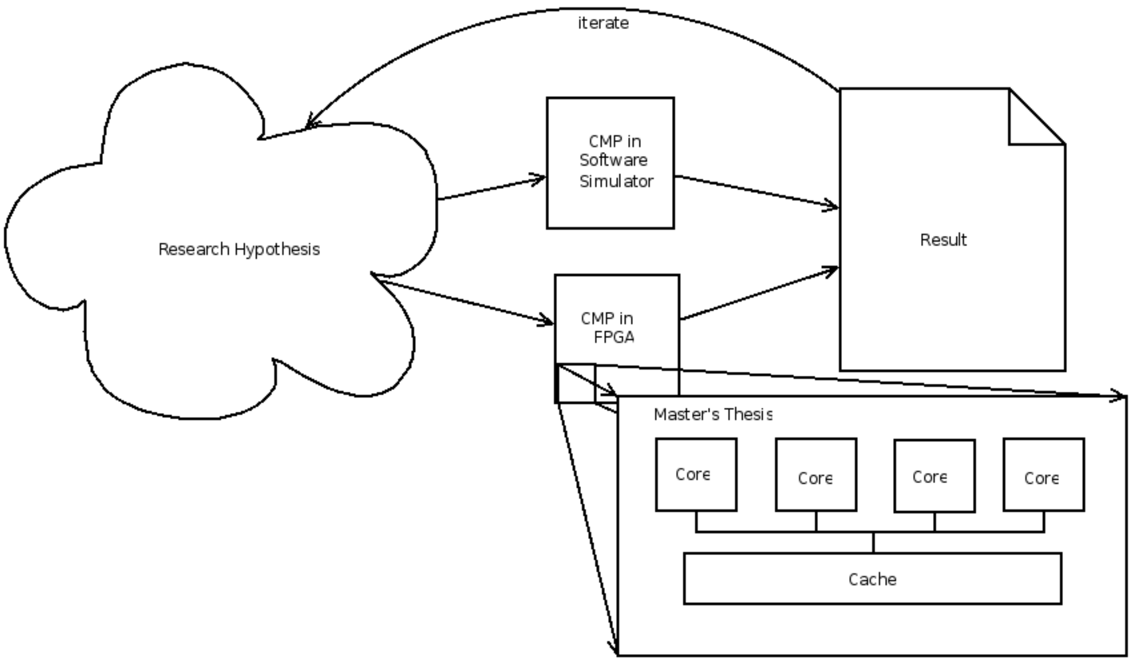

Figure 1.2: Project Overview

1.3 Project Motivation

The NTNU Computer Architecture Research Group has focused their research on Chip Multiprocessor Challenges[25]. More specific, the research group have focused their attention on intercommunication, caches, pre-fetching and scheduling. At the time this thesis was written, their main method of prototyping multicore architectures was based upon software simulators such as SimpleScalar and M5.

Since the software based simulations used to much time in producing results, the NCAR group wanted an supplementary method of prototyping their proposed multicore architectures. They also wanted a platform through which they could rapidly change the architectural parameters in order to test several configurations. This without changing the underlying behavioural code.

1.4 Contribution

Through his work with the Master's Thesis the author has contributed with the following set of items. A throughly background search has been done in the field of Chip Multiprocessors. The author has contributed by presenting a hardware based platform for prototyping novel computer architectures. Through the hardware platform the author has contributed by allowing fellow researchers to produce results more accurate, and decreased the time waiting for results.

1.5 Scope

This thesis will describe a multicore architecture and the methodology used. The thesis will present relevant research and background information needed to understand chip multicore architecture. It will not describe in detail the inner workings of processor core technology beyond what's needed to implement it in a multicore environment.

1.6 Outline

In chapter 2, the thesis will present the relevant background information for reading this paper. It will start by an displaying the relevant work, before it will continue by introducing the central concepts in computer architecture relevant to chip multiprocessors. Chapter 2 will end with a presentation of the tools used.

Chapter 3 presents the methodology, where the hardware architecture is described. Chapter 3 will end by a presentation of the controlling software and the application used for testing the hardware design.

In chapter 4 the thesis will show the result from the benchmark suite introduced in chapter 3, before the results will be further discussed in chapter 5.

Finally the thesis will conclude with chapter 6 with the conclusion and further work.

1.6.1 Appendices

Appendix A contains the benchmarks created for the processor. Appendix A also contains the makefile and linker script required in order of building the binary application.

Appendix B contains the hardware developed for this project, with the toplevel configurations in B.9, B.8 and B.7. Besides the code found outside the architecture section in appendix B.1 and B.2, which is glue code code generated by the Xilinx tools, everything has been written by the author. Some bugs has also been fixed in the glue code, as discussed in section 5.4.2.

Appendix C contains the software developed by the author to control the FPGA and act as main memory as described in section 3.6.

Chapter 2

Background

This chapter will discuss different architectural challenges designing a CMP, and present some of the proposed solutions. It will introduce the reader to the field of processor architecture and FPGA's by first presenting the already existing work before introducing central concepts, and finally the tools used to develop the FPGA framework.

2.1 Related Work

One of the central problems designing new hardware architectures is to be able to test the architecture before sending it to production. Being able to both test the hardware and software before production, is of grave importance to create a stable and optimal platform. Traditionally the software developers have lagged behind the hardware manufacturers due to traditional simulators producing results to slow. This is one of the challenges that the Research Accelerator for Multiple Processors(RAMP) project is addressing[41]. The RAMP project is a collaboration between different universities and cooperations such as Berkly, Xilinx, IBM and others[2]. Having a FPGA framework allows the software developers to test different implementations in operating systems and compilers before the architecture reaches production. To serve as the developing platform, the RAMPants[1] have opted for the Virtex5 FPGAs to provide the hardware platform. The processor cores used are a mixture of MicroBlaze soft cores, and PowerPCs hard cores. The reference designs available from the RAMP project focuses on either transactional memory, distributed systems or distributed shared memory[41].

While the RAMP project is the first one where which a well defined interface for communication between different parts, allowing for exchangeable parts, there has been different implementations of CMP in FPGAs. Amongst these, Socrates is worth mentioning. Socrates, being one of the early adapter, has focus upon proving that a CMP can be de-

[1]Member of the RAMP group

signed on a FPGA, and that it can be done rapidly[7]. The cores in Socrates are based upon an own implementation of ARM cores which in turn are connected to an crossbar mechanism to provide communication between the different elements. This allows the design to increase in size, leaving room for growth in FPGA transistor counts. The downside to the Socrates implementation is that the crossbar interconnect, as shown by Kumar et al.[21], wont scale as gracefully with the number of cores connected to it. This severely limits the amount of cores available to the Socrates platform, both in terms of FPGA resources, and in a real life situation where also factors like power consumption plays a major part.

Although the FPGA gains approval for prototyping hardware, there are still project prototyping using commodity hardware. The Stanford Hydra project is a research project testing out novel architectures for Chip Multiprocessors, basing their processor cores on four MIPS based processors[14]. To be able to simulate the processor design they have built a circuit board using four MIPS R3000 processors cores[28]. Each of the cores are connected to a floating point unit and a Virtex FPGA in order to form a single processor tile. By configuring the FPGAs, the Hydra project can simulate an array of different configurations.

2.2 CMP Architecture

Having several processor cores on a single chip brings forth new and challenging problems related to power consumption, cache strategies with multiple cores, communication between cores to ensure cache coherence and off-chip access. Although the physical scale has reached a new level, currently at a gate level of 45nm[10][2], these problems have been addressed before by the world of supercomputers. Seeing how supercomputers traditionally have several processors, distributed/shared memory, and a network between the processing nodes, many of the concepts can be transposed down to fit the scale of a single chip. Furthermore, some of the rejected ideas have resurrected seeing how the scale have changed from several processing nodes, to a single chip, moving the limits on latency and bandwidth. This have led to interesting CMP strategies, where tried and tested supercomputer paradigms have been reused.

More specific, as shown in figure 1.1, during the last years, the core frequency have stagnated, which has left the developers looking for other ways to improve the overall performance, while improving the power consumption. Since the gate size has had a drastic drop, it has been possible to put several cores on a single processor. Then by decreasing the frequency per core, the power consumption have drastically decreased. Other interesting features of the CMP, being modular, is that it allows for easier resource management. This in turn makes it possible to turn of inactive parts, in an attempt to reduce consumption of power. Another important point of focus is that of off-chip access to remote peripherals, e.g. memory. The gap between the CPU and memory has steadily increased, and has

[2] In 2007

evolved into a complex hierarchy, known as the Memory Hierarchy, figure 2.1. This states that the further memory are from the CPU registers, the higher access cost in terms of latency, but more memory it has allocated. This challenge has introduced the notion of having banks of cache distributed around the system, allowing for temporary storage closer to the processing unit. Since the storage capabilities, in effect, dictates the latency several levels of cache can be found on the CMP. In a multicore environment the challenge lies upon having the cores cooperating on retrieving lines of external data, avoiding retrieving the same data twice. This in attempt to try reduce the amount of unnecessary duplicate off-chip communication.

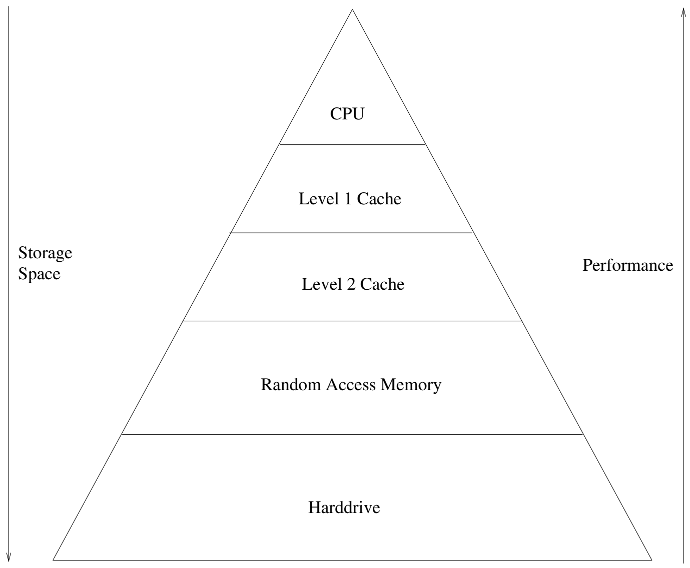

Figure 2.1: Example memory hierarchy

Although if the cores weren't to cooperate on the data retrieval, they would still need a way of invalidating cache lines. Each core might have their own copy of data in cache, and if one of the cores writes to the memory, the system need some way of notifying the other cores holding a copy of the data. If not the task would lie upon the software developer to lock memory accesses, which would be a tedious and error prone job.

2.2.1 Cores

The core is the main processing element on a traditional processor, performing instructions which it loads from memory. Traditionally the performance of a single core CPU has been given by the frequency of the internal clock, and the internal architecture of the core. The frequency has been dependant of the underlying feature size and has been steadily rising until year 2002, see figure 1.1. Combined with the fact that extracting performance per clock cycle gets exponentially harder[3], has led for a new way of extracting performance.

[3]e.g., achieving a perfect level of ILP is impossible due to the demands of perfect hardware[15]

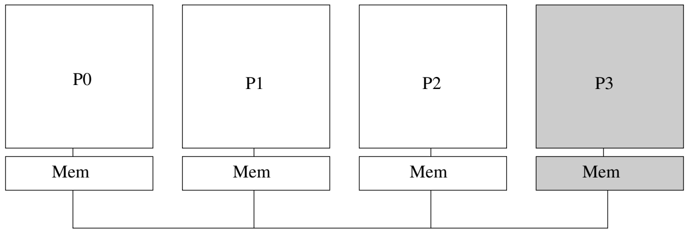

Figure 2.2: Core number 3 is disabled

Besides the raw performance challenges in designing a modern CPU, lowering the power consumption has increasingly gained popularity. The power consumed by a CPU is directly influenced by the frequency on which its internal core is set as shown in formula 2.4. As the generated heat is a function of the power consumed, the manufacturers have had problems cooling the CPU with apparatuses acceptable to the general public. This, and seeing how there is a lower bound to the latency of a signal traveling across the chip has led IBM to predict that their Power6 cores will extract the last amount of clock cycles available at circa 5Ghz[9] using the current available technology.

This has led to a search of other methods for increasing performance, acknowledging that the single core technology will stagnate at a given point using current technologies. Although at the time where this report was written a company named D-Wave in cooperation with NASA presented a co-processor using quantum technologies[1], this technology is still long from perfected, and has a long way before it is common household equipment. This has led to the manufacturers decreasing the complexity of a single core and increasing the total number of cores on each chip. This allows each core to decrease it's individual throughput, but the system as a whole will retain its performance[38].

Scaling down the complexity of a single core while decreasing the core frequency influences the power dissipation of the processor as a whole. This have led to a decrease in both the Dynamical and Static power dissipation. Static power dissipation is power dissipating due to transistor leakage[5], while Dynamical power dissipation refers to the rise and fall in current when the transistors changes state. A simplified equation for the total power dissipation is shown below in equation 2.1.

$$P_{total} = P_{static} + P_{dynamic} \tag{2.1}$$

Dynamic Power Dissipation

The dynamical power dissipation is attributed the change of state in the transistor. When a transistor is set low, it has to discharge to ground, which is the main source of dissipation. The following formula (2.2) as described by Jeraya et al.[18] and Gochman et al.[13] shows the dynamical power dissipation. Here F_0 is the clock frequency. The C_0 is the effective capacitance of the circuit, while V_0 is the voltage and α is the activity factor.

$$P_{dynamic} = \alpha * C_0 * V_0^2 * F_0 \tag{2.2}$$

In the same article by Gochman et al.[13], they show that the frequency can be approximated to be proportional of the core voltage V_0, which leads forth to formulas 2.3 and 2.4

$$F_0 \approx K_f * V_0 \tag{2.3}$$

$$P_{dynamic} = \alpha * C_0 * V_0^3 * K_f \tag{2.4}$$

As shown in equation 2.4, $P_{dynamic}$ is cubic dependant of the frequency. Thus by halving the frequency per core, the core will dissipate of one quarter of its original power. Then by doubling the number of cores on the chip, the chip will retain its performance while halving its dynamical power dissipation.

Static Power Dissipation

The static power dissipation, "leakage", is an effect of the current gate technology[12], mainly due to subthreshold and oxide leakage[12]. In their article, Ghiasti and Grunwald[12] presents equation 2.5. This models the static power loss as a product of the Voltage current, leakage current, the number of gates N and a scaling factor k. The scaling factor is dependant of the inherit complexity of the design itself.

$$P_{static} = V_{cc} \cdot I_{leak} \cdot N \cdot k_{design} \tag{2.5}$$

A overall measure of the effectively per power dissipated is shown in equation 2.6. This shows the static power leakage over the million instructions per second, giving a rough estimate of how resourceful the processor is.

$$Eff = \frac{P_{static}}{MIPS} \tag{2.6}$$

To reduce the static power dissipation, Muthana et al.[22] suggests that by reducing the I_{leak} factor, would have a great impact. One of the methods would find an architecture which allowed for disabling caches and cores, as shown in figure 2.2.1.

Spracklen et at.[38] also mentions that in a single core processor, much complexity and logic is used in a controlling context, not in performance issues. This leads to a high N in formula 2.5, and thus the effectively according to formula 2.6 decreases. Spracklen et al. also mentions that a high performance core is a complex design, which leads to a higher k_{design} and futher decreases the efficiency.

2.2.2 Cache

As the gap between the CPU performance and the memory latency have grown as shown in figure 1.1 has grown, the need for temporary storage of data has increased. This has led forth to a hierarchy of memory structures where frequently used data is located near the CPU in terms of access time, allowing faster access to more popular data. A sample structure is shown in figure 2.1, where the data with the lowest access time could be stored inside the CPU itself and its registers. Duplicates of data that is frequently used would be placed in the level 1 cache, less frequently used in the level 2 cache and so forth, whereas the data stored on the disk or other external devices would take the longest to access. This is due to external memory have a higher latency before the requested data reaches the bus, the distance the signals have to travel, and the obstacles getting there. E.g., a DDR2 has the memory clock set at 400Mhz, whilst a traditional hard drive has a seek time given in milliseconds. Both considerably higher than the internal registers to the core which operates on clock frequencies measured in gigahertz and where the registers can be accessed in a few clock cycles.

To solve this challenge, cache banks have been introduced into the computer architecture storing a subset of available memory close to the processor, and thus reducing the access time for a set of frequently used data. Which lines of data and the amount of data that the cache can store internally is given by an amount of different parameters. When the cache gets full and a new item is to be stored, the cache bank must choose which one to evict, and different strategies exists to choose the right one, such as Least-Recently-Used[35] and Random-evict[40] depending on how the cache stores its data in memory. Each cache unit can store a certain amount of datum, cache lines, in memory. What differentiate one cache organisation from another is how its chooses store its cache lines, and the different parameters controlling the behavior.

One of the easiest conceptual ways of storing cache lines would be to store the cache line at any given spot in the array of available cache lines. This strategy is known as fully associative, figure 2.3. To be able to keep track of which cache lines that is stored where in the cache, each cache lines' tag, i.e. the part of its associated meta data that describes which data that is stored in a given location, would have to be its full address. This would lead to an massive amount of overhead per cache line stored, seeing how for each line the cache must keep track of its corresponding address. Also during a lookup the cache must traverse through each line, matching its address to the requested address. This leads to a increase in latency when performing cache lookup. Another, faster way of organizing cache is directly associative cache, figure 2.5. Here a given number of the least significant bits of

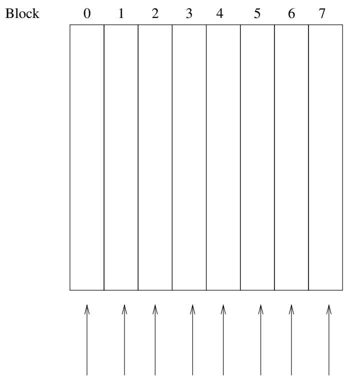

Figure 2.3: Fully Associative Cache

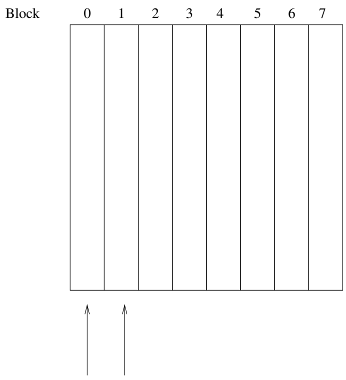

Figure 2.4: Set Associative Cache

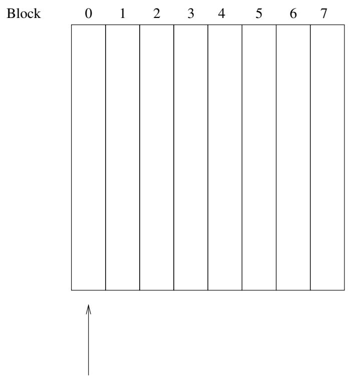

Figure 2.5: Direct Associative Cache

the address can determinate which index that the cache line will be stored in. This would be conceptually equivalent of the mathematical modulo operation. By varying the amount of bits used to determine the index, i.e. varying the number in the modulo operation, the cache can keep a different amount of cache lines. The two obvious benefits by this method contra the fully associative is, one, the cache can reduce the number of bits in its tag. If X number of bits is used to determine the index where the data is stored, address - X bits would be needed to provide the cache line's tag. The second advantage to directly associative cache is that a lookup requires significant less logic when determining a cache hit. A mixture between fully and directly associative cache is set associative, figure 2.4. The basic notion is that a set associative cache can keep several cache lines per index, i.e. the degrees of set associativity. A 1-set associative cache is the same as a directly associated cache, whereas a 2-set associative cache can store 2 cache lines per index.

Multicore Cache Architecture

Having several cores on a single chip introduces a new challenge, namely resource sharing. The basic challenge with the multicore cache architectures is the same as with a single core CPU, efficient use of off chip communication. Although the same problem, the environment has changed. Duplicate the number of cores on the chip, and the memory access will be duplicated using a naive single core cache strategy. Although the CMP is a relative new product in computer science, having several processing units in a computer isn't a new paradigm[32]. Challenges seen in CMP, e.g., communication, cache strategies, etcetera, have been addressed by earlier work. One of the more interesting effects in the CMP world is to see implementations which have been discarded in traditional supercomputing being reused in multicore CPUs. This might be strategies which have been discarded due to problems with latency, low bandwidth and so forth. Since all of the components on a CMP are placed on a single chip, old or discarded research can be re-evaluated seeing how the physical scale has changed.

To hide the gap between primary memory and the CPU, efficient off-chip communication is required. In a single core environment this can be solved using advanced prefetching, cache eviction schemes and etcetera. However, introducing multiple core on the chip have further brought new challenges. Seeing how another on-chip cache bank might hold the data requested from different core, some sort of cache cooperation is needed. Second, having several cache banks brings forth another phenomenon from the supercomputers, the Non-Uniform Memory Access(NUMA) effect, named Non-Uniform Cache Access(NUCA) [20] in CMP terminology. Although, to the core, each line of cache appears to be located in one uniform area of cache each cache bank stores a certain part of the whole. Due to the wire latency, pending on the physical location of the cache bank, accessing different parts of the memory will have different access times.

To help battling the problem several schemes have been proposed. Even though the implementations differs, they all have the same goal. Increase the off-chip communication efficiency, by making the cache banks cooperate. Chang et al. introduces an elaborate scheme based upon ideas from software[6]. Here all the cache banks are aggregated. When a cache bank evicts its data it will first try to "spill" the data over into another bank. However, if the cache is full with own data the bank will reject the "spilled" data. Another strategy is proposed by Dybdahl et al. where the instead of the LRU-scheme[35] a frequency counter is used[11]. Each cache is then allowed to grow shrink cache sets. In doing so, they allow cores with more frequently accessed data to dominate the cache.

2.2.3 Interconnect

As soon a processor has several cores on the same chip, it starts to require an interconnect network between the cores, and other resources on the chip itself such as cache banks. Depending on a various amount of underlying architectural features, such as the number of cores available, the wanted performance in terms of latency, bandwidth and finally the power consumption, the topology of the underlying interconnect varies[21]. Thus depending on the requirements of the chip in production, different topologies will suit different needs. However, there are two main groups of interconnects which will be discussed, the crossbar interconnect and a shared bus as seen in figure 2.7 and 2.6. These two represents two completely different strategies, and thus they have two different sets of characteristics.

The shared bus, figure 2.6 is a network where all of the resources, i.e. the ones on the same network, are connected to the same set of buses. Having several resources connected to the same bus presents the problem with arbitration. If several resources tries to communicates on the same time, the signal would be ruined and the transfer would have to restart. Hence the need for a mechanism which arbitrates either the signal from the resource itself onto the bus, or a device which tells which resource that are allowed to send signals onto the bus at a given time. Kumar et al.[21] discusses a mechanism in which the cores requests access to the address and data bus by communicating with a arbitration device. However, since the medium through which the devices communicates is a shared one, only one signal can be active on the bus at one time. Even so, Kumar et al. presents methods of pipelining

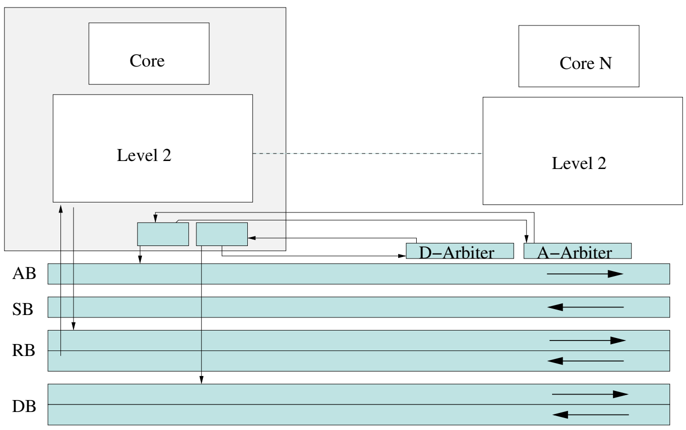

Figure 2.6: Shared Bus

As described in Kumar et al.[21]

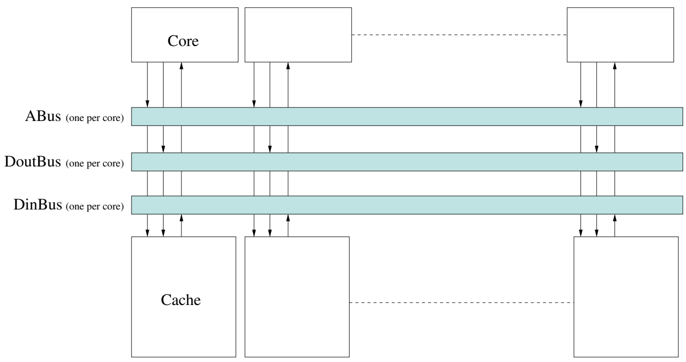

Figure 2.7: Crossbar

As described in Kumar et al.[21]

the process. One method is to include one bus for the control signals, address and data signal. Adding several buses, opens for the possibility to have one resource driving the address bus, while another one drives the data bus, fulfilling the previous request sent out on the address bus.

Unlike the shared bus, the crossbar topology, figure 2.7, relies on a direct connection between the resources on the chip. This helps reduce the time waiting for the data requested to arrive, but has a much higher cost in areal and energy consumption. An example shown in Kumar et al. shows that a crossbar mechanism introduces an area overhead of 11.4%, 22.8% and 46.8% with respectively 2-, 4-, or full sharing on a 8 core processor using a $400mm^2$ die[21]. However, cores will have have to stall less waiting for the data to arrived using the crossbar. The same paper shows that with a 8 core, fully connected crossbar and shared cache, the power consumption will match that of 3 cores in just the interconnect alone.

2.3 Introduction to FPGAs

Due to their flexibility in nature,the FPGA have gained status as "reconfigurable" hardware. Having flexible hardware is a great advantage in situations where the developer don't want to be locked down by the restrictions put forth by the ASIC. Examples would be in computer architecture research and embedded devices, where reconfigurability is the key. Having a configurable hardware unit is preferred compared to performance when developing hardware.

Although traditionally, the FPGA has been viewed upon as a device for prototyping new hardware, it has recently gained approval for usage in computationally heavy areas such as DNA string matching and several cryptology algorithms[8]. The FPGA being a reconfigurable device has always lagged behind the CPU in terms of clock frequency, and thus it has been ignored when raw clock frequency is preferred. However, the ability to act as a true parallel device outperforms the serial processor in areas where parallelism is the key feature. The reconfigurability comes on cost of frequency, and when this thesis was written, Xilinx produced FPGA's which operated at about 500 megahertz[46]. This in contrast to the modern processors provided by Intel and AMD which operated in the range of 3000 megahertz.

The way that FPGA's achieve such flexibility are based upon, two attributes. First the ability to configure each individual Configurable Logic Block(CLB), and second a configurable network connecting CLB's to each other. To program the FPGA, the developer loads a bitfile into the memory of the FPGA. Portions of the memory is connected to each resource in the FPGA, being either the routing switch, or the CLB. This leaves the developer able to create virtually any circuit by loading the right bitfile into memory, given that the FPGA have enough resources available.

Each CLB, figure 2.9, consist of several logic elements[8]. This might range from D-Flip Flops and lookup tables to coarser units as Arithmetic Logic units. Having several logic blocks in the same block gives the developer the ability to create more advanced units without having to use an excessive amount of the available resources. However, it is important to retain the possibility to have a fine grained output. An arithmetic logic unit will outperform the lookup table at arithmetic operations, but it will not give the developer enough flexibility to design more specialized units.

The routing on a FPGA is controlled by utilizing pass-through structures[8], controlling their behaviour through the bitfile. On modern FPGA's, the CLB is connected to a nearby connect box as shown in figure 2.8, forming a Island Style network[8]. At each intersection, the control bit decides if the signal should continue in the same vertical or horizontal direction, switch direction or be routed to a neighbouring CLB. That way it is

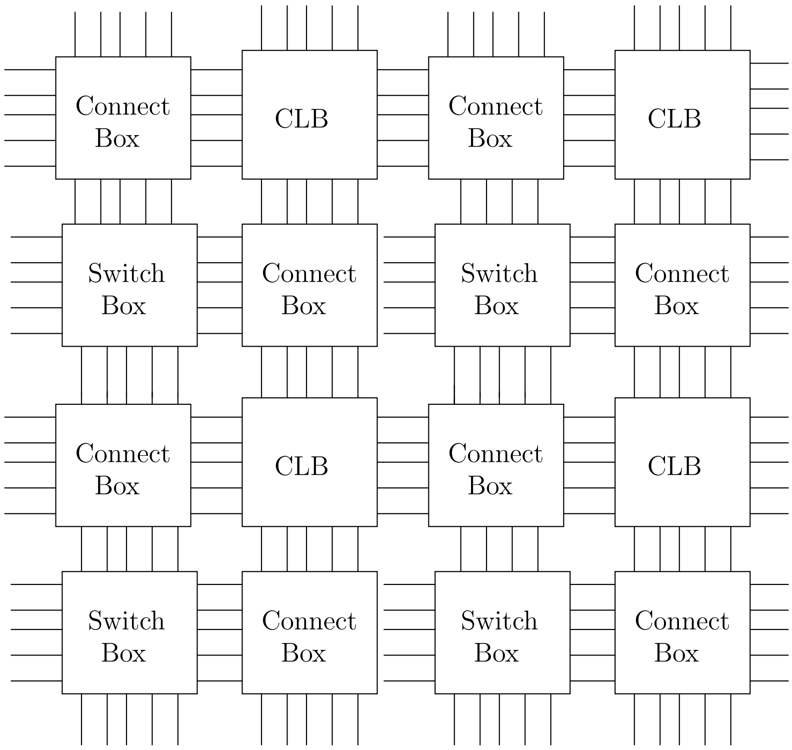

Figure 2.8: FPGA routing

possible to route the signal between different parts of the FPGA, forming complex designs utilizing more than one CLB.

When deciding upon which FPGA to use for this project, several factors are important. A modern CMP is a complex design, and complex designs utilizes a lot of logic cells. Of the FPGA's available to the author, the VirtexE 2000 has 43,200 of logic cells, while the Virtex 1000 has a number of 27,648. Greater amount of logic cells available allows for more complex implementations, using more complex structures or having more soft cores. Figure 2.10 shows a sample of soft cores and their resource in terms of slices on the VirtexE FPGA, each slice being an aggregate of two logic blocks. Other structures that are important are the interconnect, on-chip block RAM and off-chip communication. A integral part of a multicore processor is its cache banks. Thus if the FPGA has on-chip block RAM, additional slices can be saved not having to implement storage blocks using logic units.

2.4 Cores

When designing a multiprocessor on a FPGA, one of its most fundamental features is its cores. Of the cores available to a designer there are two different sub-types, depending

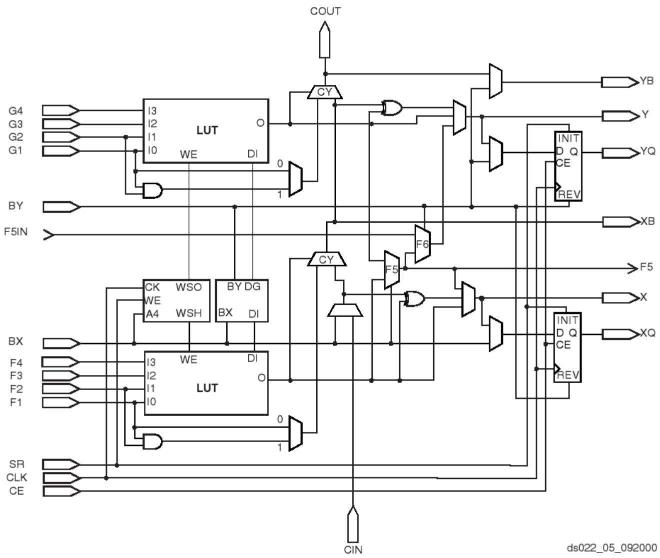

Figure 2.9: Sample CLB[42]

on (1) the wanted qualities and (2) what the hardware supports. What differentiates the hard core from a soft core is if it is a physical implementation, e.g., core molded onto the FPGA, or if it has to be implemented together with the rest of the code.

The soft core, is a separate project delivered in a form that allows the developer to synthesize it together with the rest of the code. The core might be delivered in form of VHDL, or an already implemented netlist mapped to the underlying hardware. Having an independent core gives the developer more freedom to experiment, and decouples the core from the underlying hardware. The hard core is a new phenomenon seen where the FPGA manufactures integrates existing processor cores on the FPGA itself. The hard core allows for greater performance compared to an implementation in VHDL, being a specialized circuit. However by utilizing a hard core the design gets more bound to the underlying hardware and thus scarifies flexibility for performance.

Of the soft cores available there are mainly two groups, commercial available such as the MicroBlaze[45] and freely available microcontrollers as the NanoBlaze and other experimental cores developed by hobbyists found at sites as OpenCores. The PicoBlaze[43] core is a product of the Xilinx Cooperation, and is a small 8-bit controller. For the PicoBlaze controller, Xilinx have opted for a unit which leaves a small footprint in the design, and thus have optimized away several complex instructions which would have made the controller increase in areal. Instructions such as multiply, divide and floating point calculations are non-existing. There exists schematics which shows how to connect several PicoBlaze controllers together to gain support for such instructions. However, in doing so would render the main feature of the PicoBlaze void. Being a small microcontroller, one of stressed points about PicoBlaze is that it provide an alternative to hardware circuits without the areal overhead normally associated with introducing a microcontroller.

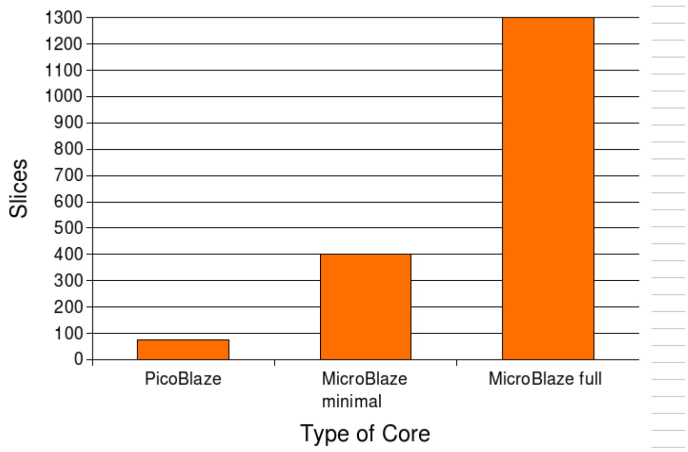

Figure 2.10: Number of Slices per Core.

2.4.1 MicroBlaze

The MicroBlaze microcontroller is the grown up version of the PicoBlaze. Being a 32-bit processor, it matches the modern day processor both in data and address width, and thus it is more suited for modern applications than its proceeder, the PicoBlaze. MicroBlaze implements a wide array of instructions, adding support for floating point, multiplication and several others on the expense of areal and resources used. However, one of the major selling points is that the MicroBlaze controller can be customized to provided the needed functionality. Support for such instructions as floating point might be omitted if the design doesn't require it, freeing space for other components. The controller also, in attempt to match the PPC-core found on some Xilinx FPGAs, confirms to IBM's CoreConnect[17] architecture[45]. This allows the hardware developer to extend the controller with extra peripheral units. Some of the most important buses includes the Local memory bus (LMB) and On-chip peripheral bus (OPB). With these two buses the controller can attach units through which it can communicate using a memory mapped scheme. The major difference between the two buses is that the LMB is a much simpler interface, connecting units which are to guarantee a one cycle response. This bus is where the designer normally would connect on-chip memory such as Block RAM controllers. OPB is a more complex bus, allowing slower peripheral units. Typical examples would be peripherals which are not memory, including units such as media access controllers and off-chip units. The final way of attaching units to the controller is through a link called Fast Simplex Link(FSL). The MicroBlaze can have 8 FSL interfaces which provides a low latency interface through which hardware accelerators can be attached.

2.4.2 PowerPC

Of the hard cores available, Xilinx have embedded IBM's PowerPC 405 on their Virtex-II Pro line of FPGAs. The 405 is an embedded core developed by IBM to suit the embedded marked. This includes a more specialized system for memory management and specialized registers for debugging etcetera. The external interface matches that of the MicroBlaze controller, so hardware developed for one would suit the other without, in theory, code rewrite. What differs the PPC405 from the MicroBlaze is that, in being a PowerPC, it must confirm to the PowerPC standard. Each PowerPC must correctly implement the User Instruction-Set Architecture(UISA). The USIA guaranties that the controller will behave exactly the same as all other PowerPC cores when in userspace. This leads forth to the PowerPC having a better support for compilers, and operating systems than the MicroBlaze core. Although a version of microcontroller-Linux has been ported to MicroBlaze, several others including NetBSD have been ported to the PowerPC platform[31, 29, 26].

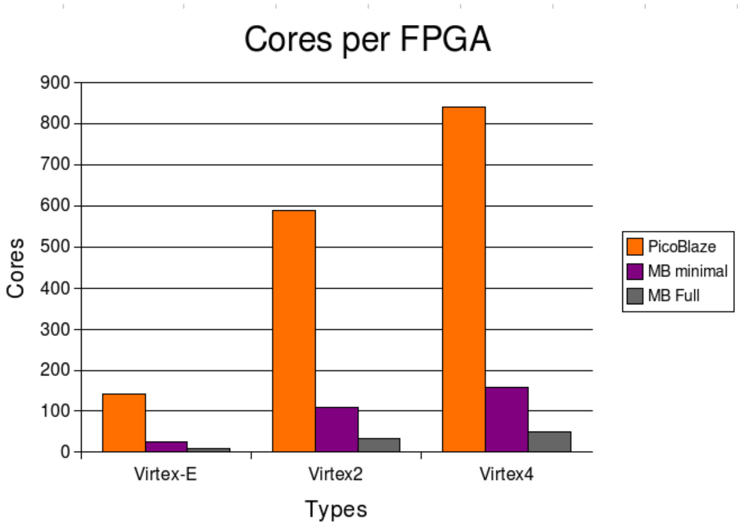

Figure 2.11: Number of Cores on a FPGA

One major implication the type of core imposes, is the number of cores available on a single FPGA. Figure 2.11 shows the amount of cores available on various types of FPGAs. However, its worth mentioning that it is an approximation. The number of cores are a function of the number of slices available on the FPGA over the number of slices occupied by a single core. In real life other resources should be taken into consideration. Even so, figure 2.11 gives an rough estimate over the number of cores available.

2.5 Environment

2.5.1 Hardware

The server which hosted the project was a CompactPCI IBM Compatible server running the Debian Gnu/Linux operating system. On board the host computer were two BenERA CompactPCI DIME-II motherboards[24]. Each of the two BenERA motherboards had two Virtex-E FPGAs as shown in figure 2.12, respectively marked red and green.

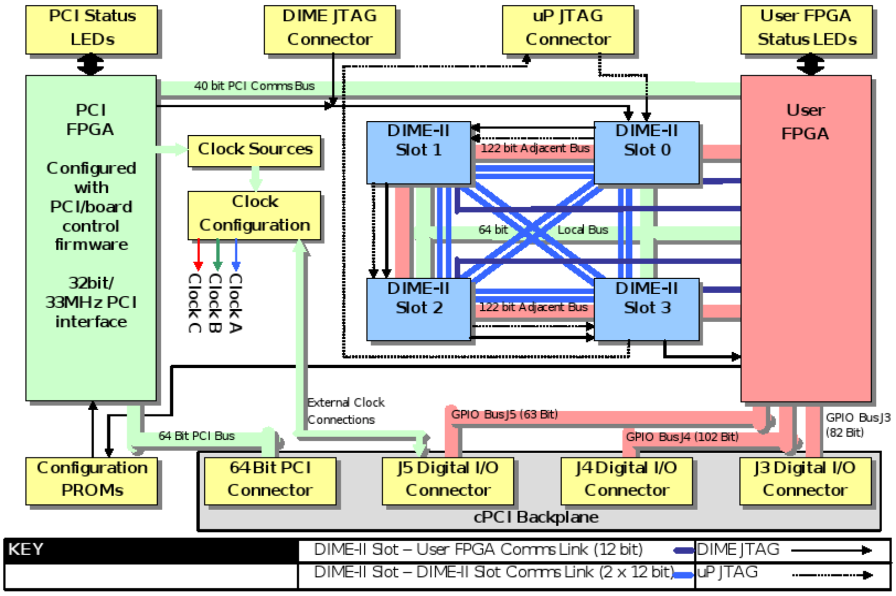

Figure 2.12: BenERA functional diagram [24]

Having two FPGAs on the same motherboard, Nallatech could occupy one to perform solely administrative functions. That FPGA is marked in the functional diagram(2.12) with green and is the PCI FPGA. The PCI FPGA acts as the bridge between the User FPGA and the host computer, simplifying the communication between the software and hardware. To achieve this, Nallatech loads the PCI FPGA with firmware that communicates through the PCI bus on the host computer. To provide communication between the host computer and the User FPGA, the PCI FPGA has a FIFO buffer which can be read and written to from both User FPGA and the software through the FUSE library described in section 2.5.2.

The user programmable FPGA was a single Virtex controlled through the PCI FPGA[24] and JTAG[33]. Further, the same motherboard could be extended with four new modules confirming to the DIME-II industrial standard. Modules that fit the DIME-II standard includes memory modules such as SDRAM and DRAM or FPGAs, including the new Virtex II-Pro.

2.5.2 Field Upgradable Systems Environment - FUSE

To provide communication between the User FPGA and the controlling software, Nallatech ships a FUSE library with their motherboards. This creates an abstraction layer between the software developer and the design running on the User FPGA[23]. The library gives the software developer the opportunity to control the clock frequency, loading bitfiles containing the FPGA design and configuring the DMA communication channel.

2.5.3 Virtex-E

The programmable FPGA hosted on the BenERA motherboard is a Virtex-E. The Virtex-E is a SRAM based FPGA[42], which means that the behaviour is defined by loading the generated bit stream into SRAM memory. The bitfile loaded into memory will control the different logic elements and routing resources, behaving close to the generic FPGA described in section 2.3.

The configurable logic block is implemented as shown in figure 2.9, with four logic cells pared into two slices. Each logic cell has in turn one four bit input lookup table, which acts as the function generator. Each CLB can aggregate their Lookup Table(LUT) s, and that way it is possible to get function generators with a greater width, totalling at 5-6. However, in the modern lines of Virtex FPGAs, the native width of the LUT is 6, such as in the Virtex 5[46]. Besides the role as a function generator, and thus as a pure logical unit, it is possible to configure the logic blocks such that it will act as memory. Done correctly, the developer can use the CLB s to create memory banks. Each CLB having a four bit input is able to store either 1x16 bit RAM or connect two CLB to generate either a 2x16 or 1x32 bit RAM block. A more resourceful way of storing RAM on the Virtex-E FPGA is to use the already existing Block RAM structures. Placed evenly spread between each row of logic elements, Xilinx has placed 96 BlockRAM on the xcv1000E model. Each BlockRAM being a 4096 bit dual port RAM gives the FPGA a total amount of 393,216 bits dedicated memory. Another feature of the BlockRAM is that it might be used as a large LUT taking the memory address as input, giving the data stored as the output.

The routing network on the Virtex FPGA is close to the one described in section 2.3. However what differs is that each connect box is connected to several other lines running a various amount of distances. Each CLB is connected to, what Xilinx calls, a General Routing Matrix(GRM) which acts as both the connect and switch box. Each GRM is is connected to their direct neighbours, the neighbours with a Manhattan distance of one. To help decrease the latency, each GRM is also connected to a longer wire, which stretches over 6 GRMs. However this wire is driven from the end GRMs, but is accessible from the boxes in between. The last set of wires is those who runs from one side of the chip to the other, allowing for rapid global communication. Finally all GRMs can access global signals such as clock signals and reset.

2.6 Tools

The tools used can be divided into two main groups, the ones used for hardware development, and the ones used to develop software. The hardware tools for this project is mainly those delivered with the Virtex-brand of FPGA s. This includes Integrated Studio Environment(ISE) which works as a IDE providing everything from syntactical analysis of the VHDL code, down to the creation of the FPGA specific bitfile. The tool to configure the MicroBlaze core is XPS, which easily allows for configuration. The software development tools used is mainly various ports of GCC to create both the running environment on the host machine, and compiling applications to run on the FPGA implemented processor.

2.6.1 GCC

GCC is the Gnu Compiler Collection, and has been ported to several platforms. One of the major advantages to the compiler, which makes it cross compatible is that they have separated the different functional layers[39]. The first layer is the language layer, converting from a programming language to an internal tree structure. Afterwards the compiler will optimize the tree, before it passes it to an machine dependant layer. By having done so, all which is needed to support a new platform is to extend the back end to support a new architecture. This has made the GCC-compiler the preferred choice for many embedded producers. Both the embedded PowerPC core and the MicroBlaze controller have a version of GCC ported by Xilinx.

2.6.2 ISE

Integrated Software Environment is the development environment created by Xilinx[37]. The application is a front end to the entire chain of tools needed from synthesising the written VHDL files, to the generation of the chip specific bitfile to be loaded onto the FPGA. The general flow in creating a bitfile starts with the developer writing VHDL code which specifies the behavior of the hardware. Then XST, will generate a Xilinx spesific netlist. Afterwards the applications will perform map, translate and placement and routing(par). This will map the resources needed by the netlist to the resources found on the actual FPGA. Afterwards the bitfile used to configure the chip is generated by the bitgen command. All this is taken hand of by the ISE, and thus decreases the development cycle.

2.6.3 XPS

To allow the user to make customizations to the MicroBlaze microcontroller, Xilinx packs XPS with their Embedded Development Kit[44]. Through the XPS application, the developer can extend the MicroBlaze or PowerPC core by attaching different modules to one of the many buses. Through the application the developer can tune the core to include certain features such as floating point unit, hardware support for multiply and division.

Chapter 3

Methodology

This chapter will describe the design of the multicore processor. It will start with a short description of the overall design notion, before it will present the most important components. After having presented the multicore processor design it will present some of the software developed to control the CPU. Finally it will present some of the developed benchmarks and test applications used to test the working CPU.

3.1 Introduction

The architecture is designed for extendibility, allowing the developer to test new and novel architectural changes, only rewriting or changing the parameter in question. The partitioning of the system is shown in figure 3.1[1]. The processor's main memory is implemented in software, while the rest of the design is implemented in hardware using VHDL. Having the memory in software allows the designer to easily calculate the number of cache misses, each software memory access not being in the on-chip cache banks. Software based memory also has the advantage that it is much cheaper in terms of areal, thus freeing resources from the FPGA which can be used for cache or other logic.

The software communicates to the hardware through the PCI FPGA, using a memory mapped scheme. On the hardware side the software communicates to the PCI FPGA, a specialized FPGA mounted to provide an interface between the programmable FPGA and the software running on the host computer. On the PCI FPGA the software accesses a FIFO buffer which is connected to the User FPGA, containing the data to be communicated to or from the processor. Being a 32 bit wide FIFO, the software will fetch 2 words, before deciding if the last transfer should be a write or read to the buffer pending on the nature of the memory access. Once transfered to the buffer, it is communicated through a bus dubiously called the Peripheral Component Interconnect which is connected to the

[1] Red marking the systems developed, blue the 3rd party cores, and gray FPGA specific modules

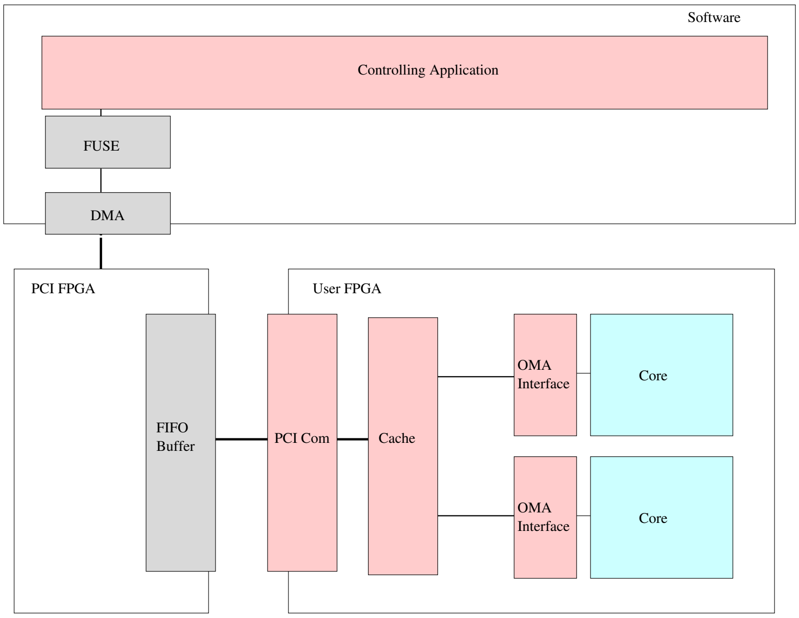

Figure 3.1: Overall Architecture

User FPGA and the PCI Communication module as shown in figure 3.1. This is where the data gets handled by the implemented design, and translated into a format which suits the internal components. Limited by the width of the buffer, the communication module has to multiplex 96 bits of data[2] into 3 words which can be transfered between the FIFO buffer and the User FPGA.

Once the data reaches FPGA, the data is communicated between the different modules, e.g. arbiter, cores and cache through the unified signal interface described in section 3.2. Having a unified interface allows all the modules to be replaced as long as they confirm to the same interface. Having this allows for the replacements of the cores themselves, freeing the design from the restrictions put forth by the MicroBlaze core and the Xilinx team, which in turn relinquishes the complexity associated to developing with the Xilinx tool chain. This leaves an overhead in the communication latency between the cores themselves and the rest of the system, having to translate from the CoreConnect architecture to an customised interface.

3.2 OMA

To uphold the demand for extendability, allowing for easily exchangeable units, a interface which all components must adhere was created. The interface itself was loosely based upon the CoreConnect architecture, although some of the fancier mechanisms removed

[2] Address, Data and Control Word

Name	Description
Valid	If the data presented by the data lines are either valid, in the case of a read or have been written to memory.
Data_I	The data to be read when the RW flag is set to high.
Data_O	The data to be written when the RW flag is set high.
Address	The address to which the data either should be read or write
RW	The direction of the transfer, will read on asserted.
Active	When this signal is high a transaction is ongoing.

Table 3.1: Signal interface

to achieve a simpler model for communication. The signal interface was based on a pure memory mapped processor architecture where all communication between the processor and the off-chip peripheral units were done through an elaborate memory scheme. The downside was the inability to communicate meta data in a separate channel, such as internal statistics, cache hit ratios and etcetera. However this could be solved using other mechanisms as JTAG interfaces and ChipScope, providing a much cleaner and more customisable mechanism to probe the hardware.

The signal interface that each component needs to confirm to is shown in table 3.1 and contains the bare minimum to provide communication. However, it is meant as interface between the different layers in the architecture, and not between specialized components, leaving room for advanced features where it is needed.

The interface defines 4 signals which are driven by the source, namely the Data Out bus, Address bus, Active flag and finally the Read not Write flag. The two foremost signals are the Data to be transfered out from the component and the address of the data that are either to be written or read. When a transaction is wanted the source will assert the active flag until the transaction is done. The direction of the transfer is signaled through the RW-flag. If the RW flag is asserted the following action is a read transfer, and the client side should drive the signal on the Data In bus, ignoring the Data Out. On the other hand, if the RW-flag is set low, the operation in question is a write transfer, and thus the source should drive the signal on the data out bus. Independent of the direction of the transfer, when the destination side is finished handling the transfer, it should assert the valid flag for one cycle and thus signal that the transfer is completed.

3.3 Core

The core, being the processing element of the CPU, is the main component of the processor. However to this project the core alone is just another component needed to be able to perform benchmarks on different architectures. Important criteria is the number of cores that can be made available in the design, its documentation and to some degree its feature set, i.e. a core can't have a too scarce instruction set or else it wont be able to run the most basic benchmarks and tests.

Of the different cores investigated, MicroBlaze was the one that satisfied the all the 3 different criteria. By opting for a soft core such as MicroBlaze the design has the ability to scale beyond the hard cores available on the FPGA, which in the case of a Virtex-II Pro is limited to two PowerPCs. Also by utilizing a soft core the implemented multiprocessor is not bound to the specific implementation of the FPGA itself and thus if the resources available on the FPGA can't support the design, the VHDL code can be synthesized to fit on a new type of FPGA. As to the feature set, the core has an RISC based ISA which is most able to fulfill the demands imposed by the benchmarking and test suite. One of the key features is the MicroBlaze's ability to cut down on the features, such as the floating point operation. This in turn cuts down on the resources utilized by a single core and allows for a greater amount of total cores on the chip, or more advanced features requiring more logic.

3.3.1 Wrapping

To be able to easily add new cores to the design, a thin proxy layer has been wrapped around the MicroBlaze core. The wrapper layer translates the signal from the CoreConnect and the OPB to a scaled down version following the specifications described in section 3.2 and table 3.1. By designing a proxy between the MicroBlaze core and the rest of the system it is possible to exchange the core themselves without having to affect the rest of the system. I.e. by abstracting the memory interface, if the system were to exchange its cores with a NanoBlaze core the only part having to be rewritten is the Off-chip Memory Access (OMA) interface to the core.

Internally the MicroBlaze has two main busses through which it can communicate with different peripheral devices such as external memory, SDRAM, different media access controllers and more traditional units found on a processor such as on chip ram blocks. The MicroBlaze core divides its internal buses between those peripherals that are able to deliver data with one cycle latency, the LMB, and multiple cycles, the OPB [45]. One cycle latency is possible if the data is stored on chip, and hence it is meant to be the bus that supports on-chip memory access. However, since the memory subsystem cannot guarantee a one cycle access to memory it has to be connected to the OPB bus.

Although the MicroBlaze core itself is a Harvard Architecture[15] core, differentiating the data space from the instruction space, these two spaces are both joined in the OPB and thus minimizing the logic needed to implement different address spaces. Although by doing this the complexity of implementing a instruction cache is greatly reduced. This opposed to exposing both a data and instruction bus outside the core itself.

A single core is conceptually made up of mainly 4 components as shown in figure 3.3.1, communicating to the rest of the system through the OMA interface. As its kernel the

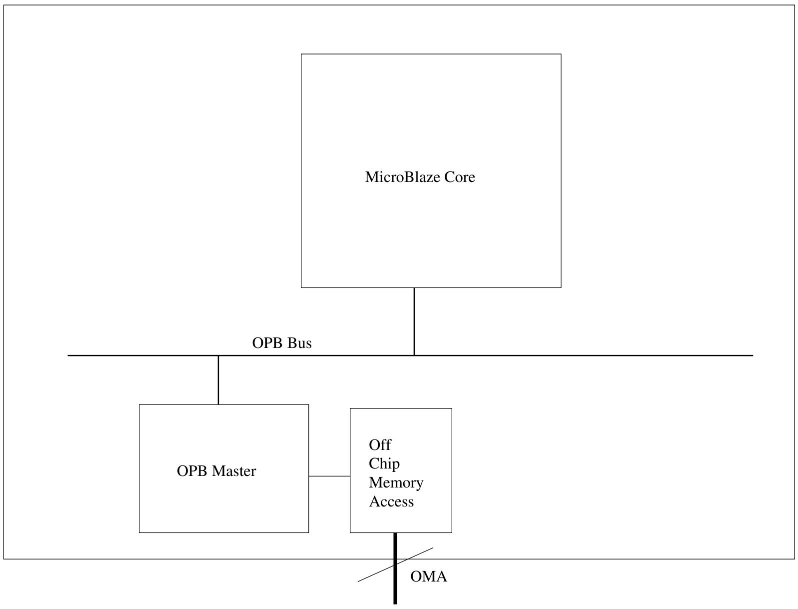

Figure 3.2: Core

core is build up around one MicroBlaze soft core. The soft core is connected to a OPB bus which confirms to the CoreConnect standard as provided by IBM[17], and on the other side a OPB Master which connects to the Off-Chip Memory Access component.

3.3.2 CPU Identification Component

To be of use in a multicore environment, a method of separating one core from another is required. This could be solved by using a special CPU version register containing the ID of the core itself, or split the address space of each core, each prepending the first bits of the address to identify the core's ID. In this architecture, that would have a serious impact on the available memory space as soon as the number of cores reaches more than 4 cores. Splitting the address space would also involve a lot of complexity in any shared level cache. However, the MicroBlaze core version 4 does not provide any user defined registers, which is a function in the newer versions of the microcontroller.

This led to the development of a mechanism using MicroBlaze's Fast Simplex Link bus. As described in section 2.4, each MicroBlaze core can be connected to up to 8 FSL-interfaces. One of those is a unit hardwired to deliver the core identification upon a request from the core itself. This gives the software developer the possibility to differentiate the different cores. This is done by issuing a GET instruction on port 0, storing the result in a register. Afterwards it is possible to check the given register and act accordingly as seen in example code 1.

```
// Get the CPUID from port 0 and store it in register A.
GET REG_A, PORT0;
switch REG_A do
    case 0
        // Code for core 0
    end
    case 1
        // Code for core 1
    end
end
```

Algorithm 1: CPU Identification

3.3.3 Implementation Challenges

One of the major challenges by utilizing the MicroBlaze microcontroller is the proprietary solution which is bundled with the core itself. The MicroBlaze code is distributed to the end user using a encrypted VHDL format to protect the IP. Hence to be able to instantiate the core in a design, the encrypted VHDL must first be configured through 3rd party applications which outputs a netlist. The major problem with the netlist is that spesific parts of the components are bound to specific components on the FPGA, relying on knowledge of the supporting FPGA. This is done by using the LOC-constraint, and the purpose is to deliver optimized hardware which fully utilizes the FPGA.

The problem arises when, through the VHDL code, several cores are being duplicated. The best way to do this would be to use the generator statement which is a key function in VHDL, and which is heavy used generating structures which are repeating in nature. Algorithm 3 shows an easy and clean way to do this. Here the *numCores* variable could be a globally set variable.

```
Data: numCores
cores : for for I in 0 to numCores -1 generate do
    instance : microBlaze PORT (
    signals => signals_core(I)
    )
end
```

Algorithm 2: Generating several cores

However, due to the nature of the MicroBlaze core, this method led to several complications. First, the Xilinx tools generates files which depends on the correct name of the MicroBlaze instance. E.g. if the core is configured with the name "System", the above mentioned example would fail. The generate statement prepends the name of the entity with a composite of the generate label and the counter. In this example a specific core would be named "cores(I).instance" in the netlist. The core being configured with the name "instance" would fail, having resources depending on the "instance" resource. Second is the LOC-constraints, each instance of the core will try to occupy the same resources on the FPGA, and thus will fail.

```
instance : microBlaze PORT (
signals_core0 => signals_core(0)
signals_core1 => signals_core(1)
);
```

Algorithm 3: Generating several cores without generate

To solve this, another strategy was adopted. Instead of having XPS generate a single core per design, the design had to have several cores. The downside to this method is that there is no easy way of configuring the number of cores in the form of VHDL, and it is dependant of configuring XPS and MicroBlaze files instead of having a variable in the VHDL code. This means that for each core in the system, a whole new set of MicroBlaze core with its assorted accessory such as the OPB-bus and OMA interface manually has to be configured.

3.4 Cache

The cache block implementation is a fully parametrisable component which can be extended with different evict strategies. One of the key points is to be able to simulate different cache organisations, different line sizes, and different cache line sizes[3]. One of the key points when cache structures are concerned is to identify fully and directly associative caches as sub-set of set-associative caches. I.e. directly associative caches can be described as 1-set associative cache, and that fully associative caches in fact are 1-set associative caches where the tag length is the full size of the address width, leaving no bits left for the index. When that is identified, two key parameters that decides the structure of a cache block is clear, namely (1) set-associativity and (2) tag-size. The bits of the address used for indexing is given by the tag-size since the index concatenated with the tag must form the full address of the cache line. I.e. if the tag size is 25, and address width is 32, width of the index must be 32-25, which is 7.

One important parameter which is not included in this design is the cache line size. The cache line size describes the amount of bytes stored per cache lines. This is omitted in the design due to several reasons. Having a cache line size greater than the data width is favorable in an environment such as the Intel Pentium 4 where the main memory is a dual port RAM which delivers 64 bytes, and the data width of the CPU is 32 bit. If the cache would have been 32 bit, the cache block would discard 60 bytes per memory access. In this case a cache line size which is greater than the data width would make sense. However, the BenERA motherboard provides a 32 bit channel off the FPGA through which the chip communicates. This means that FPGA can request 32 bits of data per off-chip memory access, and for each time it requests data it needs to multiplex the address, direction and

[3]The amount of data stored per cache line

data transfered over the bus, which is overly expensive in terms of latency. If the cache line size is to be varied from a minimum of 32 bits to a multiple of 32, it would have cost that multiple of 32 times one memory transfer to data from to and fro the main memory. Another important factor that speaks for the omit of cache line size is the heavy limitation of the SelectBlock Ram available on the VirtexE FPGA. The cache line size would mean that the number of BlockRAM used per line would increase with the with $\frac{cachelinesize}{32}$ per 127 possible index times the degree of set-associative. Seeing how the number of available blocks on the FPGA is 94 this would lead to a reduction of the address space and sets available.

3.4.1 Storage Unit

To be able efficiently store data on the FPGA itself the on-board SelectRAM blocks have been utilized. This in comparison to an implementation written in VHDL which, if written incorrectly, would failed to recognised as RAM structure and instead use CLB. The first attempt to write a cache block without SelectRAM used 103% of the total FPGA resources which obviously was unrealisable, however it was less restricted by the limitations of the SelectRAM. A single memory block, which is the component wrapping the on-board block ram is shown in figur 3.4.1 and can hold up to 127 lines of cache, one set per line. Each SelectRam block can be configured as a dual port ram with an address width of 8 bits, and a data width of 16. Seeing how the data width of the CPU is 32 bit, an encoding scheme is needed. This is solved connecting the most significant bit to either 1 or 0, as shown in figure 3.4.1, leaving 7 bits to represent the address. This decoding scheme is also used at the SelectRAM block containing the meta data such as tag, LRU count and various amounts of flags such as valid bit and dirty bit used in the evict strategy. This makes the design use 2 SelectRAM blocks per 127 lines of data stored, which is a large part of the 94 available on the VirtexE FPGA, but it frees up CLB's that can be used to implement several cores and logic.

Seeing how an address width of 7 bits, i.e. 127 different indexes, can be a little restrictive, the design can expand the address space. This is done by stacking several memory blocks, into an array. When a given address is requested, the right memory block is addressed by using the most significant bits of the address[4] to identify the right memory block. E.g., if the address 10000000_2 is requested, it will address the memory block with address 1, and in turn its memory cell 0. An expansion of the address space means that additional 2 to the power of extra bits of Single Memory blocks would be used per set, rapidly exhuming the available SelectRam blocks available. The indexes available is a direct result of the tag length parameter.

Besides the number of lines available, one of the parameters available is the number of sets per line, or the set associativity. The naive implementation of set associativity in the given FPGA environment would have been to store any number of sets in the same memory block. Although possible, that would have led to an increase in clock cycles

[4] address(7 to length)

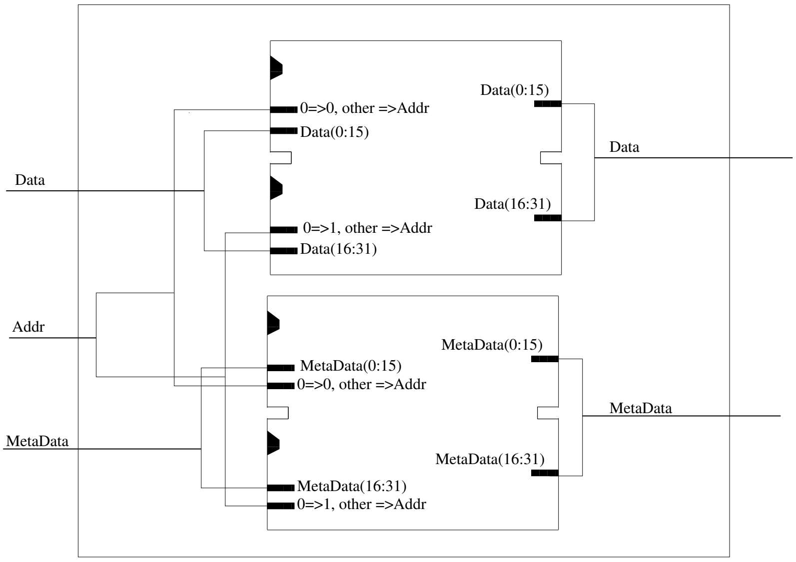

Figure 3.3: Single Memory block

wasted searching for the cache with the right tag. Done wrong the benefits of having a set associative cache would have diminished, and one would in terms of latency and logic overhead reach the scale of fully associative cache. Another more efficient way to store sets in the same SelectRAM would be to say that address 0 and 1 keeps the first set and so on. This would have required 2 extra clock cycles, or one extra clock cycle per set associativity, seeing how the chip must wait one clock cycle per memory access to the embedded SelectRAM block. Besides the waste of clock cycles, it would reduce the available indexes by the factor set associativity. Although a mutt point, since it is a direct result of all available strategies. Figure 3.4.1 shows how this implementation have done it, using a constant amount of clock cycles independent of the set associativity. For each new set added to the design, a new duplicate of the memory block gets added. Seeing how the address in the SelectRAM block is depending on only the most significant bits, the same index will be hit in all of the sets, and all sets will send both their meta data and data to a multiplexer as shown in the figure. The multiplexer will match each tag with the requested address, and output the data from the right memory set.

3.5 PCI Communication

The PCI Communication device, as seen in figure 3.5, acts as a gateway between the User FPGA and the PCI FPGA. Its main task is to provide to the system a way of communicating out of the User FPGA confirming to the OMA interface. Its role is to translate the internal signals into a format that suits the PCI FPGA. When the data are sent to the PCI FPGA, it alone will pass the data along to the host computer through DMA, and finally it will reach the controlling software. The communication between the

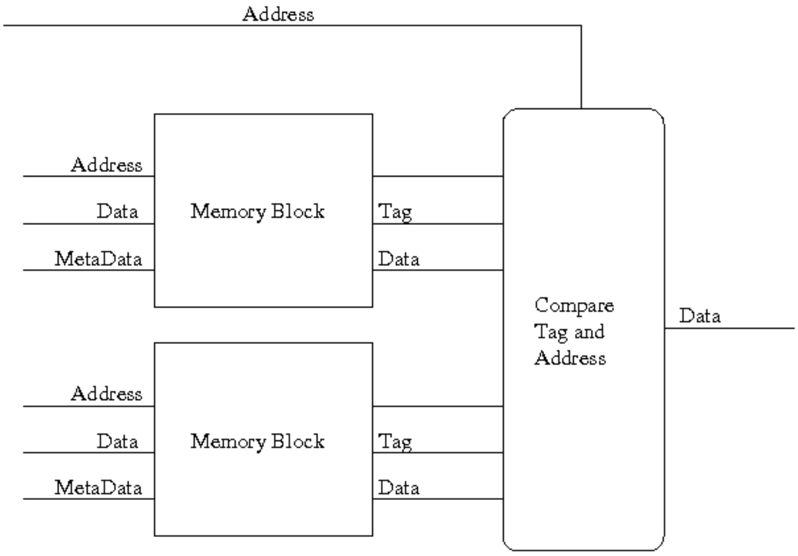

Figure 3.4: 2-way set associative block

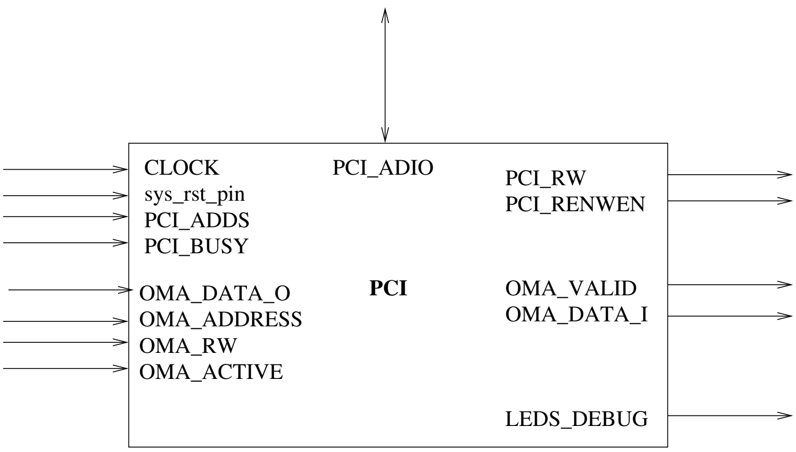

Figure 3.5: PCI Unit

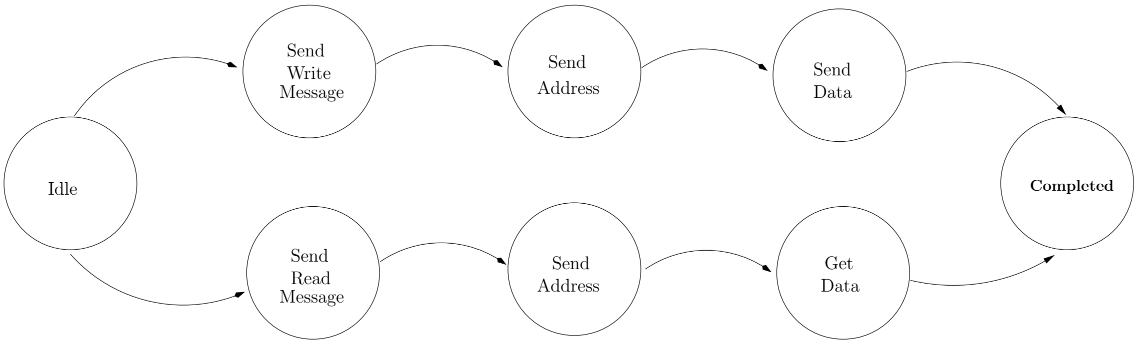

Figure 3.6: Com States

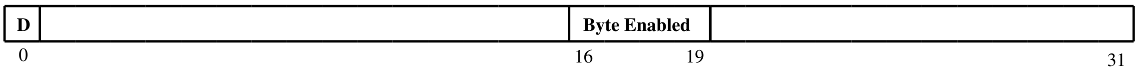

Figure 3.7: Control Word

User FPGA and the PCI FPGA happens through a dedicated 32 bit wide FIFO buffer on the PCI FPGA to which the User FPGA both reads and writes. However, seeing how the aggregated width of the data to be transfered to and fro the PCI Communication module exceeds 32 bit, a multiplexing scheme has been implemented.

To be able to access the memory banks implemented on the host computer in software, 3 important bits of informations is needed. First the direction of the transfer needs to be relayed, next the address to be operated on and finally the transaction of data which are to be stored or read from memory. With a 32-bit address and data bus, this would exceed the 32 bits available for communication through the PCI FPGA. Hence the communication will happen in the stages shown in figure 3.5. First the communication module will signal the direction of the transfer, either a write or read message. Then it will transfer the address of the cell in question. Pending on the direction of transfer, it will either get data from the PCI FPGA, or it will drive signals onto the bus itself.

A transfer gets initiated when the Active flag is asserted. When the active flag is asserted, the communication module stores the address, and in the case of a write transfer in an internal register to be able to meet the timing constraints put forth by the PCI FPGA. To initiate the communication with the controlling software on the host computer, the communication module puts a control word on the bus. The control word, shown in figure 3.5 contains information about the direction(D) of memory transfer and byte enabled bits(BE). The direction, if asserted indicates that the data should be transfered from the host computer to the User FPGA. Otherwise the transfer is a write action, and the data will flow from the User FPGA to the PCI FPGA. The Byte Enabled masks the bytes, and is ignored in the case of a read statement. When a read transaction happens, the software based RAM will fill up the entire word from memory. However, if the transaction is a write action, the software based memory will only write to the part of memory signaled by the BE-flag.

3.6 Software

To be able to communicate with the processor, and to provide a opportunity for the processor to store its memory, dedicated software has been written. The controlling application loads the bitfile containing the processor onto the FPGA, then loads the requested application into memory before it resets the design and starts its role as memory.

When the application starts, it will initialize the memory. By hosting the main memory on the host computer, the design can allocate the amount of memory available on the host computer instead of storing data in the BlockRam on the FPGA. This effectively increases the amount of storage from 96 BlockRams à 4096 bits[5] to the amount of memory available on the host computer to the maximum of 4 Gigabyte[6]. Depending on the user-input, the application will either load a given program or all zeroes into the memory. The latter intended for debugging purposes only. When the memory is loaded, the FPGA is initialized and the communication channels are set up. When the bitfile is loaded onto the FPGA, all buffers and reset signals are cleared. When everything is configured, the control is transfered to the FPGA.

```
Input: data
if state == control then
    state = getReadOrWrite ( data )
else
    if state == read then
        state = handleRead ( data )
    else
        state = handleWrite ( data )
    end
end
```

Function `handle(`*data*`)`

When the software has transfered the control to the hardware, it goes into the main loop shown in algorithm 5. The goal of the loop is to retrieve a word from the PCI Commu-

[5] 376 Kb

[6] The address space of the processor being 32 bits

while *1* **do**
    data = getWord32(from FPGA);
    handle(data);
end

Algorithm 5: Main loop

nication module[7] and pass it on to the correct function. This is done by implementing a basic decode-and-dispatch loop[36]. For each turn the main loop fetches a word from the FIFO buffer on the PCI FPGA, and dispatches this to the handler function, as seen in algorithm 4. Depending on the internal state of the software itself, the handler function will further dispatch the data to the correct method. Due to the thigh coupling between the internal states of the hardware and software, it is important that the software adheres to the states defined in section 3.5. If the software fails to send a message when the hardware is locked in the "Get Data" state, both the hardware and the software will go into a deadlock. Although a bad idea in real-world application, it was an acceptable trade-off due to the decreased complexity compared to a fault tolerant communication model.

3.7 Benchmark

To help benchmark the platform, several benchmarks has been developed. Several commercial benchmarking applications, such as Spec 2006[19] are available to hardware developers. However in the case of the Spec benchmark suit, it would require an underlying operating system and a ported version of the standard C library. Although a version of Linux exists for the MicroBlaze core[8], it has not been ported to support a multicore architecture. This combined with fact that such benchmarks are not only testing the raw processor performance, but also the 3rd party libraries deemed the traditional benchmark suites unnecessary.

The suite developed presents two different kernels, each trying to benchmark different properties in the design. The first presented is testing the raw performance available to the processor by iterating through a sequence of number, accessing the memory as little as possible. The second benchmark does the opposite, loading and storing as much to the memory as possible.

3.7.1 BogoMIPS kernel

The BogoMIPS kernel, appendix A.1, is implemented as simple for-loop, iterating a various amount of times as seen in algorithm 6. What the BogoMIPS kernel tries to to benchmark is the raw performance of the overall system just doing enough calculations to increment

[7]See section 3.5
[8]µc Linux

i = 0
for *i<MAX ITERATIONS* **do**
    i += 1
end

Algorithm 6: bogoMIPS

the loop counter. By benchmarking this, it forms the basis for comparison for the rest of benchmark kernels.

3.7.2 Load-Store kernel

The load-store kernel, found in appendix A.4, is a kernel designed to stress test the bandwidth between the processors and the memory. It does so by first storing a certain value to the Nth memory addresses after the text segment, as seen in the code below.

```
storeLoop:
   sw   R11, R2, R0     ;store R11 to address R2+R0
   addi R1,  R1, 0x4    ;add 0x4 to R1
   add  R2,  R1, R2     ;add R1 to R2
   cmp  R4,  R3, R1     ;R4 = R1 - R3
   blei R4,  storeLoop  ;if R4 <= 0
```

When the store loop is done, the kernel starts over loading the data back in to the processor. By doing so the kernel is a memory bound kernel. Besides the benchmarking function, the load-store kernel also works as a memory boundary checker.

Chapter 4

Results

This chapter will present the findings of the benchmarks presented in section 3.7. This will provide foundation for the discussion in chapter 5 where a more final analysis will be presented.

4.1 Introduction

In a modern processor design, several factors decides if the processor is regarded as an high performance CPU. Even though the goal of the thesis was not to deliver a high performance multicore architecture, it presents several results varying the architectural parameters to show a proof of concept, and that the architecture described in chapter 3 allowed for rapid prototyping.

Being a prototype processor, the performance cannot match the one of a Intel or IBM. The clock frequency of the core operated at 42MHz, and thus it had been relentless to match it to an ASIC processor running at 4GHz. The timing comparisons done in this thesis was matched to a single core processor involving no cache on the same FPGA. That allowed the benchmarks to show if a configuration was better, but can in no way be compared to another configuration running on another system solely based on the timing information.

The timing model used is based on the Pentium instruction rdtsc which loads the number of cycles into a register specified by the developer. This was done in order to get a more accurate view than the coarse system clock could deliver. However, the draw-back to using the host computer as a timing device was that other processes acquiring resources affected the end results. To help diminish the effect of resource sharing, an average of several runs is presented as the results.

The final set of data extracted from the results was operations done at the main memory

Name	Number of Cores	Cache Size	Set Associativity	Index size
sCore	1	0	0	0
mCore	2	0	0	0
mCache	2	2Kb	2	7

Table 4.1: Processor Configurations

on the host computer. For each memory read or memory write to the host computer, an internal counter was increased which kept track of the total number of memory accesses.

The tested configurations are shown in table 4.1. The first configuration, sCore, is a single core without no cache, which will form the basis for comparison. The second configuration, mCore is a dual core CMP without cache. The final configuration, the mCache, is a dual core with a two-set associative cache, using 7 bits as the index, giving it 2Kb of available storage.

4.2 Benchmark results

4.2.1 BogoMIPS Kernel

The BogoMIPS kernel was designed to investigate how long the overall system used to perform a single performance bound problem. It did this by returning the amount of time spent performing a single loop.

Figure 4.1 shows the amount of time each configuration used for a given problem set. Both the single core and the multicore configurations uses between 59301819.8 and 11619442516 cycles on the host computer to finish. Without any cache the configurations are forced to request the same instructions and data on the host computer, and thus a lot of cycles will be wasted due to the processor design that has to wait for the memory transfer to complete. The mCache configuration managed to load the data requested into cache, and always had a cache hit when the data was re-requested. This lead the mCache to finish faster than the other cores, and had a approximately time of 150'0000 cycles independent of the problem size. This showed that the at problems of this size, the results are memory bound.

As presented by figure 4.2, both of the configurations without cache, the sCore and mCore designs has between 1223 and 240046 reads from the memory itself. This shows that for each instruction, the processor has gone outside the chip and requested data from the host computer. The mCache configuration, with a cache size of 2Kb, read a constant number of memory locations independent of the problem size. It also shows that the amount of writes varies with the mCache configuration.

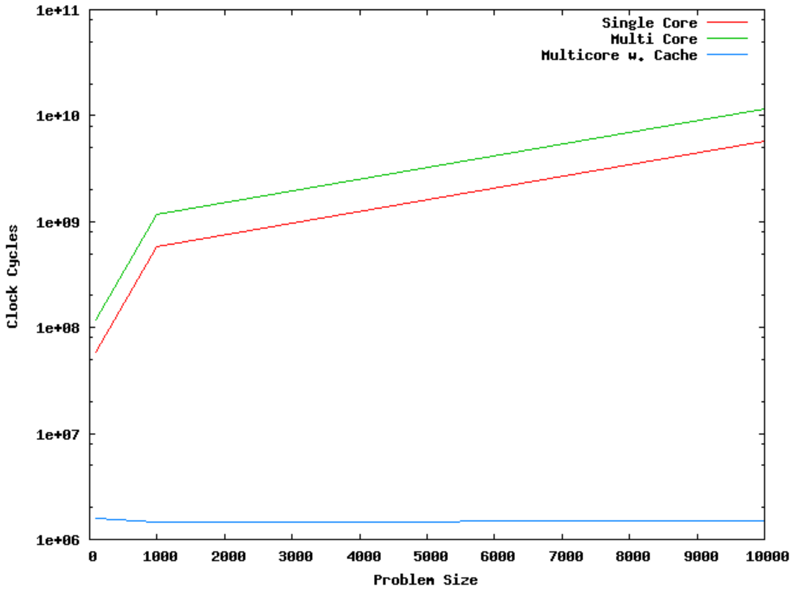

Figure 4.1: Different Configuration performance using the BogoMIPS kernel.
Logarithmic Plot

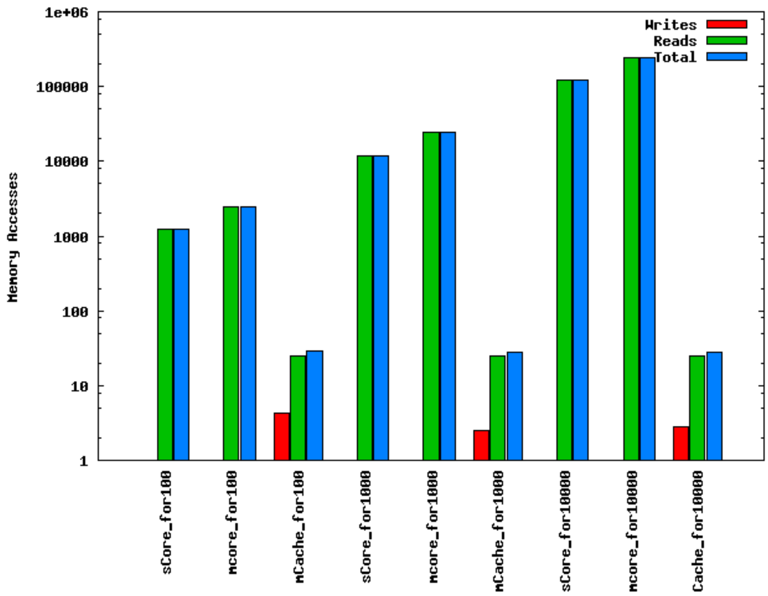

Figure 4.2: Different Memory Access patterns used by different configurations with the BogoMIPS kernel.
Logarithmic Plot

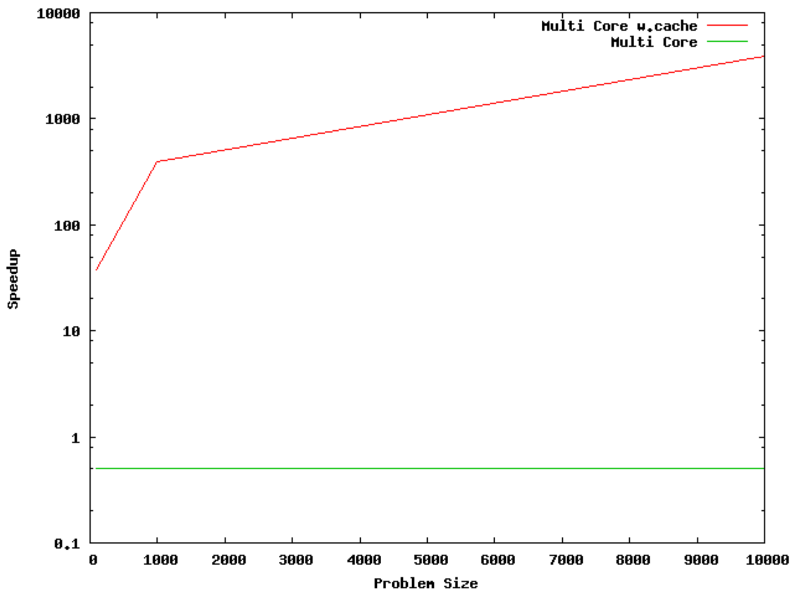

Figure 4.3: BogoMIPS Speedup
Logarithmic Plot

Figure 4.3 shows the amount of speedup gained by enabling cache on the multicore processor, which is in the magnitude of 3000. The speedup graph supports the finding presented in the above graphs, that the problem without cache is memory bound. The speedup graph also shows that with arbiter, in the case of the multi core design, that the round robin algorithm works as described in appendix B.5. By alternating between the cores on the chip, it should take approximately twice the time loading the same amount of instructions into each of the cores as the time used by a single core design. This is confirmed by graph 4.3 as it shows speedup of 0.5, compared to the single core processor.

4.2.2 Load/Store kernel

The load store kernel was designed to stress test the memory system of the processor design. The kernel tests is memory bound by first storing to the Nth first addresses, before it tries to re-read the same locations into memory.

Figure 4.4 shows the amount of cycles from used by the host computer waiting for the test to finish. The mCache configuration failed the verification phase of the test, and thus it has been omitted from the results. When the mCache kernel ran it would go in to a stall loop without sending the termination signal to the host computer to let it know that the application kernel was done. This might be due to a flaw in the internal memory structure of the cache itself, or somewhere else.

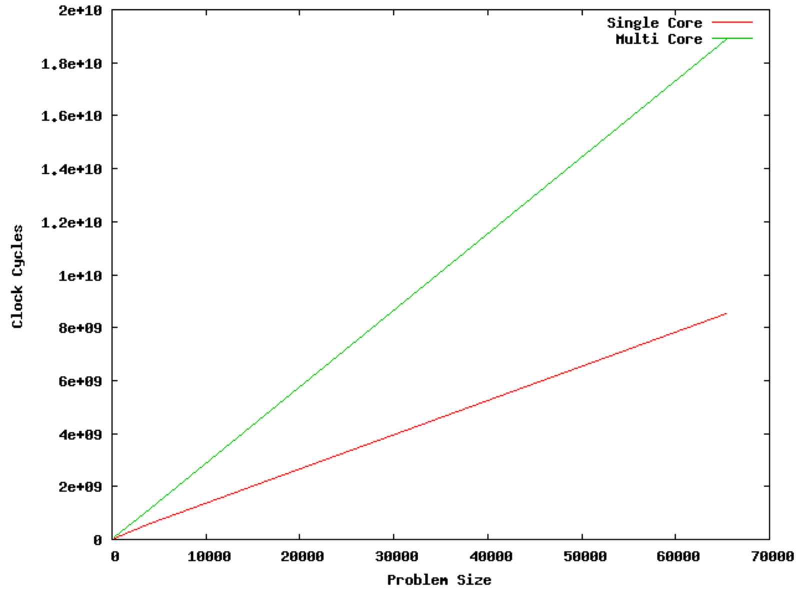

Figure 4.4: Different Configuration performance using the MemTest kernel. Logarithmic Plot.

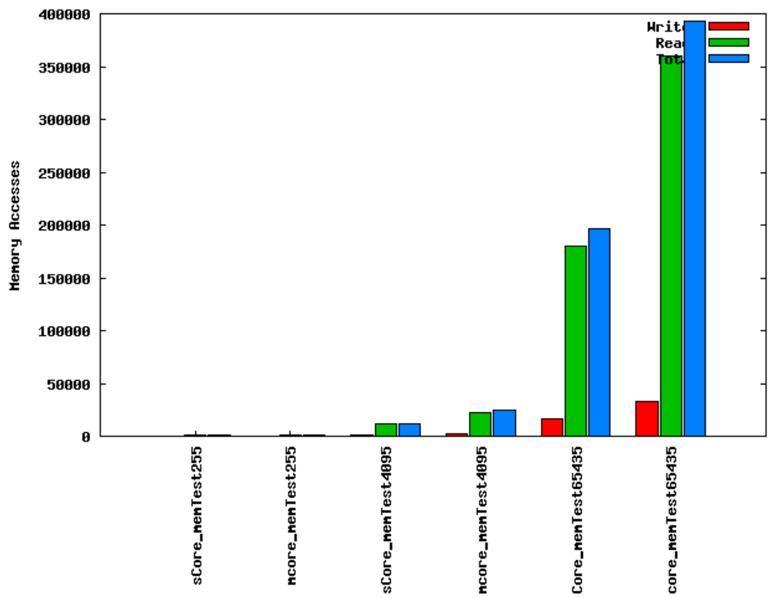

Figure 4.5: Different Memory Access patterns used by different configurations using the MemTest kernel.

Logarithmic Plot.

Figure 4.5 shows the number of memory accesses that the different configurations that cleared the validation phase had. Shown in the same figure is that the multicore architecture approximately has the double amount of memory requests as the single core configuration. This validated against the source code found in appendix A.4, where the kernel is ignorant of the number of cores available, leaving each core to request the same amount of memory as a single core found on the sCore configuration.

Chapter 5

Discussion

This chapter will present the analysis of the various topics touched in the thesis. It will start with a discussion of the core used, before it analyses the cache solution. It will then discuss the timing motivations before it continues with a presentation of the challenges involving 3rd party products. It will finish with a discussion of the results found in chapter 4.

5.1 Cores

The cores available at the beginning of the project were the MicroBlaze microcontroller and the PicoBlaze statemachine. Later in the project the Virtex-II Pro and its two PowerPC cores were made available. Of the cores available at the start of the project, Micro- and PicoBlaze, only the MicroBlaze was the one closest to a modern processor in design. The PicoBlaze controller, at 8 bit, didn't implement the most fundamental instructions found in a modern processor, hence rendered it useless in a Chip Multiprocessor design. The PowerPC hard cores on the Virtex-II Pro would have brought better compiler support to the project, and the bug presented in section 5.4.1 would most likely have been avoided. The PowerPC cores would also have done the porting of several operating systems easier, being a full processor supporting advanced features such as memory management by having a Memory Management Unit. The PPC would also have freed up a lot of resources such as LUT s, already embedded on the FPGA. However, the hard cores would break with one of the fundamental ideas in this project. By opting for a hard core, the system would have been locked down to the two cores delivered with the FPGA itself. One of the ideas with this project is that is should be possible to extend and test new architectures. Although two cores could be enough in testing cache and pre-fetching strategies, it would fall short when testing new interconnects where the amount of cores are vital to the outcome. Also, IBM predicting that they will see processors with 60 cores in the soon future, a locked down design depending solely on the two cores provided by the Virtex-II Pro would be futile. Thus the argument that embedded PPC would have freed up resources falls on the ground that with two embedded cores, there is nothing left than cache and interconnect to use resources for. This, unless an array of extra components, not including cores, are to be tested.

By utilizing the MicroBlaze core, the design gained the ability to grow in both the number of cores, and still remained flexible enough to test several solutions. One of the key features of the MicroBlaze core was its ability to turn of unneeded features. This included the division unit and the floating point unit of the processor. However, should a benchmark or an application require those two features it is possible to enable them. This would lead to the cores requiring more areal on the FPGA, but the option has been retained. By having this option the developer can decide upon which feature set being the more important one. A researcher developing an intrinsic interconnect would most likely increase the overall feature size of the core itself in order to be able to have more cores on the FPGA. A person testing new multicore algorithms on the FPGA, might be more interested in having floating point operations instead of having a large amount of cores available.

5.2 Cache

One of the features in this project is the extendible cache solution. In a modern day processor, several levels of cache exists to ensure a reduced latency between the processor core and the memory. Another possibility to separate the different cache banks, allowing each core to have a private cache. This is done both to ensure a faster access time, and to "protect" important data from being evicted by another processor core.

Since each of the components implemented in the design enforces the OMA signal interface as described in section 3.2, it is possible to connect one cache solution to another one, without having to rewrite the system. Since all the sinks[1] have the same interface, the cache component can easily be connected to the arbiter, PCI Communication module or another cache module. By connecting the cache to another cache, one will in effect get a hierarchical memory. By connecting two caches with their correspondent cores to an arbiter, which in turn connects to an new cache bank, one will have created an two level cache with level 1 private and level 2 shared.

On the Virtex FPGA, having a hierarchical memory structure will solve the evict problem, since one core would not have the possibility to request memory from another core's cache bank, and hence force forth an evict. However, to decrease the latency from having the signal traveling great distances demands great planning. On board the Virtex-E FPGA, each block RAM is evenly placed in columns as shown in figure 5.2, with the configurable logic blocks in between, all blocks guaranteeing a one cycle lookup time[42]. Hence to reduce the latency, either the logic of the cache unit must be reduced, or the distance the signal has to travel must be decreased.

[1] Components handling the incoming request

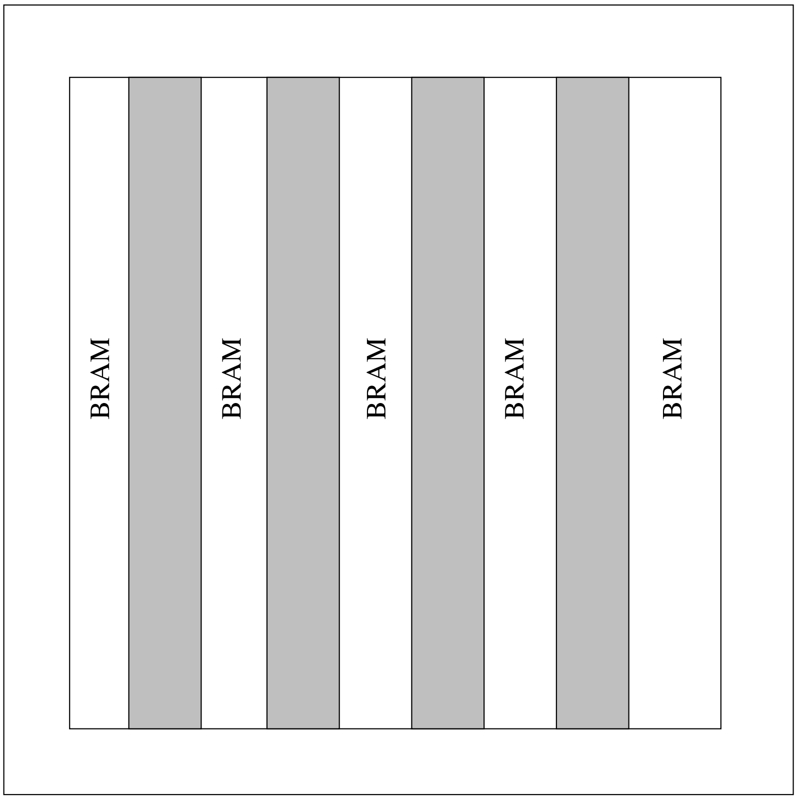

Figure 5.1: BRAM Location on the FPGA.

One of the motivations of the thesis' is to create an extendible architecture, thus the overhead in logic by making the cache solution extendible is required by the project's nature. This leaves the option of a better placement for level one cache compared to the placement of level 2 cache.

5.3 Time

One of the motivations behind the thesis was to reduce the amount of time spent waiting for the results, and thus be able to produce results in a shorter amount of time. The normal work flow for using the simulator at the NCAR group is do compile the new design, and let the simulators run over the weekend. When implementing the design in a FPGA, there are a significantly reduce in the time spent stalling waiting for the results to arrive.

Of the time spent in development, most of the time went into designing and verifying the hardware developed. A lot of time went into debugging errors that came during run-time at the FPGA itself, but which failed to appear during the simulations. This was due to the hardware giving “random” feedback, such as the FIFO buffer being full, the software might be stalling waiting for execution time hence not requesting the available data. Situations which was impossible to emulate in the simulator, and hence hard to detect even with the proper use of LEDs. Although the cost of debugging the hardware could have been significantly reduced by access to the right equipment. Xilinx delivers a product named ChipScope, which through JTAG allows the developer to embed probes in their hardware design, extracting information at run-time. At the time this thesis was written the sole way of communicating to the JTAG interface on the FPGA was through a specialized cable not available to the writer.

However, with the behavioral code working, it took at a maximum of 5 minutes integrating it to the existing code base. A lot of time were saved defining a strict signal interface through which the different subsystems could communicate. From the code were done until the bitfile came out it took at a maximum 20 minutes. This is attributed the different processes required for creating a bitfile. I.e. Translation, Placement and Routing and the Mapping phase. From the bitfile was done to all of the benchmarks and validation kernels were done an additional of 10 minutes were spent waiting.

This leaves an overall time spent waiting for results at a maximum of 30 minutes depending on the problem size of the benchmark suite. This time would differ if the complexity of the design itself changes, i.e. designing hardware which would force the placement and routing tool to spend more time.

5.4 3rd Party Challenges

This section will discuss some of the challenges introduced by the various 3rd party software and hardware solutions. It will start off with introducing a bug found in the MicroBlaze branch of the GNU Compiler Collection, before describing some of the challenges with the glue code provided by Xilinx to connect to the OPB.

5.4.1 GNU Compiler Collection for MicroBlaze

GCC as described in section 2.6.1 has been ported to suit the MicroBlaze processor core. This allows the software developer to write code in a high level language such as C or C++ to help speed up development. However in the branch of GCC developed by Xilinx, a particularly bug was found. This had to with the way that the compiler handled 32 bits immediate values in the case of a branch and loading memory addresses. The payload of an instruction can at max be 16 bits, seeing how some bits are reserved the Op-code and registers. To solve this Xilinx have introduced a special atomic Immediate instruction. The payload of a Immediate instruction will be placed in a special register on the CPU. The following instruction, which must be a compatible Immediate instruction or else the register will be flushed, will send an additional 16 bits. The 16 bits from the immediate

```
12:   30600024        brlid   r3, r0, 0x24
```

Figure 5.2: Correct assembly code

```
12:   b0000024        imm     0x24
16:   30600000        brlid   r3, r0, 0
```

Figure 5.3: Output from GCC

instruction will form the most significant bits, leaving the 16 bits from the following instruction to fill the least significant bits.

32 bit values are especially important when handling addresses. This might be the case when the developer either wants to load a word from main memory, or when the developer wants to do a jump. Figure 5.4.1 shows the correct code in case of a jump to current position + 0x24. However the output from the MicroBlaze port of GCC gave the output shown in figure 5.4.1. The code shown in figure 5.4.1 loads 0x24 into the immediate register, before it performs a brlid instruction, brlid being the branch immediate instruction. Internally the MicroBlaze core will concatenate the value of the immediate register to the payload of the brlid, the immediate register forming the high part of the word. This will have the effect of jumping to the address IP + 0x24000000, instead of the wanted IP + 0x24. The bug is due to having the software developers not being aware of the importance of order in which the immediate value has to be sent.

5.4.2 Xilinx IP Glue for OPB

To allow for rapid development of external peripheral devices, Xilinx have bundled in their Embedded Development Kit a customizable IP which glues the user developed hardware unit to the more advanced OPB. The main goal of the IP is to hide the complexity of the CoreConnect architecture. It does so by handling address decoding, making the user developed Property(IP) respond only to the given address, i.e. make certain that the component only respond to the configured address range. It also have a lot of other functionality to used by the developer, as hiding unused signals.

To help implement the wrapper layer between each MicroBlaze core and the rest of the design, the design utilized the IP glue code. This was utilized in the core wrappers described in section 3.3.1. Doing so led forth a new set of challenges. Each IP attached to the OPB or LMB has a set of parameters automatically configured by the tools provided by Xilinx. Amongst others this included the addresses space that the implemented IP should respond to. I.e. a IP can be configured to only respond to memory request between address X and Y. However, in the Xilinx IP there were two places where this address range is set, and the tools provided by Xilinx only updated one of the spots. This left implemented IPs to respond to address at $baseAddress + 0xFF$, which in effect gave the processor a 255 byte address space. The fix to this problem was to hard code in a address range which suited the project.

The second bug found in the IP glue code was regarding the address buses from the MicroBlaze core itself. From the master that the proxy connected to, two sets of address buses were provided. In the IP glue code it was asserted that the implemented IP only needed the signal of one the buses. The MicroBlaze core on the other hand alternated between the two buses when it requested addresses. This left the processor design only responding to the requested addresses which were 8-byte aligned, e.g. 0x0, 0x8, 0x16, instead of wanted 4-byte aligned addresses. This left the wrapper layer only responding to half of the addresses requested by the processor core. The fix to this problem was to multiplex the signals from each bus into a single bus in the wrapper layer, by checking which of the buses currently active.

5.5 Benchmarks

The benchmarks done in chapter 4, were designed to validate the different configurations, and to give a rough estimate of the performance of the design itself. The first benchmark was a simple for loop which tested the performance of the design itself. As presented by the graphs 4.1 and 4.2 most of the time spent by the different designs went into stalling for the memory transactions to complete. This was also the dominating part in the application designed to be memory bound. Seen in the first benchmark, the execution time of the multicore was roughly twice the time of a single core processor. This coincide with the hardware designed. As seen in appendix B.8 the multicore architecture has a arbiter between the PCI communication and the cores themselves. This means that for each request, the multicore architecture must wait for the core utilizing the memory bus to finish before it can request memory from the subsystem. This arbitration is not featured on the single core processor as shown in appendix B.9. Seeing how the first kernel is negligent of the number of cores available it forces both cores on the multicore architecture to fetch all the memory location. Since the arbitration mechanism is based upon a round robin algorithm the multicore design will have to wait twice the amount of time as a single core processor for the memory transfer to complete.

The multicore design with the cache ran much faster than both the single- and multicore architectures without cache available. This was due to most of the time was spent waiting for the memory transfers. The cache managed to store all of the requested data lines, and thus it only had to request the each data once. As shown in figure 4.2, the multicore cache design only requested 25 memory reads independent of the kernel problem size, which coincide with the size of the application itself.

In the load store benchmark, see section 4.2.2, the multicore with cache designed failed to verify during runtime. During the benchmark the core with the cache managed to go into a infinite loop without sending the termination signal. This probably means that there is a bug in the write branch of the cache solution. Throughly tests were done in the simulation bench on the cache sub-system forcing the microblaze core to do a number of writes without managing to produce the same results, the cache subsystem managed to pass all of the test vectors. The test vectors in the simulation bench also tried to investigate if the possibility that bug had to do with the amount of memory read into the cache. One possibility could have been that the tag system was flawed, an address with the same index had the possibility to falsely trigger a cache hit. However, this did not turn out to be the case in the simulator. That test vector also passed.

Besides that the mCache design did not pass its verification, the same tendency found in the for-kernel appeared in the load/store kernel. With all cache disabled, the designs were more bound by the time it takes to access memory, than the actual computations. Each memory request has to travel out on the host computer's PCI bus, through the PCI FPGA. Then the data would have to be copied from kernel to userspace before it reaches the controlling software before a acknowledging signal is sent back the same route.

This also behavior concurs with the tendencies in computer architecture, and is why a software based memory was opted for. Having the memory on the FPGA itself would have significantly decreased the run time of a single kernel. By putting the memory on the host computer, it put forth a artificial memory access delay found in most modern computers. It also presented that including cache in the design, the for loop kernel was virtually depending on the time it used waiting for the compulsive memory transactions. This was seen on the constant number of cycles spent, independent of the iterations spent.

Chapter 6

Conclusion

This chapter presents some final remarks about the project, before it presents some ideas for further works.

6.1 Conclusion

To be able to do research on the field of Computer Architecture in a period where there are a lot of parallel research projects, a platform which brings results within the hour is to a great advantage. There are a lot of methodologies for speeding up the traditional simulators, such as using clusters and a various amount of techniques to decrease the time spent in simulation[16]. However in terms of both accuracy and time, hardware outperforms software in simulation of hardware architectures.

During the period of the thesis the author has implemented a framework for simulating and experimenting with chip multiprocessors in hardware. A core has been decided upon and integrated into the chip. A parametrizable cache solution has been designed and implemented. Benchmarks which tested and probed the implemented design has been written. The benchmarks showed that both the multicore and single core architecture worked. However, the benchmarks produced found a bug in the design of the cache when accessing memory above a given size. Of software, an application which hosts the main memory, and controls the application has been written. Also, by testing the different configurations the thesis has shown that it has made it possible to test different processor configurations in a rapid manor.

This thesis has shown that it has significantly reduced the time waiting for results. The thesis has also laid the groundwork for further development of a hardware platform which can be used to prototype novel computer architectures. It has implemented the most common structures found on a chip multicore, and abstracted most of the hardware specific challenges put forth by the 3rd part vendors. Due to the strict interfaces between the different modules of the design, future developers can easily configure the platform in virtual any desired configuration. By enforcing a strict, but simplified interface to the different hardware modules the next research project would have to focus on the inherit challenges of his own design. This frees the next researcher from the implementation challenges put forth by the 3rd party vendors, leaving him to concentrate on his project alone.

Although none of the techniques involved are technologically groundbreaking by themselves, this thesis has put them together allowing the next project to test new configurations and parameters in a rapid and more accurate fashion. This will help the future researchers to increase their productivity, producing more results, one of which might be the next breakthrough in Chip Multiprocessors.

6.2 Future Work

To be able to better benchmark the design, a benchmark suite should be investigated. However, most benchmark suites requires a underlying port of the standard C library. To be able to get the standard C library working in the design a suitable operating system should be ported, this in order to provide the library with the required system calls. Some operating systems already has been ported to the MicroBlaze platform, but none of them have support for MicroBlaze in a symmetric multiprocessing environment. One idea could have been to port the embedded version of Linux to support a multicore MicroBlaze solution. Having a operating system would give the project access to a wider array of benchmarks and applications.

One of the more challenging aspects of designing the hardware was that situations impossible to predict in the simulator arose runtime on the FPGA. Without any possibility to analyze the internal state of FPGA itself, some bugs took a lot of resources locating. Xilinx has an logic analyzer, ChipScope, which can be integrated into a design, probing and returning the information found. However, it was not possible to connect to the JTAG interface at BenERA motherboard using only software, a specialized cable was needed. To help ease the development of future hardware, an solution where one connects to the JTAG interface using ChipScope would lead to an decrease in time spent in verifying stage.

One of the major changes would be to upgrade the system to use a more modern version of the Virtex FPGA used in this thesis. A newer FPGA will give the system more available BlockRAM, which is strongly advisable if the design is to incorporate more than two cores and a small shared level two cache. An modern FPGA such as the Virtex-II Pro would open for the possibility of using a newer version of the MicroBlaze core than the MicroBlaze version 4. The newer versions includes user defined registers which would have allowed the design to skip the CPU ID coprocessor designed, its sole task responding on requests for each core's id.

An future expansion would be to integrate the design into an already existing software based simulator such as the SimpleScalar or M5 simulator. To be able to do so, the design would have to be automatically generated from the configuration files. The current design allows for easy customization, configuring only needed at the top level. Thus it would be possible to automate the creation of the hardware itself. However, the challenge would lie in gathering statistics from the system. There exists no method of extracting statistics in the current design. It is, however, two approaches for information gathering from the hardware. The first would be to use the earlier mentioned ChipScope to probe the wanted signals, using an external application to interpret the data gathered. This would lead to a fully customizable method, letting the designer worry about only designing the hardware and afterwards attaching the test probes. The latter option of gathering statistics would be to design a subsystem connected to either the peripheral bus or the fast simplex link. This module could have had an shared bus which through the different components on the chip could send statistics. This is a model loosely based on the workings of the JTAG interface.

Bibliography

[1] Ben Ames. NASA backs quantum computing claim, September 2007. http://www.itworld.com/Tech/3494/070309nasaquantum/index.html.

[2] Arvind, Krste Asanovic, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Mark Oskin, David Patterson, Jan Rabaey, and John Wawrzynek. Project summary for nsf solicitation 04-588 cri: Ramp: Research accelerator for multiple processors - a community vision for a shared experimental parallel hw/sw platform. Technical report, 2005.

[3] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[4] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[5] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *International Symposium on Microarchitecture*, pages 191–201, 2000.

[6] Jichuan Chang and Gurindar S. Sohi. Cooperative Caching for Chip Multiprocessors. *SIGARCH Comput. Archit. News*, 34(2):264–276, 2006.

[7] Mikael Collin, Raimo Haukilahti, Mladen Nikitovic, and Joakim Adomat. SoCrates - A Multiprocessor SoC in 40 days. In *Conference on Design, Automation and Test in Europe 2001, Designer*, Munich, Germany, March 2001.

[8] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[9] International Business Machines Corp. IBM Unleashes World's Fastest Chip in Powerful New Computer. http://www-03.ibm.com/press/us/en/pressrelease/21580.wss, May 2007.

[10] Intel Corporation. Intel First to Demonstrate Working 45nm Chips. http://www.intel.com/pressroom/archive/releases/20060125comp.htm, January 2006.

[11] Haakon Dybdahl, Per Stenström, and Lasse Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *HiPC*, pages 22–34, 2006.

[12] S. Ghiasi and D. Grunwald. Aide de camp: Asymmetric dual core design for power and energy reduction. *Technical Report CU-CS-964-03*, May 2003. Department of Computer Science, University of Colorado, Boulder.

[13] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C. Valentine. The Intel® Pentium® M Processor: Microarchitecture and Perfomance . *IntelT̊echnology Journal*, 02, 2003.

[14] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, March 2000.

[15] John L. Hennessy and David A. Patterson. *"Computer Architecture A Quantative Approach, 3rd Edition"*. Morgan Kaufman, San Mateo, California, 2003.

[16] Ken Hines and Gaetano Borriello. Dynamic communication models in embedded system co-simulation. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 395–400, New York, NY, USA, 1997. ACM Press.

[17] IBM. The CoreConnect Bus Architecture White Paper. "`http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569910050C0FB`, January 1999.

[18] Ahmed Jeraya and Wayne Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, 2004.

[19] John L. Henning and SPEC CPU Subcommittee. SPEC CPU2006 Benchmark Descriptions . *ACM SIGARCH newsletter, Computer Architecture News*, 34, 2006.

[20] Changkyu Kim, Doug Burger, and Stephen W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *IEEE Micro*, 23(6):99–107, 2003.

[21] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multicore architectures: Understanding mechanisms, overheads and scaling. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 408–419, Washington, DC, USA, 2005. IEEE Computer Society.

[22] Prathap Muthana, Madhavan Swaminathan, Rao Tummala, Venkatesh Sundaram, Lixi Wan, S.K Bhattacharya, and P.M. Raj. Packaging of multi-core microprocessors: Tradeoffs and potential solutions. In *Electronic Components and Technology Conference, 2005. Proceedings. 55th*, volume 2, pages 1895 – 1903. IEEE, 2005.

[23] Nallatech. FUSE C/C++ API Developers Guide. `www.es.ele.tue.nl/mininoc/doc/fuse_api.pdf`, July 2002. Document Number: NT107-0068.

[24] Nallatech. Compact PCI DIME-II Motherboard. `http://www.nallatech.com/mediaLibrary/images/english/1865.pdf`, April 2007.

[25] Lasse Nattvig. NCAR NTNU Computer Architecture Research Group. . . improving multicore memory systems. `http://ncar.idi.ntnu.no`.

[26] NetBSD. NetBSD/macppc. `http://www.netbsd.org/Ports/macppc/`. Last visited: Mars, 2007.

[27] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[28] Kunle Oluktun, Mark Horowitz, and Monica Lam. Flexible architecture for simulation and testing (fast), 2003. Last Visited: 5. June 2007.

[29] OpenBSD. OpenBSD/macppc. `http://www.openbsd.org/macppc/`. Last visited: Mars, 2007.

[30] David A. Patterson. Computer Science Education in the 21st Century. *Commun. ACM*, 49(3):27–30, 2006.

[31] penguinppc.org. Introduction to PowerPC Linux. `http://penguinppc.org/about/intro.php#hardware`. Last visited: Mars, 2007.

[32] Jr. Philip Enslow. Multiprocessor Organization - a Survey. *ACM Comput. Surv.*, 9(1):103–129, 1977.

[33] Gordon D. Robinson. Why 1149.1 (jtag) really works. *Electro/94 International. Conference Proceedings. Combined Volumes*, pages 749–754, 1994.

[34] Robert R. Schaller. Moore's law: past, present, and future. *IEEE Spectr.*, 34(6):52–59, 1997.

[35] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[36] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[37] S. W. Song, J. D. Zheng, and W. B. Gardner. Prototyping a Residential Gateway Using Xilinx ISE. In *IEEE International Workshop on Rapid System Prototyping*, pages 267–269. IEEE Computer Society, 2005.

[38] Lawrence Spracklen and Santosh G. Abraham. Chip multithreading: Opportunities and challenges. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 248–252, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Richard M. Stallman. *Using and Porting the GNU Compiler Collection, For GCC Version 2.95*. Free Software Foundation, 1999.

[40] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, page 199, Washington, DC, USA, 2002. IEEE Computer Society.

[41] John Wawrzynek, Mark Oskin, Christoforos Kozyrakis, Derek Chiou, David A. Patterson, Shih-Lien Lu, James C. Hoe, and Krste Asanovic. Ramp: A research accelerator for multiple processors. Technical Report UCB/EECS-2006-158, EECS Department, University of California, Berkeley, November 24 2006.

[42] Xilinx. Xilinx Virtex-E 1.8V Field Programmable Gate Arrays Data Sheet. `http://inst.eecs.berkeley.edu/~cs150/Documents/virtexE-datasheet.pdf#search=%22VirtexE%22`, July 2002. Revision: 2.4.

[43] Xilinx. PicoBlaze 8-bit Embedded Microcontroller User Guide . `http://www.xilinx.com/bvdocs/userguides/ug129.pdf`, November 2005. Revision: 1.1.1.

[44] Xilinx. Embedded System Tools Reference Manual, Embedded Development Kit, EDK 8.2i. `http://www.xilinx.com/ise/embedded/est_rm.pdf`, June 2006. Revision: v6.0.

[45] Xilinx. MicroBlaze Processor Reference Guide. `http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf`, September 2006. Revision: 7.0.

[46] Xilinx. Virtex-5 Family Overview - LX and LXT Platforms. `http://direct.xilinx.com/bvdocs/publications/ds100.pdf`, May 2007. Revision: 3.1.

Appendix A

Benchmarks

A.1 BogoMIPS kernel

```c
#include "utils.h"

int main(){
  int i = 0 ;
  int x;

  for ( i = 0 ; i < 2; i++ ){
    x++ ;
  }
  halt();
  return 0;
}

void halt(){
  asm("addi R1, R0, 0x70000000\n\t\
  addi R11, R0, 0x7FFFFFFF\n\t\
  sw  R1, R0, R11\n\t");
}
```

A.2 Sample Linker script

```
ENTRY(main)

MEMORY
{
    mm : ORIGIN = 0x00000000, LENGTH = 0x00001000
}

SECTIONS {
    . = 0x0 ;
    .text : { *(.text) } > mm
    .rodata : { *(.text) } > mm
    .bss   : { *(.bss) } > mm
    .data : { *(.data) } > mm
}
```

A.3 Sample Makefile

```
APPNAME = for
GCC    = mb-gcc
LD   = mb-ld
LSCRIPT = linker.ld

all: ${APPNAME}

${APPNAME}:  ${APPNAME}.o
	${LD} -p --oformat=binary -o $@ ${LSCRIPT}  $^ /opt/EDK/gnu/microblaze/lin/lib/gcc/microblaze/3.4.1/libgcc.a

%.o: %.s
	${GCC} -c $^

%.s: %.c
	${GCC} -S $^


clean:
	rm *.o *.s ${APPNAME}
```

A.4 LoadStore kernel

```
.text

_start: .global _start
  .global _main

  addi  R11, R11, 0xFFFFFFFF

  addi  R2, R0, fnord // Address lookup.
  addi  R1, R0, 0     // Counter variable.

                // Change here to set the iterations.
  addi  R7, R0, 0x1   // Check the 255 * 4 bytes after fnord.

  add   R3, R0, R7    // Saves one imm instruction during the reset op.

storeLoop:
  sw    R11,  R2, R0
  addi  R1,   R1, 0x4
  add   R2,   R1, R2
   cmp    R4,   R3, R1
   blei R4,   storeLoop


  // Reset...
  addi  R2, R0, fnord // Address lookup.
  addi  R1, R0, 0     // Counter variable.
  add   R3, R0, R7    // ^-- See explanation above.

loadLoop:
    lw    R11,  R2, R0
    addi  R1,   R1, 0x4
    add   R2,   R1, R2
    cmp   R4,   R3, R1
    blei  R4,   loadLoop

terminate:
  addi R11, R0, 0x7FFFFFFF
  sw  R11, R0, R11

fnord:
```

Appendix B

Hardware

B.1 Core OMA Interface

```
-- ------------------------------------------------------------------------------
-- user_logic.vhd - entity/architecture pair
-- ------------------------------------------------------------------------------
--
-- ***************************************************************************
-- ** Copyright (c) 1995-2006 Xilinx, Inc.  All rights reserved.            **
-- **                                                                       **
-- ** Xilinx, Inc.                                                          **
-- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"         **
-- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND       **
-- ** SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,        **
-- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,        **
-- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION           **
-- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,     **
-- ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE      **
-- ** FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY              **
-- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
```

```
**
-- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR **
-- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF **
-- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS **
-- ** FOR A PARTICULAR PURPOSE. **
-- ** **
-- ***************************************************************************
--
-- ---------------------------------------------------------------------------
-- Filename:          user_logic.vhd
-- Version:           1.00.a
-- Description:       User logic.
-- Date:              Thu Feb  8 12:34:55 2007 (by Create and Import Peripheral Wizard)
-- VHDL Standard:     VHDL'93
-- ---------------------------------------------------------------------------
-- Naming Conventions:
--   active low signals:                     "*_n"
--   clock signals:                          "clk", "clk_div#", "clk_#x"
--   reset signals:                          "rst", "rst_n"
--   generics:                               "C_*"
--   user defined types:                     "*_TYPE"
--   state machine next state:               "*_ns"
--   state machine current state:            "*_cs"
--   combinatorial signals:                  "*_com"
--   pipelined or register delay signals:    "*_d#"
--   counter signals:                        "*cnt*"
--   clock enable signals:                   "*_ce"
--   internal version of output port:        "*_i"
--   device pins:                            "*_pin"
--   ports:                                  "- Names begin with Uppercase"
--   processes:                              "*_PROCESS"
--   component instantiations:               "<ENTITY>I_<#|FUNC>"
-- ---------------------------------------------------------------------------

-- DO NOT EDIT BELOW THIS LINE --------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
```

```
-- DO NOT EDIT ABOVE THIS LINE ---------------------

--USER libraries added here

-- ------------------------------------------------------------------------------
-- Entity section
-- ------------------------------------------------------------------------------
-- Definition of Generics:
--   C_AWIDTH                    -- User logic address bus width
--   C_DWIDTH                    -- User logic data bus width
--   C_NUM_CE                    -- User logic chip enable bus width
--
-- Definition of Ports:
--   Bus2IP_Clk                  -- Bus to IP clock
--   Bus2IP_Reset                -- Bus to IP reset
--   Bus2IP_Addr                 -- Bus to IP address bus
--   Bus2IP_Data                 -- Bus to IP data bus for user logic
--   Bus2IP_BE                   -- Bus to IP byte enables for user logic
--   Bus2IP_RNW                  -- Bus to IP read/not write
--   Bus2IP_RdCE                 -- Bus to IP read chip enable for user logic
--   Bus2IP_WrCE                 -- Bus to IP write chip enable for user logic
--   IP2Bus_Data                 -- IP to Bus data bus for user logic
--   IP2Bus_Ack                  -- IP to Bus acknowledgement
--   IP2Bus_Retry                -- IP to Bus retry response
--   IP2Bus_Error                -- IP to Bus error response
--   IP2Bus_ToutSup              -- IP to Bus timeout suppress
-- ------------------------------------------------------------------------------

entity user_logic is
  generic
  (
    -- ADD USER GENERICS BELOW THIS LINE ---------------
    --USER generics added here
    -- ADD USER GENERICS ABOVE THIS LINE ---------------

    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol parameters, do not add to or delete
    C_AWIDTH                       : integer              := 32;
    C_DWIDTH                       : integer              := 32;
    C_NUM_CE                       : integer              := 2
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );
  port
  (
    -- ADD USER PORTS BELOW THIS LINE ------------------
      --USER ports added here
  -- The data I send out off the chip
  OMA_DATA_O              : out std_logic_vector(0 to C_DWIDTH-1);
```

```
  -- The data I get in from outside the chip.
  OMA_DATA_I            : in std_logic_vector(0 to C_DWIDTH-1) := (others => '0');
  -- If the data on OMA_DATA_* is the data you requested.
  OMA_VALID            : in std_logic := '0';
  -- Which addr to operate on.
  OMA_ADDRESS           : out std_logic_vector(0 to C_AWIDTH-1);
  -- Read or write. On asserted we read.
  OMA_RW               : out std_logic;
  -- If there is a request pending.
  OMA_ACTIVE            : out std_logic;
  -- Leds debug
  LEDS_DEBUG            : out std_logic_vector( 0 to 15 );
  OMA_BE               : out std_logic_vector(0 to C_DWIDTH/8-1);

    -- ADD USER PORTS ABOVE THIS LINE ------------------

    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol ports, do not add to or delete
    Bus2IP_Clk                 : in  std_logic;
    Bus2IP_Reset               : in  std_logic;
    Bus2IP_Addr                : in  std_logic_vector(0 to C_AWIDTH-1);
    Bus2IP_Data                : in  std_logic_vector(0 to C_DWIDTH-1);
    Bus2IP_BE                  : in  std_logic_vector(0 to C_DWIDTH/8-1);
    Bus2IP_RNW                 : in  std_logic;
    Bus2IP_RdCE                : in  std_logic_vector(0 to C_NUM_CE-1);
    Bus2IP_WrCE                : in  std_logic_vector(0 to C_NUM_CE-1);
    IP2Bus_Data                : out std_logic_vector(0 to C_DWIDTH-1);
    IP2Bus_Ack                 : out std_logic;
    IP2Bus_Retry               : out std_logic;
    IP2Bus_Error               : out std_logic;
    IP2Bus_ToutSup             : out std_logic
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );
end entity user_logic;

-- ------------------------------------------------------------------------------

-- Architecture section
-- ------------------------------------------------------------------------------

architecture IMP of user_logic is
 constant zeros      : std_logic_vector(0 to C_NUM_CE-1) := (others => '0');
 type STATE is ( IDLE, READING, WRITING, RCOMP, WCOMP );

 signal writeEnabled   : std_logic := '0';
 signal readEnabled    : std_logic := '0';
 signal clock        : std_logic;
 signal iState         : STATE := IDLE;
 signal leds         : std_logic_vector(0 to 15) := (others => '1');

begin
```

```
-- My own signals.
writeEnabled  <= (Bus2IP_WrCE(0) or Bus2IP_WrCE(1) ) and not Bus2IP_RNW ;
readEnabled   <= (Bus2IP_RdCE(0) or Bus2IP_RdCE(1) ) and Bus2IP_RNW ;

clock        <= Bus2IP_Clk;

-- Let some signals out independently of anything else.
OMA_ADDRESS   <= Bus2IP_Addr;
OMA_DATA_O    <= Bus2IP_Data;
OMA_BE       <= Bus2IP_BE;

LEDS_DEBUG    <= leds;

-- Everything on time.
INN: process(clock)
begin
  if rising_edge(clock) then
    -- Default sending out 0x00 on the databus
    -- to the core.
    IP2Bus_Data <= (others => '0');
    IP2Bus_Ack <= '0';

    case iState is
      when IDLE =>
        if readEnabled = '1' then
          OMA_ACTIVE <= '1';
          OMA_RW <= '1';
          IP2Bus_Data <= OMA_DATA_I;
          iState <= READING;
        elsif writeEnabled = '1' then
          OMA_ACTIVE <= '1';
          OMA_RW <= '0';
          iState <= WRITING;
        end if;

      when READING =>
      -- We will use more than 16 cycles.
        IP2Bus_ToutSup      <= '1';
        IP2Bus_Data <= OMA_DATA_I;

        -- Check for doneness.
        if OMA_VALID = '1' then
          iState <= RCOMP;
          OMA_ACTIVE <= '0';
          IP2Bus_ToutSup     <= '0';
          IP2Bus_Ack        <= '1';
          leds            <= OMA_DATA_I(0 to 15);
        end if;

      when WRITING =>
      -- We will use more than 16 cycles.
        IP2Bus_ToutSup      <= '1';
      -- Check for doneness.
        if OMA_VALID = '1' then
          iState <= WCOMP;
          OMA_ACTIVE <= '0';
```

```
            IP2Bus_ToutSup      <= '0';
            IP2Bus_Ack       <= '1';
          end if;

--          when RCOMP =>
--          when WCOMP =>
        when others =>
          OMA_ACTIVE  <= '0';
          IP2Bus_Ack  <= '0';
          iState <= IDLE;
      end case;

    end if;
  end process;

  IP2Bus_Error       <= '0';
  IP2Bus_Retry       <= '0';

end IMP;
```

B.2 CPU Identifier

```
-- ------------------------------------------------------------------------------
-- cpu_identifier - entity/architecture pair
-- ------------------------------------------------------------------------------
--
-- ***************************************************************************
-- ** Copyright (c) 1995-2006 Xilinx, Inc.  All rights reserved.            **
-- **                                                                       **
-- ** Xilinx, Inc.                                                          **
-- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"         **
-- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND       **
-- ** SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,        **
-- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,        **
-- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION           **
-- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,     **
-- ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE      **
-- ** FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY              **
-- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE               **
-- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR        **
-- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF       **
-- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS       **
-- ** FOR A PARTICULAR PURPOSE.                                             **
-- **                                                                       **
-- ***************************************************************************
--
-- ------------------------------------------------------------------------------
-- Filename:          cpu_identifier
-- Version:           1.00.a
```

```vhdl
-- Description:          Example FSL core (VHDL).
-- Date:                 Wed May  2 13:52:15 2007 (by Create and Import Peripheral Wizard)
-- VHDL Standard:        VHDL'93
-- ------------------------------------------------------------------------------
-- Naming Conventions:
--   active low signals:                    "*_n"
--   clock signals:                         "clk", "clk_div#", "clk_#x"
--   reset signals:                         "rst", "rst_n"
--   generics:                              "C_*"
--   user defined types:                    "*_TYPE"
--   state machine next state:              "*_ns"
--   state machine current state:           "*_cs"
--   combinatorial signals:                 "*_com"
--   pipelined or register delay signals:   "*_d#"
--   counter signals:                       "*cnt*"
--   clock enable signals:                  "*_ce"
--   internal version of output port:       "*_i"
--   device pins:                           "*_pin"
--   ports:                                 "- Names begin with Uppercase"
--   processes:                             "*_PROCESS"
--   component instantiations:              "<ENTITY_>I_<#|FUNC>"
-- ------------------------------------------------------------------------------

library ieee;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- ------------------------------------------------------------------------------------
--
--
-- Definition of Ports
-- FSL_Clk             : Synchronous clock
-- FSL_Rst             : System reset, should always come from FSL bus
-- FSL_S_Clk       : Slave asynchronous clock
-- FSL_S_Read      : Read signal, requiring next available input to be read
-- FSL_S_Data      : Input data
-- FSL_S_CONTROL   : Control Bit, indicating the input data are control word
-- FSL_S_Exists    : Data Exist Bit, indicating data exist in the input FSL bus
-- FSL_M_Clk       : Master asynchronous clock
-- FSL_M_Write     : Write signal, enabling writing to output FSL bus
-- FSL_M_Data      : Output data
-- FSL_M_Control   : Control Bit, indicating the output data are contol word
-- FSL_M_Full      : Full Bit, indicating output FSL bus is full
--
-- -------------------------------------------------------------------------------

```

```
-- ------------------------------------------------------------------------
-- Entity Section
-- ------------------------------------------------------------------------

entity cpu_identifier is
  generic (
    CPU_ID  : integer := 32
  );
  port
  (
    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol ports, do not add or delete.
    FSL_Clk : in  std_logic;
    FSL_Rst : in  std_logic;
    FSL_S_Clk : out std_logic;
    FSL_S_Read  : out std_logic;
    FSL_S_Data  : in  std_logic_vector(0 to 31);
    FSL_S_Control : in  std_logic;
    FSL_S_Exists  : in  std_logic;
    FSL_M_Clk : out std_logic;
    FSL_M_Write : out std_logic;
    FSL_M_Data  : out std_logic_vector(0 to 31);
    FSL_M_Control : out std_logic;
    FSL_M_Full  : in  std_logic
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );

attribute SIGIS : string;
attribute SIGIS of FSL_Clk : signal is "Clk";
attribute SIGIS of FSL_S_Clk : signal is "Clk";
attribute SIGIS of FSL_M_Clk : signal is "Clk";

end cpu_identifier;

-- ------------------------------------------------------------------------
-- Architecture Section
-- ------------------------------------------------------------------------

architecture EXAMPLE of cpu_identifier is

   -- Total number of input data.
   constant NUMBER_OF_INPUT_WORDS  : natural := 1;

   -- Total number of output data
   constant NUMBER_OF_OUTPUT_WORDS : natural := 1;

begin
   FSL_M_Write <= not FSL_M_Full;
```

```

    FSL_M_Data <= CONV_STD_LOGIC_VECTOR(CPU_ID, FSL_M_Data'length);

end architecture EXAMPLE;
```

B.3 RAM Block

```
-- ----------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    10:39:13 03/22/2007
-- Design Name:
-- Module Name:    myRam - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- ----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity myRam is
  GENERIC(
    DATA_WIDTH    : natural := 32;
    ADDR_LENGTH   : natural := 8
    );
  PORT(
    clock         : IN std_logic := '0';
    reset         : IN std_logic := '0';
    enable          : IN std_logic := '0';
    writeEnabled      : IN std_logic := '0';
    address         : IN std_logic_vector(0 to ADDR_LENGTH-1) := (others => '0');
    dataIn          : IN std_logic_vector(0 to DATA_WIDTH-1)  := (others => '0');
    metaIn          : IN std_logic_vector(0 to DATA_WIDTH-1)  := (others => '0');
    dataOut         : OUT std_logic_vector(0 to DATA_WIDTH-1);
    metaOut         : OUT std_logic_vector(0 to DATA_WIDTH-1)
  );
end myRam;

```

```
architecture Behavioral of myRam is
  constant numBlocks : integer := 2 ** (ADDR_LENGTH - 7);

  subtype Data is std_logic_vector(0 to DATA_WIDTH-1);

  type DOut        is array(0 to numBlocks-1) of std_logic_vector(0 to 15);

  signal enabled    : std_logic_vector(0 to numBlocks-1);
  signal WEnabled   : std_logic_vector(0 to numBlocks-1);

  signal DoutA     : DOut;
  signal DoutB     : DOut;
  signal MoutA     : DOut;
  signal MoutB     : DOut;
  signal addrA     : std_logic_vector(0 to 7);
  signal addrB     : std_logic_vector(0 to 7);
begin

  mem: for I in 0 to numBlocks - 1 generate
  begin
    MetaData: RAMB4_S16_S16
      generic map (
       INIT_00 => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_01 => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
       INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
      port map (
       DOA => MoutA(I),           -- Port A 16-bit data output
```

```
      DOB => MoutB(I),              -- Port B 16-bit data output
      ADDRA => addrA,               -- Port A 8-bit address input
      ADDRB => addrB,               -- Port B 8-bit address input
      CLKA => clock,                -- Port A clock input
      CLKB => clock,                -- Port B clock input
      DIA => metaIn(0 to 15),       -- Port A 16-bit data input
      DIB => metaIn(16 to 31),      -- Port B 16-bit data input
      ENA => enabled(I),            -- Port A RAM enable input
      ENB => enabled(I),            -- Port B RAM enable input
      RSTA => reset,                -- Port A Synchronous reset input
      RSTB => reset,                -- Port B Synchronous reset input
      WEA => WEnabled(I),           -- Port A RAM write enable input
      WEB => WEnabled(I)            -- Port B RAM write enable input
     );
   DataMem: RAMB4_S16_S16
     generic map (
      INIT_00 => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_01 => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
      INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
     port map (
      DOA => DoutA(I),              -- Port A 16-bit data output
      DOB => DoutB(I),              -- Port B 16-bit data output
      ADDRA => addrA,               -- Port A 8-bit address input
      ADDRB => addrB,               -- Port B 8-bit address input
      CLKA => clock,                -- Port A clock input
      CLKB => clock,                -- Port B clock input
      DIA => dataIn(0 to 15),       -- Port A 16-bit data input
```

```vhdl
        DIB => dataIn(16 to 31),      -- Port B 16-bit data input
        ENA => enabled(I),          -- Port A RAM enable input
        ENB => enabled(I),          -- Port B RAM enable input
        RSTA => reset,            -- Port A Synchronous reset input
        RSTB => reset,            -- Port B Synchronous reset input
        WEA => WEnabled(I),           -- Port A RAM write enable input
        WEB => WEnabled(I)        -- Port B RAM write enable input
      );
  end generate;

  lines: for I in 0 to numBlocks -1 generate
    WEnabled(I) <= writeEnabled and enabled(I);
  end generate;

  clocked: process( clock, enabled,writeEnabled, address,dataIn ) is
  begin
    addrA(0) <= '0';
    addrB(0) <= '1';
    addrA(1 to 7) <= address(0 to 6);
    addrB(1 to 7) <= address(0 to 6);
    -- decodedAddr
    if rising_edge(clock) then
      enabled   <= (others => '0');

      if enable = '1' then
        if numBlocks > 1 then
            enabled(conv_integer(address(7 to 7+(ADDR_LENGTH-8)))) <= '1';
        else
            enabled(0) <= '1';
        end if;

      end if;

      -- Multiplex the data out.
      if numBlocks > 1 then
        metaOut(0 to 15)  <= MoutA(conv_integer(address(7 to 7+(ADDR_LENGTH-8))));
        metaOut(16 to 31)   <= MoutB(conv_integer(address(7 to 7+(ADDR_LENGTH-8))));
        dataOut(0 to 15)  <= DoutA(conv_integer(address(7 to 7+(ADDR_LENGTH-8))));
        dataOut(16 to 31)   <= DoutB(conv_integer(address(7 to 7+(ADDR_LENGTH-8))));
      else
        metaOut(0 to 15)  <= MoutA(0);
        metaOut(16 to 31)   <= MoutB(0);
        dataOut(0 to 15)  <= DoutA(0);
        dataOut(16 to 31)   <= DoutB(0);
      end if;

    end if;
  end process;

end Behavioral;
```

B.4 Cache Block

```
--------------------------------------------------------------------------------
--
-- Create Date:    12:02:44 03/29/2007
-- Design Name:
-- Module Name:    cacheBlock - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
-- Important to notice that all data coming out of the microBlaze core
-- is bit reversed. Hence the index and tag of the address is switched
-- compared to the normal way of doing it.
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity cacheBlock is
  generic(
    DATA_WIDTH        : integer := 32;
    ADDRESS_WIDTH     : integer := 32;  --Tag length is independent of the SET_A
    SET_ASSOCIATIVITY : integer := 2; -- be able to create different caches, not
    TAG_LENGTH        : integer := 25;  -- just set associative ones.
    ADDR_LENGTH       : integer := 7+1;

    LRU_SIZE        : integer := 2
  );

  port (
    CLOCK         : IN std_logic;
    RESET         : IN std_logic;
    IN_OMA_VALID      : OUT std_logic;
    IN_OMA_DATA_I     : OUT std_logic_vector(0 to 31);
    IN_OMA_DATA_O     : IN std_logic_vector(0 to 31);
    IN_OMA_ADDRESS      : IN std_logic_vector(0 to 31);
    IN_OMA_BE       : IN std_logic_vector(0 to 3);
    IN_OMA_RW       : IN std_logic;
```

```
      IN_OMA_ACTIVE    : IN std_logic;

      OUT_OMA_VALID    : IN std_logic;
      OUT_OMA_DATA_I    : IN std_logic_vector(0 to 31);
      OUT_OMA_DATA_O    : OUT std_logic_vector(0 to 31);
      OUT_OMA_ADDRESS   : OUT std_logic_vector(0 to 31);
      OUT_OMA_BE       : OUT std_logic_vector(0 to 3);
      OUT_OMA_RW       : OUT std_logic;
      OUT_OMA_ACTIVE    : OUT std_logic
   );
end cacheBlock;

architecture Behavioral of cacheBlock is
   type dlines is array (0 to SET_ASSOCIATIVITY -1) of
      std_logic_vector(0 to DATA_WIDTH-1);

   type STATE is ( IDLE, FIND_VICTIM, EVICT, GET_DATA, OVERRIDE, DONE );

   constant tagNotFound  : std_logic_vector(0 to SET_ASSOCIATIVITY -1) := (
       others => '0');

   -- Which parts shall we address, and which are left for tags.
   constant setAddrBase  : integer := ADDRESS_WIDTH - ADDR_LENGTH;
   constant lruBase     : integer := 0; -- Base addr of LRU;
   constant lruSize     : integer := 1; -- Log2(SET_ASSOCIATIVITY);
   constant dirtyBase    : integer := lruBase + lruSize;
   constant validBase    : integer := dirtyBase + 1;
   constant tagBase     : integer := validBase + 1;
   constant tagSize     : integer := setAddrBase;

   -- Control signals.
   signal enabled       : std_logic_vector(0 to SET_ASSOCIATIVITY - 1);
   signal writeEnabled    : std_logic_vector(0 to SET_ASSOCIATIVITY - 1);
   signal tagFound      : std_logic_vector(0 to SET_ASSOCIATIVITY - 1);

   -- Cache state
   signal readState     : STATE := IDLE;
   signal writeState     : STATE := IDLE;

   -- Set up the signals to be multiplexed.
   signal dout         : dlines := (others => (others => '0'));
   signal mout         : dlines := (others => (others => '0'));
   signal din          : dlines := (others => (others => '0'));
   signal min          : dlines := (others => (others => '0'));

   -- register the signal to meet timing constraints.
   signal dataIn       : std_logic_vector(0 to IN_OMA_DATA_O'length -1);

   -- BlockRAM needs delay.
   signal delay        : std_logic_vector(0 to 1) := "00";
   signal inValid       : std_logic := '0';

   impure function findVictim return integer is
      variable minValue    : integer := 90; --Change this if you have set
           associative > 90
      variable minPos     : integer := 0;
```

```
      variable current  : integer := 0;
    begin
      for I in 0 to SET_ASSOCIATIVITY -1 loop
        if mout(I)(validBase) = '0' then
          return I;
        else
          current := conv_integer(mout(I)(lruBase to lruBase + lruSize - 1));
          if current < minValue then
            minValue := current;
            minPos := I;
          end if;
        end if;
      end loop;

      return minPos;
    end function;

    function lruReduction(A:std_logic_vector) return integer is
      variable x : integer ;
    begin
      x := conv_integer(A);
      if x >= 0 then
        return x;
      else return 0;
      end if;
    end function;
begin
    -- Generate the storage blocks.
    setBlock: for I in 0 to SET_ASSOCIATIVITY - 1 generate
    begin
      cacheBlock: entity work.myRam
        generic map(
          DATA_WIDTH   => DATA_WIDTH,
          ADDR_LENGTH => ADDR_LENGTH
        )
        port map(
          clock        => CLOCK,
          reset        => RESET,
          enable        => enabled(I),
          writeEnabled  => writeEnabled(I),
          address       => IN_OMA_ADDRESS(IN_OMA_ADDRESS'length - ADDR_LENGTH to
              IN_OMA_ADDRESS'length -1 ),
          dataIn        => din(I),
          metaIn        => min(I),
          dataOut       => dout(I),
          metaOut       => mout(I)
        );
    end generate;

    -- Check if we've found something.
    -- update the tagFound signal.
    fnd: process(CLOCK,IN_OMA_ADDRESS,mout) is
    begin
      for I in 0 to SET_ASSOCIATIVITY -1 loop
        if (mout(I)(tagBase to tagBase + tagSize -1) = IN_OMA_ADDRESS(0 to
            IN_OMA_ADDRESS'length - ADDR_LENGTH- 1))
```

```
        and mout(I)(validBase) = '1'
      then
        tagFound(I) <= '1';
      else
        tagFound(I) <= '0';
      end if;
    end loop;
  end process;

  -- The main function.
  main: process(CLOCK, IN_OMA_ACTIVE) is
    variable victim : natural := 0;

    -- Function decl.
    procedure updateLRU is
    begin
      for I in 0 to SET_ASSOCIATIVITY - 1 loop
        -- Copy back, rude and elementary. From the output to the input...
        min(I) <= mout(I);

        -- Update only the LRU.
        min(I)(lruBase to lruBase + lruSize -1) <= CONV_STD_LOGIC_VECTOR(
                lruReduction(mout(I)(lruBase to lruBase + lruSize -1)),
                lruSize);
      end loop;
    end updateLRU;

    procedure evictLine is
    begin
      -- If victim dirty
      -- dump it to memory.
      OUT_OMA_BE        <= (others => '1');
      OUT_OMA_RW        <= '0';
      OUT_OMA_ACTIVE    <= '1';
      OUT_OMA_DATA_O    <= dout(victim);
      -- The address..

      OUT_OMA_ADDRESS(IN_OMA_ADDRESS'length - ADDR_LENGTH to IN_OMA_ADDRESS'length -1 ) <=
        IN_OMA_ADDRESS( IN_OMA_ADDRESS'length - ADDR_LENGTH to IN_OMA_ADDRESS'length -1);
      -- Get the tag from the metadata.
      OUT_OMA_ADDRESS(0 to OUT_OMA_ADDRESS'length - ADDR_LENGTH- 1 ) <= mout(victim)(tagBase to tagBase + tagSize - 1);
    end evictLine;

    procedure doWriteHit is
    begin
      updateLRU;
      min(victim)(validBase) <= '1';
      min(victim)(dirtyBase) <= '1';
      min(victim)(lruBase to lruSize -1) <= (others => '1');
      min(victim)(tagBase to tagBase + tagSize -1) <= mout(victim)(tagBase to tagBase+tagSize-1);
      din(victim)          <= dataIn;
      writeEnabled(victim)  <= '1';
```

```

    inValid <= '1';
  end doWriteHit;

  procedure doReadHit is
  begin
    updateLRU;
    for I in 0 to SET_ASSOCIATIVITY -1 loop
      if ( tagFound(I) = '1' ) then
        IN_OMA_DATA_I <= dout(I);
        exit;
      end if;
    end loop;

    inValid <= '1';
  end doReadHit;

  procedure doReadMiss is
  begin
    -- 1) Find victim
    -- 2) Evict
    -- 3) Get data from memory
    -- 4) Override
    case readState is
      when IDLE =>
        readState <= FIND_VICTIM;
      when FIND_VICTIM =>
        victim := findVictim;
        readState <= EVICT;
      when EVICT =>
        if OUT_OMA_VALID = '1' then
          OUT_OMA_ADDRESS <= IN_OMA_ADDRESS;
          OUT_OMA_RW      <= '1';
          OUT_OMA_ACTIVE  <= '1';
          -- Send out a new request.
          readState <= GET_DATA;
        else
          if  mout(victim)(dirtyBase) = '1' and
            mout(victim)(validBase)  = '1' then

            -- If it's dirty evict it.
            evictLine;
          else
            OUT_OMA_ADDRESS <= IN_OMA_ADDRESS;
            OUT_OMA_RW      <= '1';
            OUT_OMA_ACTIVE  <= '1';
            readState <= GET_DATA;
          end if;
        end if;

      when GET_DATA =>
        OUT_OMA_ACTIVE  <= '1';
        if ( OUT_OMA_VALID = '1' ) then

          -- Update metadata.
          min(victim)(validBase)     <= '1';
```

```
            min(victim)(dirtyBase)      <= '0';
            min(victim)(lruBase to lruSize -1) <= (others => '1');

             min(victim)(tagBase to tagBase + tagSize -1) <= IN_OMA_ADDRESS(0 to IN_OMA_ADDRESS'length - ADDR_LENGTH -1);
            din(victim)             <= OUT_OMA_DATA_I;

            -- Send stuff out.
            IN_OMA_DATA_I     <= OUT_OMA_DATA_I;

            -- writeEnabled asserted for two cycles.
            readState <= DONE;
            writeEnabled(victim)      <= '1';
          end if;
        when DONE =>
          inValid <= '1';
          writeEnabled(victim)      <= '1';
          readState <= IDLE;

        when others =>
          readState <= IDLE;
      end case;
    end doReadMiss;


    procedure doWriteMiss is
    begin
      --- 1) Find victim.
      --- 2) Evict. E.g., die bastard cache line
      --- 3) Override.

      case writeState is
        when IDLE =>
          writeState <= FIND_VICTIM;
        when FIND_VICTIM =>
          victim := findVictim;
          writeState <= EVICT;
        when EVICT =>
          if OUT_OMA_VALID = '1' then
            writeState <= OVERRIDE;
          else
            if  mout(victim)(dirtyBase) = '1' and
              mout(victim)(validBase)  = '1' then
              -- If it's dirty and valid evict it.
              evictLine;
            else
              writeState <= OVERRIDE;
            end if;
          end if;
        when OVERRIDE =>
          min(victim)(validBase) <= '1';
          min(victim)(dirtyBase) <= '1';
          min(victim)(lruBase to lruSize -1) <= (others => '1');
          min(victim)(tagBase to tagBase + tagSize -1) <= IN_OMA_ADDRESS(0 to IN_OMA_ADDRESS'length - ADDR_LENGTH -1);
          din(victim)        <= dataIn;
```

```
          writeState <= DONE;

          -- writeEnabled asserted for two clock cycles.
          writeEnabled(victim)  <= '1';
          inValid <= '1';
        when DONE =>
          writeEnabled(victim)  <= '1';
          writeState <= IDLE;

        when others =>
          writeState <= IDLE;
      end case;

    end doWriteMiss;
    -- decl. ends.

  begin
    dataIn       <= IN_OMA_DATA_O;
    IN_OMA_VALID  <= inValid;

    if rising_edge(CLOCK)  then
      inValid    <= '0';
      OUT_OMA_ACTIVE  <= '0';

      writeEnabled <= (others => '0');
      enabled    <= (others => '0');

      if ( inValid = '0' )  then
        enabled <= (others => '1');
      else
        enabled <= (others => '0');
      end if;

      if IN_OMA_ACTIVE = '1' and inValid = '0' then

        if IN_OMA_ADDRESS = x"7FFFFFFF" and IN_OMA_RW = '0' then
          -- DO NOT cache the termination signal.
          -- We're done anyways, so lets do this the
          -- dirty way.
          OUT_OMA_ADDRESS <= IN_OMA_ADDRESS;
          OUT_OMA_RW      <= '0';
          OUT_OMA_ACTIVE  <= '1';
        else
          -- We need to let the signals propagate from the block RAM to us.
          if delay = "11" or ( writeState /= IDLE or readState /= IDLE ) then
            if IN_OMA_RW = '1' then
              if tagFound /= tagNotFound then
                -- Ladies and Gentlemen, we have hit.
                doReadHit;
              else
                doReadMiss;
              end if;
            elsif IN_OMA_RW = '0' then
              if tagFound /= tagNotFound then
                doWriteHit;
```

```
              else
                doWriteMiss;
              end if;
            end if;

            delay <= "00";
          elsif delay = "00" then
            -- Block RAM need time to propagate through...
            delay <= "01";
          else
            delay <= "11";
          end if;

        end if;
      end if;
    end if;
  end process;

end Behavioral;
```

B.5 Arbiter

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    10:28:34 04/13/2007
-- Design Name:
-- Module Name:    arbiter - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

library WORK;
use WORK.coreStuff.ALL;

entity arbiter is
  port (
    CLOCK          : IN std_logic;
    RESET          : IN std_logic;

    IN_OMA_VALID       : OUT coreBit;
    IN_OMA_DATA_I      : OUT coreVector;
    IN_OMA_DATA_O      : IN  coreVector;
    IN_OMA_ADDRESS       : IN  coreVector;
    IN_OMA_BE        : IN  coreBe;
    IN_OMA_RW        : IN  coreBit;
    IN_OMA_ACTIVE      : IN  coreBit;

    OUT_OMA_VALID      : IN  std_logic;
    OUT_OMA_DATA_I       : IN  std_logic_vector(0 to 31);
    OUT_OMA_DATA_O       : OUT std_logic_vector(0 to 31);
    OUT_OMA_ADDRESS      : OUT std_logic_vector(0 to 31);
    OUT_OMA_BE         : OUT std_logic_vector(0 to 3);
```

```
    OUT_OMA_RW        : OUT std_logic;
    OUT_OMA_ACTIVE    : OUT std_logic
  );
end arbiter;

architecture Behavioral of arbiter is
  signal active      : std_logic := '0';
  shared variable currentActive : natural := 0;
begin

  f: process is
  begin
    IN_OMA_VALID   <= (others => '0');
    IN_OMA_VALID(currentActive)    <= OUT_OMA_VALID;
    IN_OMA_DATA_I(currentActive)     <= OUT_OMA_DATA_I;
    OUT_OMA_DATA_O      <= IN_OMA_DATA_O(currentActive);
    OUT_OMA_ADDRESS     <= IN_OMA_ADDRESS(currentActive);
    OUT_OMA_BE          <= IN_OMA_BE(currentActive);
    OUT_OMA_RW          <= IN_OMA_RW(currentActive);
    OUT_OMA_ACTIVE      <= IN_OMA_ACTIVE(currentActive);
  end process;

  switcher: process(CLOCK, IN_OMA_ACTIVE, OUT_OMA_VALID ) is
  begin
    if rising_edge(CLOCK) then
      -- If someone already are sending and receiving
      -- data, we'll wait for the ack to get back..
      if active = '1' then
        -- Set no active on VALID data.
        if OUT_OMA_VALID = '1' then
          active      <= '0';
        end if;


      else
        for I in 0 to numCores - 1 loop
          if IN_OMA_ACTIVE((currentActive + I) mod numCores) = '1' then
            -- Find the active one.
            currentActive      := (currentActive + I) mod numCores;
            active        <=  '1';
            exit;
          end if;
        end loop;
      end if;
    end if;
  end process;

end Behavioral;
```

B.6 PCI COM

```
-- --------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    17:14:16 11/10/2006
-- Design Name:
-- Module Name:    com - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- --------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity pciCom is
    Port (
    --- PCI FPGA Spesific.
    CLOCK      : in std_logic;
    RESET       : in std_logic;

    PCI_ADDS     : in  STD_LOGIC; -- ADIO is DATA when high, else Address.
    PCI_EMPTY    : in  STD_LOGIC; -- Is empty
    PCI_BUSY     : in  STD_LOGIC; -- Can't write quite yet.
    PCI_RW      : out  STD_LOGIC; -- Will write to the PCI FPGA on high.
    PCI_RENWEN   : out  STD_LOGIC; -- High disables communication.
    PCI_ADIO     : inout  STD_LOGIC_VECTOR (0 to 31);

    -- LED DEBUG..
    LED_DEBUG : OUT std_logic_vector(0 to 15);

    --- Internal chat with the memory bus
    OMA_VALID    : OUT std_logic;
    OMA_DATA_I   : OUT std_logic_vector(0 to 31);
    OMA_DATA_O   : IN std_logic_vector(0 to 31);
    OMA_ADDRESS : IN std_logic_vector(0 to 31);
```

```
    OMA_BE     : IN std_logic_vector(0 to 3);
    OMA_RW     : IN std_logic;
    OMA_ACTIVE : IN std_logic
    );
end pciCom;

architecture Behavioral of pciCom is
  -- State machine variables.
  type States is (  IDLE,
                 READ_START, READ_ADDR, READ_DATA, READ_IDLE,
                 WRITE_START, WRITE_ADDR, WRITE_DATA,
                 COMPLETE
               );

  -- constant signals to write out to the databus.
  constant writeSignal : std_logic_vector(31 downto 0) :=(1 => '1', others => '0');
  constant readSignal : std_logic_vector(31 downto 0) := (0 => '1', others => '0');

  -- state signals
  constant sIDLE     : std_logic_vector(0 to 15) := (0 => '0', others => '1');
  constant sREAD_START : std_logic_vector(0 to 15) := (1 => '0', others => '1');
  constant sREAD_ADDR : std_logic_vector(0 to 15) := (2 => '0', others => '1');
  constant sREAD_DATA   : std_logic_vector(0 to 15) := (3 => '0', others => '1');
  constant sREAD_IDLE : std_logic_vector(0 to 15) := (4 => '0', others => '1');
  constant sWRITE_START : std_logic_vector(0 to 15):= (5 => '0', others => '1');
  constant sWRITE_ADDR  : std_logic_vector(0 to 15) := (6 => '0', others => '1');
  constant sWRITE_DATA  : std_logic_vector(0 to 15) := (7 => '0', others => '1');
  constant sCOMPLETE  : std_logic_vector(0 to 15) := (8 => '0', others => '1');

  signal state   : States;
  signal data_o : std_logic_vector(0 to 31) := (others => '1');
  signal data_i : std_logic_vector(0 to 31) := (others => '1');
  signal address  : std_logic_vector(0 to 31) := (others => '1');

  signal clockCount : std_logic_vector(0 to 31) := (others => '0');

begin

  countClock: process(CLOCK,RESET, state) is
  begin
    if rising_edge(CLOCK) then
      if RESET = '1' or state = WRITE_START or state = READ_START then
        clockCount <= (others => '0');
      else
        clockCount <= clockCount + 1;
```

```vhdl
      end if;
    end if;
  end process;

  fnord: process(CLOCK, RESET) is
  begin
    if RESET = '1' then
      PCI_RENWEN <= '1' ;
      state <= IDLE;

    elsif rising_edge(CLOCK) then
--      PCI_RENWEN <= '0'; -- Default to enable com.
--      OMA_VALID <= '0'; -- NOT valid data per default.

      if PCI_EMPTY = '0' and state = READ_DATA then
          -- Gather data from input.
          data_i <= PCI_ADIO;
          -- Debug fnord.
          LED_DEBUG <= PCI_ADIO(0 to 15);
          OMA_VALID <= '1';
          state <= COMPLETE;

      elsif PCI_BUSY = '0'  then
      -- else
        OMA_VALID <= '0'; -- NOT valid data per default.
        case state is
          when IDLE =>
            PCI_RENWEN <= '1'; -- We're not enabling com when idle.
            PCI_ADIO <= ( others => 'Z' );

            -- Write on low!
            if OMA_RW = '0' and OMA_ACTIVE='1' then
              state <= WRITE_START;
              PCI_RENWEN <= '0';
            elsif OMA_RW = '1' and OMA_ACTIVE = '1' then
              state <= READ_START;
              PCI_RENWEN <= '0';
            else
              state <= IDLE;
            end if;

          -- Writing to the bus.
          when WRITE_START =>
            PCI_ADIO <= writeSignal;
            PCI_ADIO(16 to 19) <= OMA_BE;
            state <= WRITE_ADDR;
          when WRITE_ADDR =>
            PCI_ADIO <= address;
            state <= WRITE_DATA;
          when WRITE_DATA =>
            PCI_ADIO <= data_o;
            -- Debug fnord
            OMA_VALID <= '1';
            state <= COMPLETE;

          -- Reading from the bus.
```

```
          when READ_START =>
            PCI_ADIO <= readSignal;
            state <= READ_ADDR;
          when READ_ADDR =>
            PCI_ADIO <= address;
            state <= READ_IDLE;
          when READ_IDLE =>
            PCI_ADIO <= (others => 'Z');
            state <= READ_DATA;


          -- The requested transaction is done!
          when COMPLETE =>
            PCI_ADIO <= data_i;
            state <= IDLE;
          when others =>
            null;
        end case;
      end if;
  end if;

  end process;

--  -- show state on led
--  lSate: process(state) is
--  begin
--    case state is
--      when IDLE =>
--        LED_DEBUG <= sIDLE;
--      when READ_START =>
--        LED_DEBUG <= sREAD_START;
--      when READ_ADDR =>
--        LED_DEBUG <= sREAD_ADDR ;
--      when READ_DATA =>
--        LED_DEBUG <= sREAD_DATA;
--      when READ_IDLE =>
--        LED_DEBUG <= sREAD_IDLE ;
--      when WRITE_START =>
--        LED_DEBUG <= sWRITE_START;
--      when WRITE_ADDR=>
--        LED_DEBUG <= sWRITE_ADDR;
--      when WRITE_DATA=>
--        LED_DEBUG <= sWRITE_DATA;
--      when COMPLETE=>
--        LED_DEBUG <= sCOMPLETE;
--    end case;
--  end process;

  comb: process(state, OMA_DATA_O, OMA_ADDRESS, RESET, data_i) is
  begin


    if RESET = '1' then
      data_o <= (others => 'Z');
      address <= (others => 'Z');
    else
```

```
        PCI_RW     <= '1'; -- We're writing as default
        --OMA_VALID <= '0';

        OMA_DATA_I <= data_i;
        address <= OMA_ADDRESS;
        data_o <= OMA_DATA_O;

        case state is
          when READ_DATA =>
            PCI_RW <= '0'; -- READ !!
          when COMPLETE =>
            -- OMA_VALID <= '1';
            null;
          when others =>
            null;
        end case;
  end if;
  end process;

end Behavioral;
```

B.7 Toplevel - mCache

```
--
--------------------------------------------------------------------------------
-- Create Date:    13:58:29 11/20/2006
-- Design Name:
-- Module Name:    toplevel - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

use work.coreStuff.all;

entity toplevel is
  PORT(
     sys_clk_pin          : IN std_logic;
     reset               : IN std_logic;
     fpga_0_LEDS_GPIO_d_out_pin   : OUT std_logic_vector(0 to 15);
     PCI_ADDS            : in  STD_LOGIC;
     PCI_EMPTY             : in  STD_LOGIC;
     PCI_BUSY            : in  STD_LOGIC;
     PCI_RW              : out  STD_LOGIC;
     PCI_RENWEN            : out  STD_LOGIC;
     PCI_ADIO            : inout  STD_LOGIC_VECTOR (0 to 31));
end toplevel;

architecture Behavioral of toplevel is
     signal OMA_VALID   : std_logic;
     signal OMA_DATA_I : std_logic_vector(0 to 31) := (others =>'0');
     signal OMA_DATA_O : std_logic_vector(0 to 31) := (others =>'0');
     signal OMA_ADDRESS   : std_logic_vector(0 to 31) := x"DEADBEEF";
     signal OMA_RW    : std_logic;
     signal OMA_ACTIVE : std_logic;
     signal OMA_BE    : std_logic_vector (0 to 3) := (others => '0');

```

```
    signal CACHE_VALID  : std_logic;
    signal CACHE_DATA_I : std_logic_vector(0 to 31) := (others =>'0');
    signal CACHE_DATA_O : std_logic_vector(0 to 31) := (others =>'0');
    signal CACHE_ADDRESS  : std_logic_vector(0 to 31) := ( others => '0');
    signal CACHE_RW   : std_logic;
    signal CACHE_ACTIVE : std_logic;
    signal CACHE_BE   : std_logic_vector  (0 to 3) := (others => '0');

    signal CORE_VALID   : corebit;
    signal CORE_DATA_I  : corevector := (others => (others =>'0'));
    signal CORE_DATA_O  : corevector := (others => (others =>'0'));
    signal CORE_ADDRESS : corevector := (others => (others =>'0'));
    signal CORE_RW    : corebit;
    signal CORE_ACTIVE  : corebit;
    signal CORE_BE    : corebe;


    signal sPCI_ADDS  : std_logic := '0';
    signal sPCI_EMPTY   : std_logic := '0';
    signal sPCI_BUSY  : std_logic := '0';
    signal sPCI_RW    : std_logic := '0';
    signal sPCI_RENWEN  : std_logic := '1';

    signal rst_inv    : std_logic;
    signal rst_internal : std_logic;


  COMPONENT core
  PORT(
    sys_clk_pin  : IN std_logic;
    sys_rst_pin  : IN std_logic;
    opb_external_memory_0_OMA_DATA_I_pin   : IN std_logic_vector(0 to 31);
    opb_external_memory_0_OMA_VALID_pin    : IN std_logic;
    opb_external_memory_1_OMA_DATA_I_pin   : IN std_logic_vector(0 to 31);
    opb_external_memory_1_OMA_VALID_pin    : IN std_logic;
    opb_external_memory_0_OMA_DATA_O_pin   : OUT std_logic_vector(0 to 31);
    opb_external_memory_0_OMA_ADDRESS_pin    : OUT std_logic_vector(0 to 31);
    opb_external_memory_0_OMA_RW_pin      : OUT std_logic;
    opb_external_memory_0_OMA_ACTIVE_pin   : OUT std_logic;
    opb_external_memory_0_LEDS_DEBUG_pin   : OUT std_logic_vector(0 to 15);
    opb_external_memory_0_OMA_BE_pin      : OUT std_logic_vector(0 to 3);
    opb_external_memory_1_OMA_DATA_O_pin   : OUT std_logic_vector(0 to 31);
    opb_external_memory_1_OMA_ADDRESS_pin    : OUT std_logic_vector(0 to 31);
    opb_external_memory_1_OMA_RW_pin      : OUT std_logic;
    opb_external_memory_1_OMA_ACTIVE_pin   : OUT std_logic;
    opb_external_memory_1_LEDS_DEBUG_pin   : OUT std_logic_vector(0 to 15);
    opb_external_memory_1_OMA_BE_pin      : OUT std_logic_vector(0 to 3)
    );
  END COMPONENT;



  COMPONENT pciCom
    Port (
    CLOCK       : IN  STD_LOGIC;
    RESET       : IN  STD_LOGIC;
```

```
        PCI_ADDS      : IN     STD_LOGIC;
        PCI_EMPTY       : IN     STD_LOGIC;
        PCI_BUSY      : IN     STD_LOGIC;
        PCI_RW        : OUT    STD_LOGIC;
        PCI_RENWEN      : OUT    STD_LOGIC;
        PCI_ADIO      : INOUT STD_LOGIC_VECTOR (0 to 31);
--      LED_DEBUG : OUT std_logic_vector(0 to 15);
        OMA_VALID       : OUT    STD_LOGIC;
        OMA_DATA_I      : OUT    STD_LOGIC_VECTOR(0 to 31);
        OMA_BE        : IN   STD_LOGIC_VECTOR(0 to 3);
        OMA_DATA_O      : IN   STD_LOGIC_VECTOR(0 to 31);
        OMA_ADDRESS     : IN   STD_LOGIC_VECTOR(0 to 31);
        OMA_RW        : IN   STD_LOGIC;
        OMA_ACTIVE      : IN   STD_LOGIC
        );
    END COMPONENT;

begin

    Inst_system: core PORT MAP(
        opb_external_memory_0_LEDS_DEBUG_pin   => fpga_0_LEDS_GPIO_d_out_pin,
        sys_clk_pin                  => sys_clk_pin,
        sys_rst_pin                  => rst_inv,
        opb_external_memory_0_OMA_DATA_O_pin   => CORE_DATA_O (0) ,
        opb_external_memory_0_OMA_DATA_I_pin   => CORE_DATA_I (0) ,
        opb_external_memory_0_OMA_VALID_pin    => CORE_VALID(0) ,
        opb_external_memory_0_OMA_ADDRESS_pin    => CORE_ADDRESS(0) ,
        opb_external_memory_0_OMA_RW_pin     => CORE_RW (0) ,
        opb_external_memory_0_OMA_ACTIVE_pin   => CORE_ACTIVE(0) ,
        opb_external_memory_0_OMA_BE_pin     => CORE_BE(0) ,
        opb_external_memory_1_OMA_DATA_O_pin   => CORE_DATA_O (1) ,
        opb_external_memory_1_OMA_DATA_I_pin   => CORE_DATA_I (1) ,
        opb_external_memory_1_OMA_VALID_pin    => CORE_VALID(1) ,
        opb_external_memory_1_OMA_ADDRESS_pin    => CORE_ADDRESS(1) ,
        opb_external_memory_1_OMA_RW_pin     => CORE_RW (1) ,
        opb_external_memory_1_OMA_ACTIVE_pin   => CORE_ACTIVE(1) ,
        opb_external_memory_1_OMA_BE_pin     => CORE_BE(1)
    );

    arbiter: entity work.arbiter PORT MAP(
        CLOCK         =>  sys_clk_pin,
        RESET         =>  rst_inv,

        IN_OMA_DATA_O     => CORE_DATA_O,
        IN_OMA_DATA_I     => CORE_DATA_I,
        IN_OMA_VALID      => CORE_VALID,
        IN_OMA_ADDRESS      => CORE_ADDRESS,
        IN_OMA_RW       => CORE_RW,
        IN_OMA_ACTIVE     => CORE_ACTIVE,
        IN_OMA_BE       => CORE_BE,

        OUT_OMA_VALID     => CACHE_VALID,
        OUT_OMA_DATA_I      => CACHE_DATA_I,
        OUT_OMA_DATA_O      => CACHE_DATA_O,
        OUT_OMA_ADDRESS     => CACHE_ADDRESS,
        OUT_OMA_BE        => CACHE_BE ,
```

```
    OUT_OMA_RW          => CACHE_RW,
    OUT_OMA_ACTIVE      => CACHE_ACTIVE

  );

  cache : entity work.cacheBlock PORT MAP (
    CLOCK               => sys_clk_pin ,
        RESET               => rst_inv ,
        -- From the arbiter to Cache
        IN_OMA_VALID      => CACHE_VALID,
        IN_OMA_DATA_I     => CACHE_DATA_I,
        IN_OMA_BE         => CACHE_BE,
        IN_OMA_DATA_O     => CACHE_DATA_O,
        IN_OMA_ADDRESS    => CACHE_ADDRESS,
        IN_OMA_RW         => CACHE_RW,
        IN_OMA_ACTIVE     => CACHE_ACTIVE,

        -- From the cache to comm.
        OUT_OMA_VALID     => OMA_VALID,
        OUT_OMA_DATA_I    => OMA_DATA_I,
        OUT_OMA_BE        => OMA_BE,
        OUT_OMA_DATA_O    => OMA_DATA_O,
        OUT_OMA_ADDRESS => OMA_ADDRESS,
        OUT_OMA_RW        => OMA_RW,
        OUT_OMA_ACTIVE    => OMA_ACTIVE
  );

  communication: pciCom PORT MAP(
    CLOCK       => sys_clk_pin ,
    RESET     => rst_inv ,
    PCI_ADDS    => sPCI_ADDS,
    PCI_EMPTY     => sPCI_EMPTY,
    PCI_BUSY    => sPCI_BUSY,
    PCI_RW      => sPCI_RW,
    PCI_RENWEN    => sPCI_RENWEN,
    PCI_ADIO    => PCI_ADIO,

    OMA_BE      => OMA_BE,
    OMA_VALID     => OMA_VALID,
    OMA_DATA_I    => OMA_DATA_I,
    OMA_DATA_O    => OMA_DATA_O,
    OMA_ADDRESS => OMA_ADDRESS,
    OMA_RW      => OMA_RW,
    OMA_ACTIVE    => OMA_ACTIVE
    );

  IBUF_inst : IBUF
   generic map (
      IBUF_DELAY_VALUE => "0",
      IFD_DELAY_VALUE => "AUTO",
      IOSTANDARD => "DEFAULT")
   port map (
      O => rst_internal ,
      I => reset
   );

```

```
  f: process(PCI_ADDS, PCI_EMPTY, PCI_BUSY, sPCI_RW, sPCI_RENWEN,
      rst_internal) is
  begin
    sPCI_ADDS    <= PCI_ADDS ;
    sPCI_EMPTY   <= PCI_EMPTY;
    sPCI_BUSY <= PCI_BUSY;
    PCI_RW      <= sPCI_RW;
    PCI_RENWEN   <= sPCI_RENWEN ;
    rst_inv     <= not rst_internal;
  end process;

end Behavioral;
```

B.8 Toplevel - mCore

```
-- ----------------------------------------------------------------------------
-- Create Date:    13:58:29 11/20/2006
-- Design Name:
-- Module Name:    toplevel - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- ----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

use work.coreStuff.all;

entity toplevel is
  PORT(
     sys_clk_pin          : IN std_logic;
     reset              : IN std_logic;
     fpga_0_LEDS_GPIO_d_out_pin   : OUT std_logic_vector(0 to 15);
     PCI_ADDS           : in   STD_LOGIC;
     PCI_EMPTY            : in   STD_LOGIC;
     PCI_BUSY           : in   STD_LOGIC;
     PCI_RW             : out   STD_LOGIC;
     PCI_RENWEN           : out   STD_LOGIC;
     PCI_ADIO           : inout   STD_LOGIC_VECTOR (0 to 31));
end toplevel;

architecture Behavioral of toplevel is
     signal OMA_VALID   : std_logic;
     signal OMA_DATA_I  : std_logic_vector(0 to 31) := (others =>'0');
     signal OMA_DATA_O  : std_logic_vector(0 to 31) := (others =>'0');
     signal OMA_ADDRESS   : std_logic_vector(0 to 31) := x"DEADBEEF";
     signal OMA_RW    : std_logic;
     signal OMA_ACTIVE  : std_logic;
     signal OMA_BE    : std_logic_vector (0 to 3) := (others => '0');

```

```
    signal CORE_VALID   : corebit;
    signal CORE_DATA_I  : corevector := (others => (others =>'0'));
    signal CORE_DATA_O  : corevector := (others => (others =>'0'));
    signal CORE_ADDRESS : corevector := (others => (others =>'0'));
    signal CORE_RW      : corebit;
    signal CORE_ACTIVE  : corebit;
    signal CORE_BE      : corebe;


    signal sPCI_ADDS   : std_logic := '0';
    signal sPCI_EMPTY   : std_logic := '0';
    signal sPCI_BUSY   : std_logic := '0';
    signal sPCI_RW     : std_logic := '0';
    signal sPCI_RENWEN  : std_logic := '1';

    signal rst_inv      : std_logic;
    signal rst_internal : std_logic;


  COMPONENT core
  PORT(
    sys_clk_pin : IN std_logic;
    sys_rst_pin : IN std_logic;
    opb_external_memory_0_OMA_DATA_I_pin   : IN std_logic_vector(0 to 31);
    opb_external_memory_0_OMA_VALID_pin    : IN std_logic;
    opb_external_memory_1_OMA_DATA_I_pin   : IN std_logic_vector(0 to 31);
    opb_external_memory_1_OMA_VALID_pin    : IN std_logic;
    opb_external_memory_0_OMA_DATA_O_pin   : OUT std_logic_vector(0 to 31);
    opb_external_memory_0_OMA_ADDRESS_pin    : OUT std_logic_vector(0 to 31);
    opb_external_memory_0_OMA_RW_pin      : OUT std_logic;
    opb_external_memory_0_OMA_ACTIVE_pin   : OUT std_logic;
    opb_external_memory_0_LEDS_DEBUG_pin   : OUT std_logic_vector(0 to 15);
    opb_external_memory_0_OMA_BE_pin      : OUT std_logic_vector(0 to 3);
    opb_external_memory_1_OMA_DATA_O_pin   : OUT std_logic_vector(0 to 31);
    opb_external_memory_1_OMA_ADDRESS_pin    : OUT std_logic_vector(0 to 31);
    opb_external_memory_1_OMA_RW_pin      : OUT std_logic;
    opb_external_memory_1_OMA_ACTIVE_pin   : OUT std_logic;
    opb_external_memory_1_LEDS_DEBUG_pin   : OUT std_logic_vector(0 to 15);
    opb_external_memory_1_OMA_BE_pin      : OUT std_logic_vector(0 to 3)
    );
  END COMPONENT;



  COMPONENT pciCom
    Port (
    CLOCK       : IN   STD_LOGIC;
    RESET       : IN   STD_LOGIC;
    PCI_ADDS       : IN      STD_LOGIC;
    PCI_EMPTY        : IN      STD_LOGIC;
    PCI_BUSY       : IN      STD_LOGIC;
    PCI_RW         : OUT     STD_LOGIC;
    PCI_RENWEN       : OUT     STD_LOGIC;
    PCI_ADIO       : INOUT STD_LOGIC_VECTOR (0 to 31);
--    LED_DEBUG : OUT std_logic_vector(0 to 15);
    OMA_VALID        : OUT     STD_LOGIC;
```

```
    OMA_DATA_I     : OUT   STD_LOGIC_VECTOR(0 to 31);
    OMA_BE        : IN  STD_LOGIC_VECTOR(0 to 3);
    OMA_DATA_O     : IN   STD_LOGIC_VECTOR(0 to 31);
    OMA_ADDRESS    : IN   STD_LOGIC_VECTOR(0 to 31);
    OMA_RW        : IN  STD_LOGIC;
    OMA_ACTIVE     : IN   STD_LOGIC
    );
  END COMPONENT;

begin

  Inst_system: core PORT MAP(
    opb_external_memory_0_LEDS_DEBUG_pin  => fpga_0_LEDS_GPIO_d_out_pin,
    sys_clk_pin                 => sys_clk_pin,
    sys_rst_pin                 => rst_inv,
    opb_external_memory_0_OMA_DATA_O_pin  => CORE_DATA_O (0)  ,
    opb_external_memory_0_OMA_DATA_I_pin  => CORE_DATA_I (0)  ,
    opb_external_memory_0_OMA_VALID_pin   => CORE_VALID(0)  ,
    opb_external_memory_0_OMA_ADDRESS_pin   => CORE_ADDRESS(0)  ,
    opb_external_memory_0_OMA_RW_pin     => CORE_RW (0)  ,
    opb_external_memory_0_OMA_ACTIVE_pin  => CORE_ACTIVE(0)  ,
    opb_external_memory_0_OMA_BE_pin     => CORE_BE(0)  ,
    opb_external_memory_1_OMA_DATA_O_pin  => CORE_DATA_O (1)  ,
    opb_external_memory_1_OMA_DATA_I_pin  => CORE_DATA_I (1)  ,
    opb_external_memory_1_OMA_VALID_pin   => CORE_VALID(1)  ,
    opb_external_memory_1_OMA_ADDRESS_pin   => CORE_ADDRESS(1)  ,
    opb_external_memory_1_OMA_RW_pin     => CORE_RW (1)  ,
    opb_external_memory_1_OMA_ACTIVE_pin  => CORE_ACTIVE(1)  ,
    opb_external_memory_1_OMA_BE_pin     => CORE_BE(1)
  );

  arbiter: entity work.arbiter PORT MAP(
    CLOCK        =>  sys_clk_pin,
    RESET        =>  rst_inv,

    IN_OMA_DATA_O    => CORE_DATA_O,
    IN_OMA_DATA_I    => CORE_DATA_I,
    IN_OMA_VALID     => CORE_VALID,
    IN_OMA_ADDRESS     => CORE_ADDRESS,
    IN_OMA_RW       => CORE_RW,
    IN_OMA_ACTIVE    => CORE_ACTIVE,
    IN_OMA_BE       => CORE_BE,

    OUT_OMA_VALID    => OMA_VALID,
    OUT_OMA_DATA_I     => OMA_DATA_I,
    OUT_OMA_DATA_O     => OMA_DATA_O,
    OUT_OMA_ADDRESS    => OMA_ADDRESS,
    OUT_OMA_BE       => OMA_BE  ,
    OUT_OMA_RW       => OMA_RW,
    OUT_OMA_ACTIVE     => OMA_ACTIVE

  );


  communication: pciCom PORT MAP(
    CLOCK       => sys_clk_pin,
```

```
      RESET     => rst_inv ,
      PCI_ADDS   => sPCI_ADDS,
      PCI_EMPTY    => sPCI_EMPTY,
      PCI_BUSY   => sPCI_BUSY,
      PCI_RW      => sPCI_RW,
      PCI_RENWEN   => sPCI_RENWEN,
      PCI_ADIO   => PCI_ADIO,

      OMA_BE      => OMA_BE,
      OMA_VALID     => OMA_VALID,
      OMA_DATA_I   => OMA_DATA_I,
      OMA_DATA_O   => OMA_DATA_O,
      OMA_ADDRESS => OMA_ADDRESS,
      OMA_RW      => OMA_RW,
      OMA_ACTIVE   => OMA_ACTIVE
      );

   IBUF_inst : IBUF
    generic map (
        IBUF_DELAY_VALUE => "0",
        IFD_DELAY_VALUE => "AUTO",
        IOSTANDARD => "DEFAULT")
    port map (
        O => rst_internal ,
        I => reset
    );

   f: process(PCI_ADDS, PCI_EMPTY, PCI_BUSY, sPCI_RW, sPCI_RENWEN,
       rst_internal) is
   begin
      sPCI_ADDS    <= PCI_ADDS ;
      sPCI_EMPTY   <= PCI_EMPTY;
      sPCI_BUSY <= PCI_BUSY;
      PCI_RW      <= sPCI_RW;
      PCI_RENWEN   <= sPCI_RENWEN ;
      rst_inv    <= not rst_internal;
   end process;

end Behavioral;
```

B.9 Toplevel - sCore

```
----------------------------------------------------------------------------------
-- Create Date:    13:58:29 11/20/2006
-- Design Name:
-- Module Name:    toplevel - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity toplevel is
  PORT(
     sys_clk_pin  : IN std_logic;
     reset  : IN std_logic;
     fpga_0_LEDS_GPIO_d_out_pin  : OUT std_logic_vector(0 to 15);
     PCI_ADDS     : in   STD_LOGIC;
     PCI_EMPTY    : in   STD_LOGIC;
     PCI_BUSY     : in   STD_LOGIC;
     PCI_RW      : out  STD_LOGIC;
     PCI_RENWEN   : out  STD_LOGIC;
     PCI_ADIO     : inout  STD_LOGIC_VECTOR (0 to 31));
end toplevel;

architecture Behavioral of toplevel is
     signal OMA_VALID   : std_logic;
     signal OMA_DATA_I  : std_logic_vector(0 to 31) := (others =>'0');
     signal OMA_DATA_O  : std_logic_vector(0 to 31) := (others =>'0');
     signal OMA_ADDRESS   : std_logic_vector(0 to 31) := x"DEADBEEF";
     signal OMA_RW    : std_logic;
     signal OMA_ACTIVE  : std_logic;
     signal OMA_BE    : std_logic_vector (0 to 3) := (others => '0');


     signal sPCI_ADDS   : std_logic := '0';
```

```
    signal sPCI_EMPTY    : std_logic := '0';
    signal sPCI_BUSY   : std_logic := '0';
    signal sPCI_RW     : std_logic := '0';
    signal sPCI_RENWEN   : std_logic := '1';

    signal rst_inv     : std_logic;
    signal rst_internal : std_logic;


  COMPONENT system
  PORT(
    opb_external_memory_0_LEDS_DEBUG_pin : OUT std_logic_vector(0 to 15);
    sys_clk_pin : IN std_logic;
    sys_rst_pin : IN std_logic;
    opb_external_memory_0_OMA_DATA_I_pin : IN std_logic_vector(0 to 31);
    opb_external_memory_0_OMA_VALID_pin : IN std_logic;
    opb_external_memory_0_OMA_DATA_O_pin : OUT std_logic_vector(0 to 31);
    opb_external_memory_0_OMA_ADDRESS_pin : OUT std_logic_vector(0 to 31);
    opb_external_memory_0_OMA_RW_pin : OUT std_logic;
    opb_external_memory_0_OMA_BE_pin : OUT std_logic_vector(0 to 3);
    opb_external_memory_0_OMA_ACTIVE_pin : OUT std_logic
    );
  END COMPONENT;

  COMPONENT pciCom
    Port (
    CLOCK      : in std_logic;
    RESET      : in std_logic;
    PCI_ADDS     : in  STD_LOGIC;
    PCI_EMPTY    : in  STD_LOGIC;
    PCI_BUSY     : in  STD_LOGIC;
    PCI_RW     : out  STD_LOGIC;
    PCI_RENWEN   : out  STD_LOGIC;
    PCI_ADIO     : inout  STD_LOGIC_VECTOR (0 to 31);
--    LED_DEBUG : OUT std_logic_vector(0 to 15);
    OMA_VALID    : OUT std_logic;
    OMA_DATA_I   : OUT std_logic_vector(0 to 31);
    OMA_BE     : IN std_logic_vector(0 to 3);
    OMA_DATA_O   : IN std_logic_vector(0 to 31);
    OMA_ADDRESS : IN std_logic_vector(0 to 31);
    OMA_RW     : IN std_logic;
    OMA_ACTIVE   : IN std_logic
    );
  END COMPONENT;

begin

   Inst_system: system PORT MAP(
    opb_external_memory_0_LEDS_DEBUG_pin   => fpga_0_LEDS_GPIO_d_out_pin,
    sys_clk_pin                 => sys_clk_pin,
    sys_rst_pin                 => rst_inv,
    opb_external_memory_0_OMA_DATA_O_pin   => OMA_DATA_O  ,
    opb_external_memory_0_OMA_DATA_I_pin   => OMA_DATA_I  ,
    opb_external_memory_0_OMA_VALID_pin    => OMA_VALID,
    opb_external_memory_0_OMA_ADDRESS_pin => OMA_ADDRESS,
    opb_external_memory_0_OMA_RW_pin      => OMA_RW  ,
```

```
    opb_external_memory_0_OMA_ACTIVE_pin   => OMA_ACTIVE,
    opb_external_memory_0_OMA_BE_pin       => OMA_BE
  );

  communication: pciCom PORT MAP(
    CLOCK      => sys_clk_pin,
    RESET      => rst_inv,
    PCI_ADDS   => sPCI_ADDS,
    PCI_EMPTY    => sPCI_EMPTY,
    PCI_BUSY   => sPCI_BUSY,
    PCI_RW     => sPCI_RW,
    PCI_RENWEN   => sPCI_RENWEN,
    PCI_ADIO   => PCI_ADIO,
    OMA_BE     => OMA_BE,
    OMA_VALID    => OMA_VALID,
    OMA_DATA_I   => OMA_DATA_I,
    OMA_DATA_O   => OMA_DATA_O,
    OMA_ADDRESS => OMA_ADDRESS,
    OMA_RW     => OMA_RW,
    OMA_ACTIVE   => OMA_ACTIVE
    );

  IBUF_inst : IBUF
      generic map (
       IBUF_DELAY_VALUE => "0",
       IFD_DELAY_VALUE => "AUTO",
       IOSTANDARD => "DEFAULT")
      port map (
       O => rst_internal,
       I => reset
      );

  f: process(PCI_ADDS, PCI_EMPTY, PCI_BUSY, sPCI_RW, sPCI_RENWEN,
      rst_internal) is
  begin
    sPCI_ADDS    <= PCI_ADDS ;
    sPCI_EMPTY   <= PCI_EMPTY;
    sPCI_BUSY <= PCI_BUSY;
    PCI_RW     <= sPCI_RW;
    PCI_RENWEN   <= sPCI_RENWEN ;
    rst_inv    <= not rst_internal;
  end process;

end Behavioral;
```

Appendix C

Software

C.1 Controller.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

#include "controller.h"
#include "cpu.h"
#include "memory.h"
#include "util.h"


// #define DEBUG

static Memory *m;
static STATE state;

int run = 1;

unsigned long long int memRead = 0, memWrite = 0;

unsigned char byteEnabled = 0;

__inline__ unsigned long long int rdtsc() {
  unsigned long long int x;
  __asm__ volatile (".byte␣0x0f,␣0x31" : "=A" (x));
  return x;
}


void termi(int code){
  printf("\nNow␣exiting␣with␣code:␣%d\n", code);

#ifdef DEBUG
  printf("Want␣to␣display␣a␣memory␣map?\n");
  if ( getchar() == 'y' ){
```

```
      printf("Memory Map\n");
      mem_print(m);
    }
#endif

  printf("\n");
  exit(code);
}

void trap_seg(int signal){
  printf("Seg.fault\n");
  termi(5);
}

int handle(unsigned int command){
  unsigned int reversed = reverseInt(command);

#ifdef DEBUG
  printf("Doing state: 0x%08x, r: 0x%08x\n",command, reversed);
#endif

  if( state == IDLE ) {
#ifdef DEBUG
    printf("Reverse: 0x%08X\n", reversed);
#endif
    if( reversed == READ_SIGNAL ){
      state = R_ADDR;
    } else if ( (reversed & 0xFF) == 0x02 ){
      state = W_ADDR;
      // Byte Enabled Hack.
      byteEnabled = (reversed >> 4*3) & 0xF;
#ifdef DEBUG
      printf("Byte Lines: 0x%08X\n",byteEnabled);
#endif
    }
  }else if( WRITING <= state && state <= W_DATA){
    state = handleWrite(command,state);
  }else if( READING <= state && state <= R_DATA) {
    state = handleRead(command,state);
  }else if ( state == COMPLETE ){
#ifdef DEBUG
    printf("Operation took 0x%08X operations\n",reversed);
#endif
    state = IDLE;
  }

}

int handleRead(unsigned int command,const STATE state){
  /* The FPGA tries to access a memory location */
  unsigned int reversed = reverseInt(command);
  unsigned int r;

#ifdef DEBUG
  printf("Now handling read to memory: 0x%X\n",reversed);
#endif
```

```
  int i = mem_read(reversed ,NULL,m);
  r = reverseInt(i);

#ifdef DEBUG
  printf("Sending value: 0x%08x, r=0x%08x\n",i,r);
#endif
  writeWord32(i);
  memRead++;

  // IDLE STAGE HACK!
  getWord32();

  return COMPLETE;
}

int handleWrite(unsigned int command,const STATE state){
  unsigned int reversed = reverseInt(command);
  /* The FPGA tries to write to a memory location */

  unsigned int i = 0;

  switch( state ) {
    case W_ADDR:
      i = getWord32();
#ifdef DEBUG
      printf("Storing to mem: 0x%08x, data: 0x%08x\n",reversed ,i);
#endif
      if ( reversed == 0x7FFFFFFF ) {
        printf("\nTerminate signal from CPU\n");
        run = 0 ;
      //  termi(2);
      }
      memWrite++;

      if ( run ) mem_write(reversed ,i  ,byteEnabled, m);
    default: return COMPLETE;
  }

}

print_state() {
  switch(state) {
    case IDLE:
      printf("State: IDLE\n");
      break;
    case WRITING:
      printf("State: WRITING\n");
      break;
    case W_ADDR:
      printf("State: W_ADDR\n");
      break;
    case W_DATA:
      printf("State: W_DATA\n");
      break;
    case READING:
      printf("State: READING\n");
```

```
            break;
        case R_ADDR:
            printf("State: R_ADDR\n");
            break;
        case R_DATA:
            printf("State: R_DATA\n");
            break;
        case COMPLETE:
            printf("State: COMPLETE\n");
            break;
        default: printf("State: UNKNOWN\n");
    }
}

int main(int argc, char **argv) {
    MemBlock b;
    int f = 0;

    if( argc < 2 ) {
        printf("Usage: %s <bitfile> [application]\n", argv[0]);
        exit(1);
    }

//    signal(SIGSEGV, trap_seg);

    // Init the memory
    m = mem_init();
    int mm;
    if ( argc > 2 ){
        if ( (mm = mem_load(m,argv[2]  ) < 0 )){
            printf("Error opening file: %s, error: %d\n", argv[2], mm );
            exit(1);
        }
    }

    state = IDLE;

#ifdef DEBUG
    printf("Now opening the card\n");
#endif
    f = openCard(1, argv[1]);
#ifdef DEBUG
    printf("OpenCard returned: %d\n\n",f);
#endif

    //================== Configuration done.
    unsigned long long int start, end, runtime;
    start = end = 0 ;

    f = '\0';

    start = rdtsc();
    do {
#ifdef DEBUG
        print_state();
#endif
```

```c

    switch(f){
      case 's':
#ifdef DEBUG
        printf("Now writing\n");
#endif
        writeWord32(0xFF);
        break;
      case 'r':
#ifdef DEBUG
        printf("Reset\n");
#endif
        break;
      default:
        break;
    }

    handle(getWord32());
#ifdef DEBUG
    printf("Press q to quit, everything else to continue\n");
    getchar();
#endif
  }while ( run );
  end = rdtsc();

  printf("Stats:\n");
  printf("\tRunning time: %llu\n", end - start);
  printf("\tTotal mem access: %llu\n", memRead + memWrite);
  printf("\tTotal mem reads: %llu\n", memRead );
  printf("\tTotal mem writes: %llu\n", memWrite );

  printf("Did you find anything?\nNow exiting\n");
  f = closeCard();
  printf("Close card returned: %d\n",f);
  printf("Now printing memory\n");

  for(f=0; f<m->mem_used;f++){
    b = m->data[f];
    printf("Cell %d contains %02x\n",b.addr,b.data);
  }

  termi(3);

  return 1;
}
```

C.2 Controller.h

```
#ifndef __CONTROLLER_H
#define __CONTROLLER_H

#define READ_SIGNAL    0x0000001
#define WRITE_SIGNAL   0x0000002


typedef enum {
  IDLE = 0,
  WRITING,
  W_ADDR,
  W_DATA,
  READING,
  R_ADDR,
  R_DATA,
  COMPLETE
}STATE;

#endif
```

C.3 memory.h

```c

#ifndef __MEMORY_H
#define __MEMORY_H

#define START_MEM 20
#define GROW_RATE 20

// #define MEM_SIZE 0x4FC000C
#define MEM_SIZE 0x5000000

#include "rb.h"

struct _MemBlock {
  unsigned int addr;
  unsigned char data;
  struct _MemBlock *prev, *next;
};

typedef struct _MemBlock MemBlock;

typedef struct {
  unsigned int max_addr;
  unsigned int mem_used;
  unsigned int mem_size;
  MemBlock *data;
  struct rb_table *table;
  unsigned char *bytes;
}Memory;


Memory *mem_init();
int mem_load(Memory *m, char *filename);
unsigned int mem_read(unsigned int addr,int *error, Memory *m);
unsigned int mem_write(unsigned int addr, unsigned int data,unsigned char byteEnabled, Memory *m);
unsigned int mem_writeByte(unsigned int addr, unsigned char data, Memory *m);
int mem_clean(Memory *m);
int mem_print(Memory *m);
int mem_zero(Memory *m, unsigned int offset);

#endif
```

C.4 memory.c

```
/**
 * Simple memory holder..
 * Implementing a array,
 * A hash isn't worth the implementation
 * time, and I don't want to allocate 4Gb
 * of data for a traditional array.
 *
 * Kenneth Oestby <kenneo@idi.ntnu.no>
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "memory.h"
#include "util.h"
#include "rb.h"

int mem_load(Memory *m, char *filename){
  FILE *f = NULL;
  int memLoc = 0;
  unsigned int *data = 0;
  data =(unsigned int*)malloc(sizeof(unsigned int));
  *data = 0;

  if ( !m || !filename ) return -1;

  if( ! ( f= fopen(filename,"r") )) {
    return -2;
  }

  while( fread((void*)data, sizeof(unsigned char),
    1, f)) {
    mem_writeByte(memLoc++, reverseInt8(*data),m);
  }

  free(data);

  return memLoc;
}

int mem_print(Memory *m){
  if ( ! m  || !m->data) return -1;

  int i = 0;
  int x ;
  for( i = 0 ; i < MEM_SIZE; i++ ) {
    if ( !(i % 32) )
      printf("\n␣%03X␣", i );

    printf("%02hhX␣", mem_read(i, &x, m) );
  }
}
```

```

int mem_clean(Memory *m){
  return 1;
}

/*
 * Zero out the memory from the given offset.
 */
int mem_zero(Memory *m, unsigned int offset){
  if ( !m ) return -1;

  bzero(m->data + offset, (m->mem_size - offset) * sizeof(MemBlock));
  return 1;
}

Memory *mem_init(){
  Memory *m    = NULL;
  m = (Memory*)malloc(sizeof(Memory));
  m->data     = NULL;
  m->mem_size    = 0;
  m->mem_used    = 0;
  m->max_addr    = 0;

  /*
   * Bite of some bits of memory..
   *
   * If you're going use the CPU,
   * you're probably going to want
   * some memory to go along in the
   * first place. Malloc is _slow_!
   * Default is 20.. 20 * 32 should be enough
   * for everybody.
   */

  if ( ! ( m->data = (MemBlock*)malloc(sizeof(MemBlock) * START_MEM) ) ) {
    mem_clean(m);
    return NULL;
  }

  // m->table = rb_create(compare_ints,NULL,NULL);

  m->bytes = (unsigned char*)malloc(MEM_SIZE * sizeof(char));

  m->mem_size = START_MEM;
  mem_zero(m, 0);
  return m;
}

unsigned int mem_read(unsigned int addr,int *error,Memory *m){
  if ( !m ){
    if (error) *error = -1;
  }/*else if( !m->mem_used ){
    if (error) *error = -2;
  }else if( addr > m->max_addr ) {
    if (error) *error = -3;
  }*/
```

```

   if ( error && *error < 0 ) return 0;

   /*
   int i = 0, j = 0;
   MemBlock *tmp = NULL;
   unsigned int returnMe = 0;

   for ( i = 0; i < m->mem_used; i++ ) {
      if ( m->data[i].addr == addr ){
         tmp = m->data + i;
         do {
            printf("Fnord:%p\n",tmp);
            printf("Next:%p\n",tmp->next);
            returnMe |= tmp->data << j * 8;
         }while( (tmp = tmp->next) && ++j < 4 ) ;
      }
   }

   return returnMe;
   */

   return m->bytes[addr+3] << 24  | m->bytes[addr+2] << 16 | m->bytes[addr+1] << 8 | m->bytes[addr];
//   return m->bytes[addr];
}

int updatePointers(Memory *m, unsigned int addr){
   if ( !m ) return -1;

   // Slow implementation on write to help speed up read.
   // Update the next and prev-pointer.
   int i = 0 ;
   for(i = 0; i<m->mem_used; i++){
      if( m->data[i].addr == addr -1 )
         m->data[i].next = m->data + m->mem_used -1;
      else if( m->data[i].addr == addr +1 )
         m->data[i].prev = m->data + m->mem_used -1;
   }

   return 1;
}

unsigned int mem_writeByte(unsigned int addr, unsigned char data, Memory *m){
   if( !m ) return 0;
   static int i = 0;

   /*
   // Go through the array of memory to find the memory block.
   // Future implementation would need a hash or something
   // faster..
   for ( i = 0; i < m->mem_used; i++ ) {
      if ( m->data[i].addr == addr ){
         m->data[i].data = data;
         return 1;
```

```
    }
  }

  if ( m->max_addr < addr )
    m->max_addr = addr;

  // If not found in the array.. Expand it..
  if ( m->mem_used < m->mem_size ) {
    m->data[m->mem_used].addr = addr;
    m->data[m->mem_used++].data = data;
  }else {
    // We need to grow the actual data array.
    m->mem_size += GROW_RATE;
    printf("Now growing to %d\n",m->mem_size);
    m->data = (MemBlock*)realloc((void*)m->data,m->mem_size*sizeof(MemBlock));

    m->data[m->mem_used].addr = addr;
    m->data[m->mem_used++].data = data;
  }

  updatePointers(m, addr);
  */

  m->bytes[addr] = data;

  return 1;
}

unsigned int mem_write(unsigned int addr, unsigned int data, unsigned char byteEnabled, Memory *m){
  int i = 0 ;
  char d;

  // Split up the data in based upon byteEnabled field!
  for(i = 0; i<3; i++){
    if ( byteEnabled & ( 1 << i ) ){
      d = (data >> sizeof(char) * 4) & 0xFF;
      mem_writeByte(addr+i, d, m);
    }else break;
  }

  return 1;
}
```

C.5 cpu.c

```
#include "cpu.h"
#include "util.h"

#include <stdio.h>

int openCard(int cardNum, char *bitfile){
  char error[1024];
  DWORD errorNum;

  cpu.locate = NULL;
  cpu.card = NULL;
  cpu.dma = NULL;
  cpu.send = NULL;
  cpu.recv = NULL;

  cpu.locate = DIME_LocateCard(dlPCI,
          mbtALL,
          NULL,
          dldrDEFAULT,
          dlDEFAULT);

  if( !cpu.locate ){
    DIME_GetError(NULL,&errorNum,error);
    printf("\nLocate␣Error␣#%d\n%s\n",errorNum,error);
    closeCard();
    return -1;
  }

  if(!( cpu.card = DIME_OpenCard(cpu.locate,
            cardNum,dccOPEN_DEFAULT)))
  {
    DIME_GetError(NULL,&errorNum,error);
    printf("\nCard␣Error␣#%d\n%s\n",errorNum,error);
    closeCard();
    return -2;
  }

  // Setup chat.
  if( setupDMA() < 1 ) {
    printf("DMA␣failed\n");
    closeCard();
    return -3;
  }

  if( DIME_JTAGControl(cpu.card, djtagCONFIGSPEED,djtagMAXSPEED100) ) {
    printf("JTAG␣died\n");
    DIME_GetError(NULL,&errorNum,error);
    printf("\nError␣#%d\n%s\n",errorNum,error);
    closeCard();
    return -4;
  }

  // Finally set the clock before configuring the card:
  DIME_SetOscillatorFrequency (cpu.card, CLOCK_NUM, CLOCK_FREQ, NULL);
```

```c

  if(configureCard(bitfile) < 1){

    printf("Could not configure FPGA\n");
    closeCard();
    return -5;
  }

  if( resetCard() < 1 ){
    printf("Error resetting card.\n");
    closeCard();
    return -6;
  };

  return 1;
}

int setupDMA(){
  char error[1024];
  DWORD errorNum;

  bzero(cpu.recvBuffer,sizeof(int)*BUFFER_SIZE);
  bzero(cpu.sendBuffer,sizeof(int)*BUFFER_SIZE);

  if( !cpu.locate || ! cpu.card )
    return -1;

  if( !( cpu.send = DIME_LockMemory(cpu.card,
            (DWORD*)cpu.sendBuffer,
            sizeof(cpu.sendBuffer))))
  {
    printf("Could not setup send buffer\n");
    return -2;
  }

  if( !( cpu.recv = DIME_LockMemory(cpu.card,
            (DWORD*)cpu.recvBuffer,
            sizeof(cpu.recvBuffer))))
  {
    if( cpu.send ) DIME_UnLockMemory(cpu.card, cpu.send);
    printf("Could not setup send buffer\n");
    return -3;
  }

  if(! (  cpu.dma = DIME_DMAOpen(cpu.card, 1,0) ) ) {
    if( cpu.send ) DIME_UnLockMemory(cpu.card, cpu.send);
    if( cpu.recv ) DIME_UnLockMemory(cpu.card, cpu.recv);

    printf("Error opening DMA Channel numero uno\n");

    return -4;
  }

  // DO NOT INCREASE THE LOCAL MEMORY ADDR AFTER USE!!
  DIME_DMAControl(cpu.card, cpu.dma,ddmaLOCALNOINC,0);

```

```c
  return 1;
}

int configureCard(char *bitfile){
  char error[1024];
  DWORD errorNum;

  if( !bitfile || !cpu.card )
      return -1;

  if( !DIME_BootVirtexSingle(cpu.card, bitfile) ) {
    DIME_GetError(NULL,&errorNum, error);
    printf("Configure Error #%d\n%s\n",errorNum, error);
  }

  return 1;
}

int resetCard(){
  // ENABLE = 0, DISABLE = 1
  DIME_CardResetControl(cpu.card, drONBOARDFPGA, drENABLE, 0);
  DIME_CardResetControl(cpu.card, drINTERFACE, drTOGGLE, 0);
  DIME_CardResetControl(cpu.card, drONBOARDFPGA, drDISABLE, 0);

  return 1;
}

int closeCard(){
  if( cpu.card) {
    if( cpu.send ) DIME_UnLockMemory(cpu.card, cpu.send);
    if( cpu.recv ) DIME_UnLockMemory(cpu.card, cpu.recv);
    if( cpu.dma ) DIME_DMAClose(cpu.card, cpu.dma,ddmaCLOSETERMINATE);

    DIME_CloseCard(cpu.card); }

  // Close down the card
  if( cpu.locate ) DIME_CloseLocate(cpu.locate);

  cpu.card = (cpu.locate = NULL);

  return 1;
}

int writeCard(int size) {
  if( !cpu.card ) return -1;
  if( !cpu.dma )  return -2;
  if( !cpu.send ) return -3;


  if( ddmaOK !=
      DIME_DMAWriteFromLockedMem(cpu.card, cpu.dma,
          cpu.send, 0, 1, ddmaBLOCKING) )
  {
    printf("Error writing stuff..!");
  }

```

```
  return 1;
}

int readCard(int size){
  if( !cpu.card ) return -1;
  if( !cpu.dma ) return -2;
  if( !cpu.recv ) return -3;

  bzero(cpu.recvBuffer,BUFFER_SIZE);

  if ( ddmaOK != DIME_DMAReadToLockedMem(cpu.card,
        cpu.dma, cpu.recv,
        0, size, ddmaBLOCKING ) ) {
    printf("Error reading stuff\n");
  }

  return size;
}

// User functions.
unsigned int getWord32(){
  readCard(1);
  return cpu.recvBuffer[0];
}

unsigned int writeWord32(unsigned int word){
  cpu.sendBuffer[0] = HostToCoreI(word);
  writeCard(1);
  return 1;
}
```

C.6 cpu.h

```
/*
 * CPU Controller for Master Thesis Project
 * 2006 - Kenneth Oestby <kenneo@idi.ntnu.no>
 *
 *
 */

#include <dimesdl.h>

#ifndef __CPU_H

#define BUFFER_SIZE 2048
#define CLOCK_NUM 2
#define CLOCK_FREQ  40

typedef struct {
  DIME_HANDLE    card;
  LOCATE_HANDLE locate;
  DWORD   leds;
  unsigned int sendBuffer[BUFFER_SIZE];
  unsigned int recvBuffer[BUFFER_SIZE];
  DIME_MEMHANDLE   send;
  DIME_MEMHANDLE   recv;
  DIME_DMAHANDLE   dma;
}CPU;

static CPU cpu;

int openCard(int cardNum, char *bitfile);
int setupDMA();
int closeCard();
int configureCard(char *filename);
int writeCard(int size);
int readCard(int size);
int resetCard();


unsigned int getWord32();
unsigned int writeWord32(unsigned int word);


#endif
```

C.7 util.h

```
/**
 * Several utility functions
 * to make the life easier
 *
 * 2007 - Kenneth Oestby <kenneo@idi.ntnu.no>
 *
 */

unsigned int CoreToHostI(unsigned int i);
unsigned int HostToCoreI(unsigned int i);
unsigned long ByteSwap2 (unsigned long nLongNumber);
unsigned int reverseInt(unsigned int i);
unsigned char reverseInt8(unsigned char i);
```

C.8 util.c

```
/**
 * Several utility functions
 * to make the life easier
 *
 * 2007 - Kenneth Oestby <kenneo@idi.ntnu.no>
 *
 */

static char isBigEndian = 1;

unsigned long ByteSwap2 (unsigned long nLongNumber)
{
  return (((nLongNumber&0x000000FF)<<24)+((nLongNumber&0x0000FF00)<<8)+
          ((nLongNumber&0x00FF0000)>>8)+((nLongNumber&0xFF000000)>>24));
}

unsigned int CoreToHostI(unsigned int i){
  if ( isBigEndian )
    return i;

  return ByteSwap2(i);
}
unsigned int HostToCoreI(unsigned int i){
  if ( isBigEndian )
    return i;

  return ByteSwap2(i);
}

/*
 * Reverses the integer..
 */
unsigned int reverseInt(unsigned int number){
  int i = 0, j = 0;
        unsigned int tmp = 0;

        for(i=sizeof(int)*8-1,j=0;i>=0;i--,j++)
                tmp |= ((number >> i) & 1) << j;

        return tmp;
}

unsigned char reverseInt8(unsigned char number){
  int i = 0, j = 0;
        unsigned char tmp = 0;

        for(i=sizeof(unsigned char)*8-1,j=0;i>=0;i--,j++)
                tmp |= ((number >> i) & 1) << j;

        return tmp;
}
```