Hochschule
Augsburg University of
Applied Sciences

Masterarbeit

Studiengang
Informatik

Michael Seider

Implementation and Evaluation of a ParaNut Processor on an FPGA Using VHDL
(Implementierung eines ParaNut-Prozessors in VHDL mit Evaluation auf einer FPGA-Plattform)

Prüfer: Prof. Dr. Gundolf Kiefer

Abgabe der Arbeit: 18. November 2013
(Wintersemester 2013/14)

Verfasser der Arbeit:
Michael Seider
Biburger Str. 2a
86482 Aystetten
Telefon:+49 821 487108
michael.seider@hs-augsburg.de

Fakultät für Informatik
Telefon:+49 821 5586-3450
Fax: +49 821 5586-3499

Hochschule Augsburg
University of Applied Sciences
An der Fachhochschule 1
D - 86161 Augsburg

Telefon:+49 821 5586-0
Fax: +49 821 5586-3222
http://www.hs-augsburg.de
poststelle@hs-augsburg.de

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderern als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche oder sinngemäße Zitate als solche gekennzeichnet habe.

Aystetten, 18. November 2013

____

Michael Seider

Kurzfassung

In dieser Arbeit wird die Implementierung eines *ParaNut*-Prozessors vorgestellt. Dieser basiert auf der ParaNut-Architektur, einer offenen, skalierbaren Mehrkern-Prozessorarchitektur. Ziel der Arbeit ist dabei eine funktionsfähige VHDL Implementierung zum Einsatz auf einem FPGA. Der Prozessor wird zunächst auf Grundlage einer taktgenauen SystemC-Referenzimplementierung entwickelt. Anschließend wird der Prozessor in das OpenRISC Reference Platform System-on-Chip (ORPSoC) integriert und auf einer FPGA Hardware-Plattform realisiert. Nach Validierung der korrekten Funktion der ParaNut-Implementierung werden verschiedene Experimente auf der Hardwareplattform ausgeführt, welche die Leistungsfähigkeit der Architektur bestimmen sollen. Die Performance-Evaluation umfasst dabei neben der reinen Ausführungsgeschwindigkeit von Benchmarkprogrammen auch eine genaue Untersuchung der Performanz interner Komponenten der ParaNut-Architektur. Schließlich wird auch die korrekte Funktion und Leistungsfähigkeit des Prozessors mit mehreren Prozessor-Kernen bestimmt. Abschließend wird die Implementierung noch hinsichtlich Ressourcenverbrauch und maximaler Frequenz für den eingesetzten FPGA untersucht, wobei Vorschläge für mögliche Optimierungen genannt werden.

Stichworte

ParaNut, Multicore System-on-Chip, FPGA, OpenRISC

Abstract

This work presents the implementation of a *ParaNut* processor. It is based on the ParaNut architecture, an open, scalable multi-core processor architecture. The aim of this work is a functional implementation of a ParaNut processor on an FPGA using VHDL. The processor is implemented on basis of a cycle accurate SystemC reference implementation. The processor is then embedded into the OpenRISC Reference Platform System-on-Chip (ORPSoC), which is synthesized for use on an FPGA platform. After validating the correct functionality of the ParaNut VHDL implementation, experiments are performed that determine the performance of the architecture. Performance measurements include running standard benchmark programs as well as a detailed analysis of the performance of internal components of the ParaNut architecture. Eventually, the correct operation and performance of the processor with multiple cores is investigated. Lastly, the implementation is examined in terms of resource usage and timing, whereupon suggestions for possible optimizations are proposed.

Keywords

ParaNut, Multi-Core System-on-Chip, FPGA, OpenRISC

Contents

List of Figures

List of Tables

1 Introduction

Since the beginning of modern computing, the demands to the computational power of electrical computer systems have risen continually. For many years, the constant increase of processor clock rates ensured this performance goal. However, due to increased power requirements that arise from scaling transistors and wires in chips, clock rates have not improved as much as in the years before 2002, when the annual average growth of processor performance declined from 52% to 20% [1]. Since transistor size decreased and still continues to grow smaller, more complex logic can be integrated into the chip that is dedicated to increase the number of instructions that can be executed per clock cycle. For example, out-of-order execution or speculative execution with branch prediction further increase efficient usage of execution resources in a CPU. These techniques exploit instruction level parallelism (ILP) by overalapping the execution of multiple instructions.

However, the general direction of development has led to multi-core processors that integrate multiple CPU cores in a single chip. Multi-core CPUs can sustain multiple threads of execution simultaneously, which is referred to as thread level parallelism (TLP). Operating systems can use this, for example, to schedule multiple tasks to execute simultaneously on different cores. However, single applications that make use of TLP are more demanding to the programmer because of issues that arise from accessing shared data from multiple threads. Race conditions can lead to hard to find bugs in programs. By locking critical sections of code, other problems can arise like deadlocks or livelocks. A multi-threaded program may even have worse performance than a single-threaded program because of synchronisation overhead. It may even be difficult to find sections of code that can be parallelized to begin with.

A third form of parallelization is data level parallelism (DLP). Data is distributed across several CPUs, all of which execute the same instruction on a separate slice of data simultaneously (SIMD). A good example for a situation in a program where data level parallelism could be exploited by SIMD operations are loops, like shown in the following piece of code.

```
for (i=0; i<16; i++)
    a[i] = b[i] + c[i];
```

The loop sequentially takes every of the 16 elements of the arrays b and c, adds them, and stores the result in a. In a sequential mode of operation, for every iteration, the loop variable has to be incremented, two operands have to be loaded and added, the result has to be stored, and a conditional jump has to be made. This amounts to 16*6 = 96 operations for all iterations. Depending on the instruction set architecture (ISA), even more instructions will have to be executed. E.g., for an ISA that does not have autoincrement addressing modes, instructions for incrementing registers for load and store addresses may have to be inserted. A vector operation could achieve all that with a single instruction. One drawback of vector operations however is insufficient support through high level programming languages. Instead, specialized code has to be programmed in assembler language.

While most modern VLSI processor designs for standard computer hardware exploit all of the three techniques, they are not widely supported in soft-core processors intended to run on FPGA hardware. While exploiting instruction level parallelism often involves the use of complex logic and therefore is not primarily suited for FPGA applications, multiple simple cores can fit on an FPGA. Vector operations also tend to be well suited for implementation on FPGAs due to their highly parallel nature. A project that incorporates both the SIMD concept and parallelism on thread-level in a scalable design is the *ParaNut* project. It also tackles the problem of limited vector operation support in high level languages.

1.1 Requirements Specification

The goal of this work is the design of a functional VHDL implementation of a ParaNut processor that can be synthesized for FPGA hardware. The VHDL model should at least support the functionality of the ParaNut SystemC reference design, with one CPU core. To verify the correctness of the implementation, programs should be able to execute correctly. While it is difficult to verify the correct behaviour of the processor for all possible states it can take, it is sufficient that a set of programs which is later being used to evaluate the implemented ParaNut processor executes correctly. For example, correct behaviour can be asserted by examining output from programs on a terminal which shows that the program executed correctly. This requires embedding the ParaNut in a system-on-chip (Soc) that provides suitable output capabilities, e.g. a UART for connecting to a serial interface on a host PC. In order to be able to evaluate the ParaNut processor in terms of performance, a suitable performance measuring device needs to be implemented. Finally, resource usage and timing are to be examined for the given FPGA platform.

1.2 Structure of This Work

This work begins with an overview of existing soft-core processor architectures in chapter 2. While some of them are briefly outlined, the OpenCores project and the OpenRISC 1000 architecture, on which the instruction set of the ParaNut is based, are illustrated more clearly. Chapter 3 moves on to specify the special concept behind the ParaNut architecture. A SystemC implementation of a ParaNut processor is then outlined, as it will serve as a reference for the ParaNut VHDL implementation that is presented in this work. The core components of the ParaNut VHDL implementation are explained in chapter 4. A hardware unit that is developed to evaluate the performance of specific components that are critical to the design of the ParaNut architecture is introduced. Chapter 5 states the general ideas the VHDL code of the ParaNut complies to. A central configuration mechanism that enables the ParaNut VHDL model to be implemented according to different design goals is presented. Multiple aspects of the ParaNut VHDL implementation are evaluated in chapter 6. After introducing the FPGA platform and the system-on-chip used for evaluation, a set of benchmarks is run and the results are compared to an implementation of the OR1000 architecture, the OR1200. In the next step, different configurations of core components of the ParaNut are evaluated, in order to determine their influence on overall performance. The benchmarks are also run with multiple cores in order to verify the correctness of the design and measure performance for an increased workload. The chapter ends with an comprehensive examination of resource usage and timing for an FPGA. Based on the results, suggestions to possible improvements of the ParaNut implementation are made. Chapter 7 summarizes the results of this thesis. Concluding remarks are given in chapter 8, where suggestions on the further development of the ParaNut VHDL implementation are given.

2 Fundamentals

Before having a look at the ParaNut architecture, some of the many existing implementations of soft-core processors are briefly reviewed in section 2.1. The OpenRISC 1000 architecture, which defines the instruction set architecture for the ParaNut processor, is then highlighted in section 2.2.

2.1 State of The Art

This section outlines some of the existing processor architectures in terms of multi-core processing support and availability.

MicroBlaze is a soft-core IP processor from Xilinx that is targeted at FPGA platforms [2]. The hardware description for the MicroBlaze is not open source. Xilinx provides the Embedded Development Kit (EDK) which is used to configure and build a complete system-on-chip with the MicroBlaze. For FPGAs with limited resources, different aspects of the MicroBlaze can be configured in order to reduce area consumption, but otherwise decrease performance.

An example for an instruction set architecture (ISA) that is open and non-proprietary is the *SPARC* ISA, which was developed by Sun Microsystems. Several implementations of the SPARC architecture exist. With the OpenSPARC T1 and OpenSPARC T2, the hardware descriptions for two implementations of the SPARC ISA were released under open source licenses [3]. While the OpenSPARC T1 is based on the 32-bit SPARC V8 architecture, the OpenSPARC T2 is an implementation of the more recent 64-bit SPARC V9 architecture. Although the designs initially were not ready for use on FPGAs because of high area consumption, an area optimized design has enabled a variant of the OpenSPARC T1 to be synthesized and fit on a Xilinx FPGA [4].

Another implementation of the SPARC V8 ISA is the *Leon3* processor, developed by Aeroflex Gaisler [5]. It is part of the GRLIB IP library. The source code is released under the GNU GPL license and is targeted at implementation on FPGAs. The GRLIB also provides many IP cores that can be used to create a complete SoC design and an environment for synthesizing the design for FPGA target devices from different vendors. The Leon3 processor is highly configurable and can be optimized to application specific needs. The GRLIB also supports creating a Leon3 based multi-core processor system [6].

The *Parallella* [7] platform is a multi-core development platform that is built upon a ZYNQ 70x0 system-on-chip which features an ARM A9 dual-core CPU and an integrated FPGA. The parallella draws its highly parallel capabilities from a special multi-core accelarator chip, the *Epiphany* [8], which is developed by Adapteva. While the software development tools and libraries are based on open source software, the hardware of the Epiphany IP core is not and is only available integrated into the Parallella platform. The Epiphany architecture is a scalable multi-core architecture where multiple floating-point RISC CPUs are connected to a shared-memory over a 2D mesh network that is organized as a matrix. Computations for array-like data can be split into several tasks, each of them which is then processed by one of the RISC nodes. A message passing API is provided that can be used to share data among nodes.

2.2 OpenRISC

OpenCores is a worldwide effort for the development of open source hardware. The project started back in 1999 and its website [9] is a place for the OpenCores community to develop and share their projects with one another. At the time of this writing, it hosts a total of over 1100 projects, with a community of nearly 200.000 users and an increase of 2000 new users last month. As the name OpenCores implies, most projects are licensed under a copyright license like the GPL or the less restrictive LGPL or BSD license. This is in line with the idea of building a complex SoC that may consist of many IP cores, just like a piece of software that uses free libraries. One of OpenCores' main contributors, Julius Baxter, has developed a new license, the *Open Hardware Description License* [10]. It addresses particular issues that arise when re-using free hardware IP cores in own designs and that are not covered by existing copyright licenses such as the GPL or LGPL, and is specifically tailored for hardware description (RTL) source code [11]. Its long term goal is to both grow interest in open source hardware development as well as making it easier for companies to re-use open-source IP cores in proprietary designs.

2.2.1 OpenRISC Overview

OpenRISC is the prime example of projects of the OpenCores community. At the heart lies the OpenRISC 1000 (OR1K) architecture [12] which aims at the development of a highly configurable RISC CPU. It is released under the GNU General Public License. The OpenRISC 1000 architecture is an open architecture which means that every aspect of it is not only publicly available, but also allows for a multitude of implementations with different design goals. While some basic features of OR1K are mandatory, there are many optional features that let implementations greatly enhance their capabilities. Or, if, for example, simplicity and low area consumption are paramount, the design can be reduced to what is absolutely necessary. With some of the optional features, an OpenRISC compatible processor can be designed that supports all important features of today's microprocessors, being able to run full-sized operating systems such as Linux. The *OR1200* processor is an implementation of the OR1K architecture, which is however not designed to support multiple cores. Section 2.2.2 will highlight some of the details of the OR1K architecture.

In order to develop a fully functional system-on-chip, a peripheral system surrounding the CPU is needed. This system is realised in the form of another effort by OpenCores named *OpenRISC Reference Platform System-on-chip* (ORPSoC). It is centered around the open source hardware computer bus *Wishbone Bus*, interfacing different IP cores with each other, and includes the OR1200 implementation of the OR1K architecture. The Wishbone bus enables an OR1K based processor to use peripherals such as controllers for main memory, serial interface, debug interfaces, etc. Many of the IP cores found at OpenCores come with a Wishbone bus interface, ready for use in an ORPSoC project. Some of the basic components of a microprocessor system like a 16550 compatible UART, also part of the OpenCores project, as well as an implementation of the Wishbone Bus, are already included in the ORPSoC.

On top of providing a compilation of IP cores and interconnect for a complete SoC, ORPSoC has been ported to a range of FPGA boards. In fact, ORPSoC is intended to make the process of customising a SoC design and porting it to a new FPGA board as simple as possible. It therefore has facilities for simulating Verilog RTL designs as well as synthesising Verilog and VHDL source code for different target FPGA technologies.

The *OpenRISC GNU tool chain* [13] consists of the most important tools for compiling software from C/C++/Assembler source code as well as examining, modifying, executing, and debugging resulting binaries.

2.2.2 The OpenRISC 1000 Architecture

OR1K defines a RISC load-store architecture with support for both 32 and 64 bit wide registers and address space. It therefore generally suits medium to high performance applications and allows for powerful designs that can run modern operating systems like Linux. The following subsections will highlight features defined by the OR1K architecture that are particularly important to the implementation of the ParaNut processor.

2.2.3 Instruction Set Architecture

The OpenRISC 1000 instruction set architecture is a flexible architecture which consists of different instruction subsets which are either mandatory or optional. Table 2.1 lists the five instruction subsets.

Table 2.1: Summary of OR1K instruction subsets

Instruction Subset	Instructions featured
ORBIS32	32-bit integer instructions 32-bit load and store instructions Program flow instructions Special instructions Basic DSP instructions
ORBIS64	64-bit integer instructions 64-bit load and store instructions
ORFPX32	Single-precision floating-point instructions
ORFPX64	Double-precision floating-point instructions 64-bit load and store instructions
ORVDX64	Vector instructions DSP instructions

An OR1K compliant CPU must implement at least the ORBIS32 or ORBIS64 instruction subset. All other instruction subsets are optional. In each subset, there are Class I and Class II instructions of which instructions of Class I must always be implemented, and those of Class II are optional. An implementation may choose to implement any of the instructions in Class II. This allows for great flexibility in the implementation of the CPU. A detailed list of all instructions can be viewed in the OR1K architecture manual [12].

3 ParaNut

3.1 Overview

The ParaNut is a new project by Gundolf Kiefer from the University of Applied Sciences Augsburg. Its architecture is specified in the *ParaNut Architecture Description and Reference Manual* which can be obtained from the website of the Efficient Embedded Systems workgroup [14]. Designed to be an open and scalable multiprocessor system, the ParaNut instruction set is based on the OpenRISC 1000 specification. Therefore, compiler support is available from the start through the OpenRISC GCC tool chain. Since the ParaNut is targeted at FPGA implementations with different and often limited resource budgets, the ParaNut architecture does not aim at a complex design that employs techniques such as instruction level parallelism to exploit parallelism. Instead, a new concept of SIMD (single instruction, multiple data) vectorization is used to exploit parallelism in a way that is easy to implement by application programmers: Instead of having to implement code in assembler for specialized SIMD instructions, SIMD support in the ParaNut is available by means of programming in a high level language like C. Thread level parallelism is supported and is not as expensive in terms of logic usage but imposes more work on the application programmer, since critical code sections need to be properly secured. This chapter first shows the general architectural concept behind the ParaNut architecture. It then outlines the ParaNut SystemC model which is the reference implementation for the VHDL implementation that was designed in this work.

3.2 Hardware Architecture

Figure 3.1 shows an example of the architecture of a ParaNut processor.

A ParaNut processor is made up of at least 1 processor core, the Central processing unit (CePU). Besides the CePU a ParaNut can have one ore more co-processing units (CoPUs). They can have the same features as the CePU and all cores will then act as a multi-core CPU. However, a ParaNut CPU can run in different modes from 0 to 3.

A CePU can only run in mode 3, or set inactive (mode 0). In mode 3, a CePU or CoPU carries out instructions, handles interrupts and exceptions, and can execute privilieged instructions. In this example, CoPU 1 runs in mode 2 (Thread), which allows it to execute instructions, but interrupts and exception handling are disabled as are priviliged instructions. Mode 1 (linked mode, or vector mode) puts the CoPU under the control of the CePU. This means that it does not process instructions on its own but executes ALU operations under the control of the CePU. The registers of the CoPU can be seen as a slice of a vector CPU. This is how the SIMD concept of the ParaNut architecture is implemented. Mode 0 sets the CoPU inactive and it does not process instructions nor handle exceptions.

Since the ParaNut architecture is scalable, a CoPU does not have to support all modes up to 3. E.g., in order to build a highly parallel vector processor that does not need multiple autonomous CPU cores, CoPUs only have to support a maximum capability of 1. This introduces the capability of a core and reflects the maximum supported mode the CoPU can run in. A CoPU

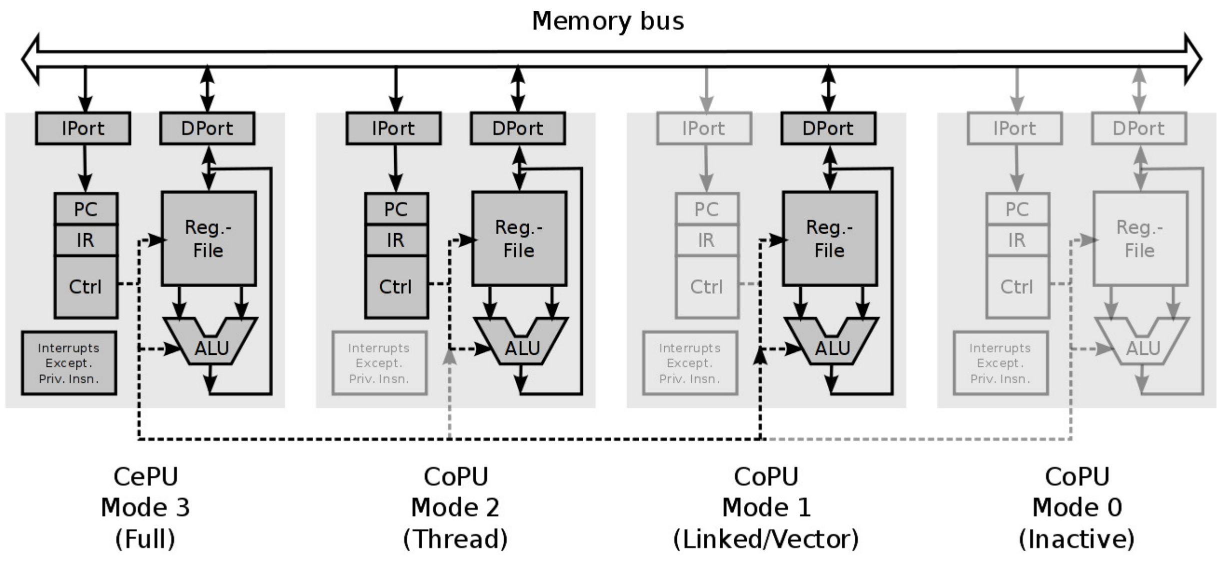

Figure 3.1: Architecture of a ParaNut processor (image courtesy of Gundolf Kiefer [15])

then only has to implement the hardware which is need to support its maximum capability. A CoPU with capability 2 does not need to implement interrupts and exception handling as well as privileged instructions, and a capability 1 CoPU additionally loses its internal controller and instruction port.

3.3 SystemC Model

The ParaNut SystemC model, developed by Gundolf Kiefer, is a cycle accurate implementation of the ParaNut architecture. It serves as the reference implementation for the VHDL model that has been developed in this work. At present, only CePUs are implemented in the model. Figure 3.2 shows the general modular structure of the ParaNut SystemC model for n CPU cores.

All CPUs are connected to the memory unit (MEMU). It has access to main memory via a Wishbone bus compatible interface and contains a unified instruction/data cache. A CePU consists of the 3 sub-modules execution unit (EXU), instruction fetch unit (IFU), and load/store unit (LSU). The IFU implements the instruction port and has a small buffer for instructions. The LSU implements the data port with a small buffer for store operations. Both IFU and LSU interface with the MEMU through read ports (RP) and write ports (WP). The IFU only has to read instructions and therefore only has one read port. The LSU needs to both read and write data, so it has one read and write port. Every CePU will therefore need two read ports and one write port. Both IFU and LSU are controlled by the EXU which implements the ORBIS32 instruction subset as well as all OR1K architecture specific registers and exception handling.

The SystemC model is verified with a test bench that can execute programs compiled for the OR1K architecture. The model also has means of profiling execution times for instructions. Although the SystemC model is a cycle accurate model of a ParaNut processor, not all modules have been implemented at the register transfer level (RTL). To be able to run the ParaNut on FPGA hardware, a synthesisable VHDL model has been developed in this work which is presented in chapter 4.

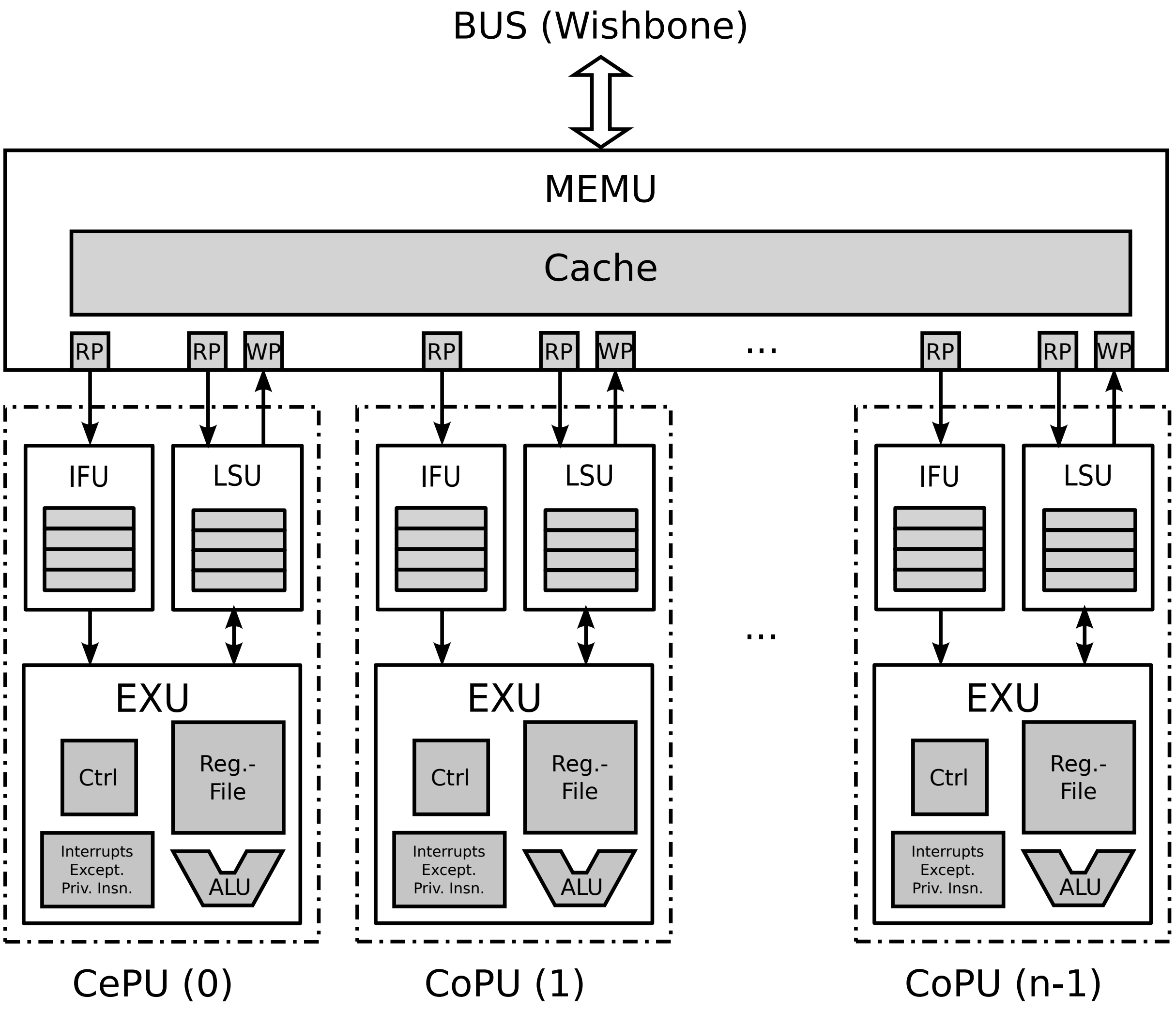

Figure 3.2: Modular structure of the ParaNut SystemC model

4 Hardware architecture

This chapter describes the hardware architecture of the ParaNut VHDL implementation that was developed in this work. It uses the ParaNut SystemC model which is introduced in chapter 3 as a reference design. The VHDL implementation sticks to the reference design as close as possible, especially where it is implemented at the register transfer level. Overall, the main components of the SystemC model have been adopted. While the VHDL model was developed, changes to its architecture have been transferred back to the SystemC model. At the time of this writing, only cores with the highest capability 3 are supported, which means they contain a full-featured EXU with instruction and data ports (also see figure 3.2). Thus, every additional core will generate EXU, IFU, and LSU. This should of course change in the future, where a CoPU with capability 2 will omit interrupt handling, exception processing, and privileged instructions and a CoPU with capability 1 will additionally lose its instruction port and all related logic. The following sections show detailed descriptions of the main modules of the ParaNut VHDL implementation.

4.1 Instruction Fetch Unit (IFU)

Fetching instructions in the ParaNut is handled by the instruction fetch unit (IFU). This is mostly done autonomously by the instruction fetch pipeline which consists of a buffer for program counter addresses and associated instructions. They are both configurable in the number of entries. Jumps and requests for new instructions are controlled externally by the EXU (see section 4.4). Figure 4.1 shows the general structure of the address and instruction buffers.

New addresses are constantly being generated every clock cycle by adding an offset to the last valid address and storing it in the buffer until the buffer is full. The first 3 of N entries (0..2) of the buffer represent previous (PPC), current (PC), and next (NPC) program counter values and are exposed to the EXU. When a jump occurs, the jump target address is loaded from the EXU into the 2nd entry of the address buffer. The IFU utilizes a read port interface to the MEMU to fetch instructions for valid addresses from the address buffer. A new address is therefore selected from the address buffer and stored in a register. Only when there is a jump instruction in the buffer, fetching new instructions is halted for the address that follows the jump delay slot (i.e. the jump target address). This is done for two reasons: First, the jump target address for an unconditional branch may not have been calculated yet. Second, it may not have been decided yet if a conditional branch is taken or not. In both cases, instructions for invalid addresses would be fetched which must not happen. Moreover, this prevents unnecessary memory accesses that could lead to delays for other ports trying to access the MEMU because of occupied resources. When the EXU requests a new instruction, the contents of the buffers are shifted by one place and the oldest entry is removed. Since an address is generated every clock cycle, a read request to the MEMU can be issued in every clock cycle, too, provided that it is served in the same clock cycle. The MEMU can theoretically do so by reading bursts using the bus interface.

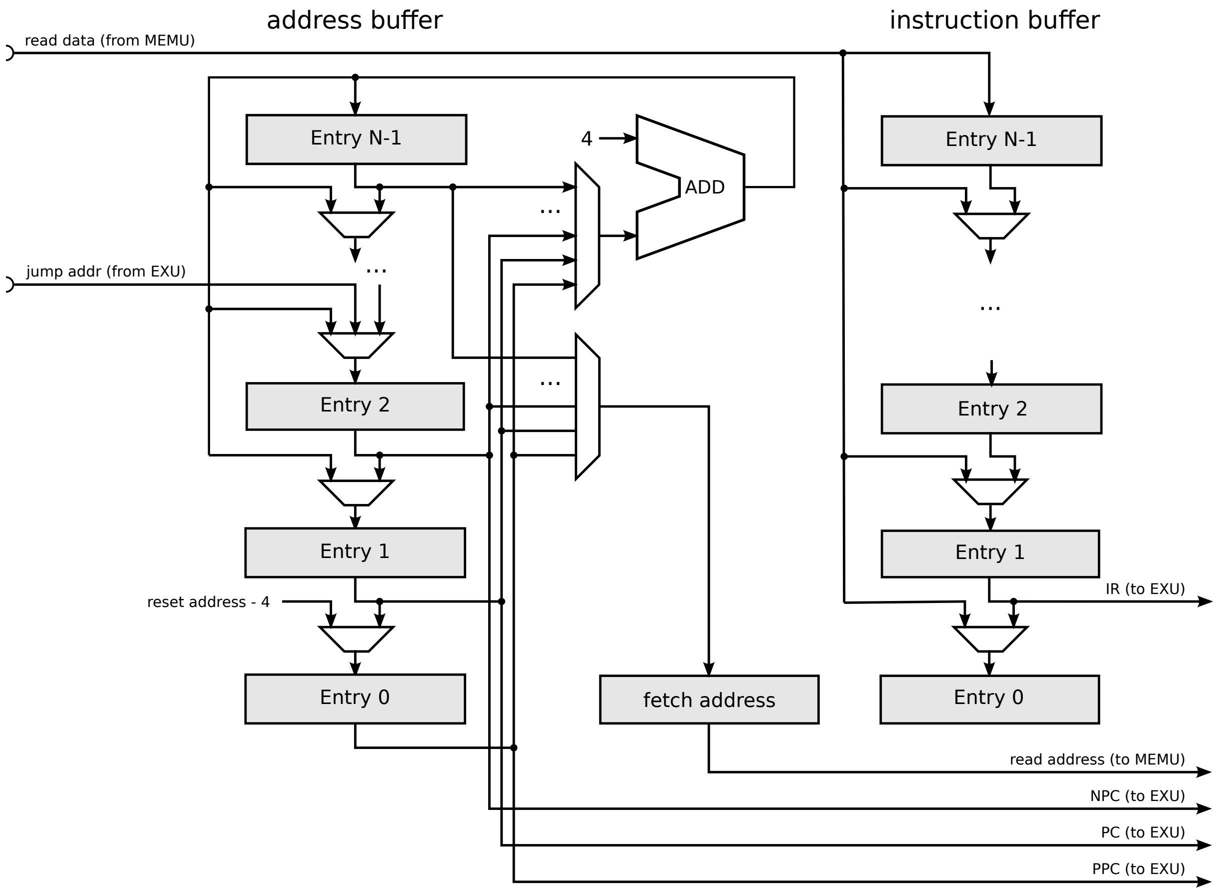

Figure 4.1: Block diagram of the address and instruction buffers of the IFU.

4.2 Load Store Unit (LSU)

The LSU serves all read and write requests to memory for the EXU and is designed to decrease write latency for store instructions. A write buffer that can be configured in the number of entries is used to this end. Figure 4.2 shows the layout of a buffer entry.

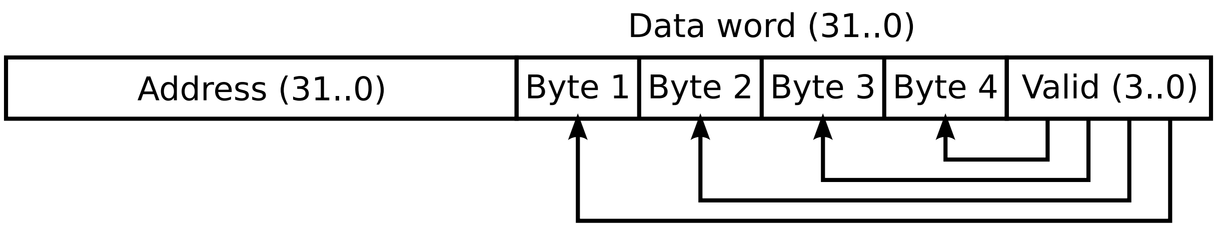

Figure 4.2: Layout of a write buffer entry of the LSU.

Every entry holds a 32-bit address, 32-bit data word, and a 4-bit valid tag. A valid bit set to ’1’ indicates that the corresponding byte of the data word in the buffer is to be written to memory. Every word that is written by the EXU is placed in the buffer first before writing to memory. This allows for writes to be completed in the same clock cycle as the write request to the LSU was issued if the buffer is not full. Valid bytes from words that are queued in the buffer and are to be written to memory must be forwarded to a read operation with the same address. Otherwise, modified bytes would not be catched and false data could be read. This can also help in reducing read accesses to the MEMU if a full hit (all of the requested bytes are valid) occurs. The block diagram in figure 4.3 shows how the write buffer is embedded into the LSU.

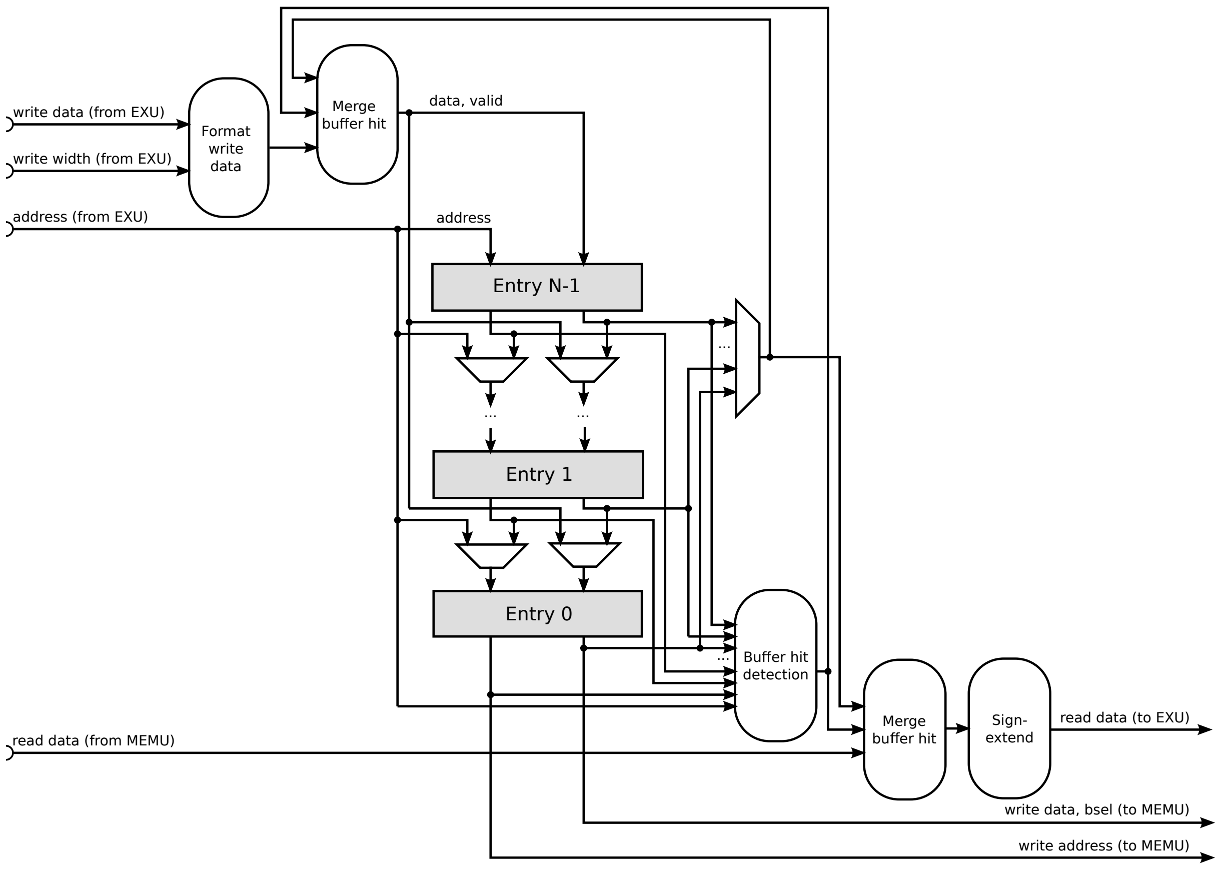

Figure 4.3: Block diagram of the LSU.

When writing, the buffer is first searched for a hit. A write hit occurs if an entry with the same address is already in the buffer and one of its bytes is marked valid. The existing entry will then be merged with the new write request by updating valid bits and associated bytes in the data word. An exception to this rule are writes to special addresses, i.e. I/O addresses. If, for example, a peripheral core needs to be reset by toggling a bit in one of its registers to '1' and back to '0' again, the first write (reset bit set to '1') could be merged with the second write (reset bit set to '0') and thus '1' would never be written. Therefore, write to special addresses need to wait until the buffer is flushed.

When reading, valid bytes that have a hit in the buffer are being forwarded. If all requested bytes are valid, the read request can be served completely from the write buffer and be completed in the same clock cycle that the request was made. Otherwise, the LSU forwards the request to memory. If there is a partial hit in the store buffer, modified bytes are additionally merged with the data that was read from memory.

4.3 Memory Unit (MEMU)

The memory unit is the central part for all read and write accesses to the memory hierarchy. Its general structure is shown in figure 4.4.

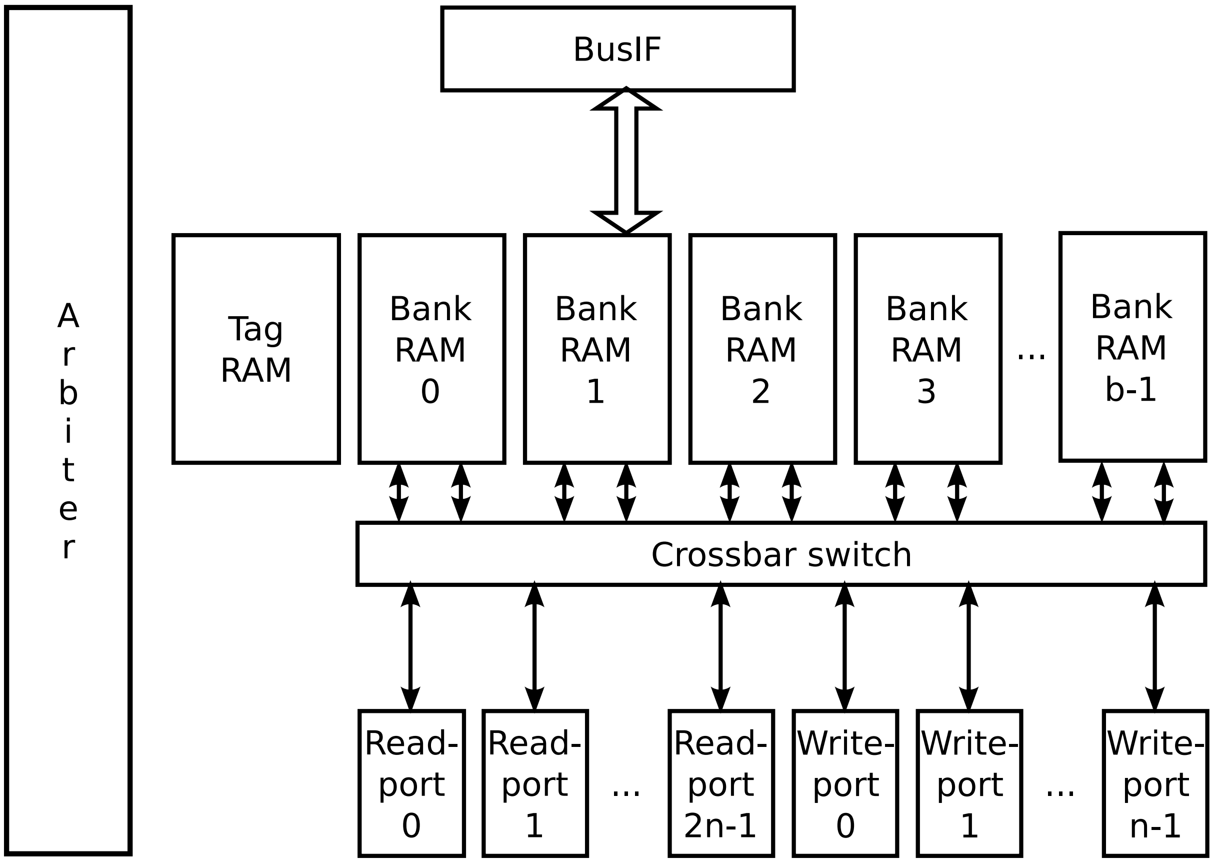

Figure 4.4: Block structure of the memory unit.

The MEMU provides access to the Wishbone bus and features a unified cache for both instruction and data memory. Read and write ports, which are employed by the IFU and LSU, share this single cache and the bus interface. As the name suggests, the arbiter module deals with arbitrating accesses to shared resources, which are the bus interface, tag RAM, and bank RAM. Read ports and write ports first have to request permission from the arbiter in order to access a shared resource. If the access is granted, this is signaled to the corresponding port with a grant signal. The bus interface also deals with reading and writing cache lines during a cache line refill and therefore has to request access for tag and bank RAM from the arbiter. The arbiter is a mealy type state machine and can grant access in the same clock cycle as the request was made. Accesses are granted according to different priorities which are listed in table 4.1, beginning with the highest priority (0).

Table 4.1: Arbiter priorities

Priority	Module
0	bus interface
1	read port (data)
2	read port (instruction)
3	write port (data)

Cache memory is organized in banks with a width of 4 bytes. Every bank can be accessed individually. Since the number of ports for a single bank is limited, accesses need to be arbitrated. Tag RAM is duplicated for each CPU core which enables CPUs to read tags for different addresses in parallel. However, writing to tag RAM is restricted to one access at a time.

A cached read access by a read port will take at least 2 clock cycles. In the first clock cycle, it will request access to the tag RAM from the arbiter. If the access is granted in that cycle, the tag can be read in the next clock cycle. If it is a hit, the read port will request the corresponding bank from the arbiter and can immediately acknowledge to the LSU or IFU if the access is granted. Since the access to cache memory is pipelined, data will arrive in the clock cycle after the request was acknowledged. If there was no hit in the cache, the read port will then request and employ the bus interface to fetch the data from main memory, which is also done for uncached read accesses. A read port can also complete a read access in one clock cycle, if it happens to catch an incoming data word that is transferred from main memory to the cache by the bus interface (bus interface hit). This is due to the bus interface forwarding read data when reading from main memory. If a burst access to main memory yields a new data word in every clock cycle, the IFU can use this to quickly refill its instruction buffer.

A cached write access will start by the same actions as a read access, by determining if the address to be written is already in the cache, and whether data has to be fetched from main memory first. However, any writer to the cache needs to have exclusive access to a whole cache line through means of a line lock mechanism, which is administered by the arbiter. This mechanism is employed a) by the bus interface when replacing a whole cache line with data from main memory and b) by a write port when writing to bank RAM. To this end, only one of all write ports or the bus interface is allowed to modify the same cache line at a time. The line lock mechanism also prevents conflicting write accesses to tag RAM. These restrictions are necessary in order to guarantee cache coherency. Non-conflicting accesses to the same cache line are read-only accesses and accesses to different neighbouring addresses. An address is neighboured if it has an offset of one word (i.e. 4 bytes) to the previous/next address and therefore addresses a different bank. It is intended to allow concurrent access to a single cache line by all CPUs during vectorized operation, where every core operates on a chunk of data with an address offset of one word. E.g., for a 4-fold vectorized operation on data starting at address 0x2000 that is computed by 4 CPUs, the addresses for the CPUs and their respective read/write ports will be as shown in table 4.2.

Table 4.2: Vectorized access to cache banks

Core	**Address**	**Bank**
CePU(0)	0x2000	0
CoPU(1)	0x2004	1
CoPU(2)	0x2008	2
CoPU(3)	0x200c	3

A crossbar switch is used to route any of the bank RAM ports to a read or write port. Since bank memory is implemented as two-port block RAM on the FPGA, reads from two different addresses of a single bank can be routed through the switch at a time.

4.4 Execution Unit (EXU)

The EXU implements ParaNut architecture specific registers and the execution pipeline for the ORBIS32 instruction subset. A comprehensive list of all implemented instructions can be found in the *ParaNut processor architecture manual* [14]. A full set of 32 registers is implemented in the EXU. Alternative sizes for register files are not yet supported by the VHDL implementation. This section explains the general structure of the execution pipeline and lists detailed information about implemented special purpose registers.

4.4.1 Pipeline Structure

The pipeline structure of the EXU is shown in figure 4.5. The two stages in the EXU pipeline are the instruction decode stage (ID) and the execute stage (EX). Their range of control is enclosed by dotted outlines. Pipeline registers are indicated as gray rectangles with the name of the stage above them. The results of every stage are saved in the pipeline registers with the same name as the stage.

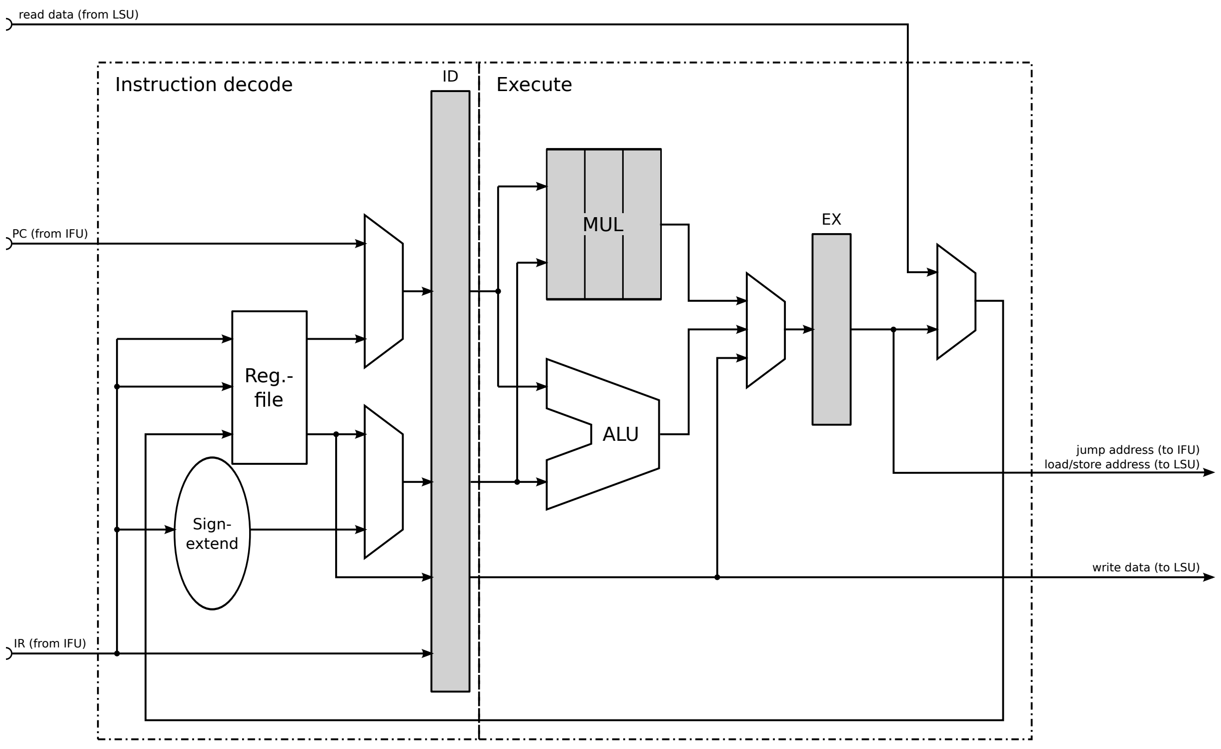

Figure 4.5: Block structure of the execution pipeline.

The first stage, ID, directly takes the program counter (PC) and instruction register (IR) outputs from the IFU. The IR is decoded to instruction set independent operations which are stored in the ID stage's registers. At the same time, the register file is accessed and sign-extended immediate values are computed. Two values are then selected for the input operands of the EX stage, depending on the value of the IR. The first operand is selected among PC and the value from the register file that is addressed by the IR for operand A (regfile[IR.A]). The second operand is multiplexed between regfile[IR.B] and the sign-extended immediate from the IR. Both are then stored in the ID stage's registers and will later directly serve as input operands to the ALU, shift, or multiplication operations.

Next in line is the EX-stage. It is a multi-cycle implementation which takes the decoded output from the ID-stage. Any result from ALU, shift, or multiplication operations is saved in a result register in the EX-stage which is the only additional register for that stage. All other operands as well as pipeline control signals and program counter can be taken from the ID-stage registers. These registers will hold their contents for as long as the multicycle operation in the EX-stage has not been completed. The minimum number of clock cycles needed to complete an instruction is 2 clock cycles for ALU and shift operations as well as jumps/branches (*Note:* The delay of shift operations is 1 clock cycle only when not using a serial shift implementation. Otherwise, the delay is proportional to the number of shifted bits.). This reflects the minimum delay for cached memory read accesses in the MEMU, which is also 2 clock cycles. Hence, it can be expected that new instructions are delivered by the IFU every 2nd clock cycle at most. Thus, implementing a pipeline that can handle instructions in 1 clock cycle would be unnecessary from a performance point of view and actually result in a more complex design with higher resource usage. However, the VHDL source code has been written with an implementation of more stages in mind as this can be achieved through merely modifying control of the pipeline stages. In the first clock cycle, the result of the current instruction is computed and stored in the EX-stage register. The ALU is used for arithmetic logical operations as well as address calculation for load and store instructions and jumps. For ALU and shift operations, the second clock cycle will take the result from the register and store it in the register file. During this clock cycle, the next instruction can already be decoded and placed in the ID-stage registers. The register file is implemented with forwarding logic in order to avoid pipeline data hazards that arise in this situation. Addresses for jumps and branches as well as load and store operations are calculated in the same manner. While a jump/branch can be handled within a total of 2 clock cycles, load and store operations must wait until the LSU has served the request to memory and therefore stall the pipeline. After the requested data has been served, it is stored in the register file. Other operations like multiplication and shifting (when not using a one clock cycle implemenatation like a barrel shifter) that cannot complete in one clock cycle will also stall the pipeline. Table 4.3 summarizes the clock cycle times needed for execution of different types of instructions. It is further examined in section 6.2, whether and how well these delays can be reflected by the actual implementation in hardware.

Exceptions are handled by an additional stage that is normally not interfering with the flow of instructions in the pipeline. Only when an exception occurs the pipeline is stalled until all issued instructions have completed and the exception can be handled. An instruction is issued when it is committed from the ID to the EX stage. Therefore, the contents of the IF stage registers are preserved and NOPs are inserted in the ID stage registers until the exception has been handled, i.e. until the jump to the exception handler was performed.

Table 4.3: CPI for different instruction types

Instruction type	**CPI**
program flow	2
arithmetic & logical	2
shift (barrel shifter)	2
shift (serial shifter)	number of shifts + 2
multiplication	number of multiplier pipeline stages + 2
load	min. 4 (cached access)
store	min. 3

4.4.2 General Purpose Registers (GPRs)

The EXU implements one register file with a full set of 32-bits wide general purpose registers. Register R0 is hard-wired to zero. The GPRs are not mapped to the SPR space.

4.4.3 Special Purpose Registers (SPRs)

All special-purpose registers are implemented in the EXU. Unimplemented registers read a zero value. As opposed to the OR1K architecture manual, the SR[SM] and SR[SUMRA] bits, which are used to implement a privileged mode, are not decoded. Thus, all SPRs are always readable and writable. However, for operating systems that run processes with a privileged CPU mode the SM and SUMRA bits need to be implemented. Table 4.4 lists the implemented SPRs.

Table 4.4: ParaNut Special Purpose Registers (SPRs)

GRP	REG	Name	Mode	Description
0	0	VR	R	Version register
0	1	UPR	R	Unit Present register
0	2	CUPCFGR	R	CPU Configuration register
0	5	DCCFGR	R	Data Cache Configuration register
0	6	ICCFGR	R	Instruction Cache Configuration register
0	16	NPC	RW	NPC (next PC) mapped to SPR space
0	17	SR	RW	Supervision register
0	18	PPC	R	PPC (previous PC) mapped tp SPR space
0	21:28	ISR0:ISR7	R	Implementation-specific registers
0	32:47	EPCR0:EPCR15	R	Exception PC registers (all mapped to single register)
0	48:63	EEAR0:EEAR15	R	Exception EA registers (all mapped to single register)
0	64:79	ESR0:ESR15	R	Exception SR registers (all mapped to single register)
0	1024:1055	GPR0:GPR31	RW	GPRs mapped to SPR space
24	0	PNCPUS	R	ParaNut Number of CPUs
24	1	PNM2CAP	R	ParaNut Mode-2 Capability
24	2	PNCPUID	R	ParaNut CPU ID
25	0	PNHCTRLR	RW	ParaNut Histogram Control register
25	64..127	PNHISTD64..127	R	ALU instruction histogram registers
25	128..191	PNHISTD128..191	R	SHIFT instruction histogram registers
25	192..255	PNHISTD192..255	R	MUL instruction histogram registers
25	256..319	PNHISTD256..319	R	LOAD instruction histogram registers
25	320..383	PNHISTD320..383	R	STORE instruction histogram registers
25	384..447	PNHISTD384..447	R	JUMP instruction histogram registers
25	448..511	PNHISTD448..511	R	OTHER instruction histogram registers
25	512..575	PNHISTD512..575	R	IFU histogram registers
25	576..639	PNHISTD576..639	R	Cache line read histogram registers
25	640..703	PNHISTD640..703	R	Cache line write histogram registers
25	704..767	PNHISTD704..767	R	IFU cache read hit histogram registers
25	768..831	PNHISTD768..831	R	IFU cache read miss histogram registers
25	832..895	PNHISTD832..895	R	LSU cache read hit histogram registers
25	896..959	PNHISTD896..959	R	LSU cache read miss histogram registers
25	960..1023	PNHISTD960..1023	R	LSU cache write hit histogram registers
25	1024..1087	PNHISTD1024..1087	R	LSU cache write miss histogram registers
25	1088..2047	PNHISTD1088..2047	R	Custom histogram units

From the ParaNut architecture, only the read-only registers PNCPUS and PNM2CAP are implemented. PNM2CAP currently returns an all-ones value since no CoPUs are supported yet. This and because there is no hardware synchronisation yet between CPUs, the PNCE, PNLM, PNX and PNXID0..PNXID31 registers are not yet implemented. The VHDL implementation introduces the PNCPUID register. It gives every CPU a unique ID which can be read via the l.mfspr instruction and can be used for implementing a simple synchronisation mechanism in software. This has been done in order to run software on a ParaNut processor with multiple CPU cores.

SPR group 25 addresses the ParaNut Histogram Control register (PNHCTRLR) and all ParaNut histogram units (PNHISTD64..2047). The unit implements a performance counter for different types of events and is described in more detail in section 4.4.6. Every unit has a reserved address range of 64 entries and are globally controlled by the HEN bit of the ParaNut Histogram Control register (PNHCTRLR).

Version Register (VR)

The ParaNut uses the now deprecated Version Register. Future versions should use registers AVR and VR2 for version information. Table 4.5 lists the fields of the VR.

Table 4.5: ParaNut Version Register (VR)

Bit(s)	Name	Mode	Value	Description
31:24	VER	R	0x1f	Version (0x1f = ParaNut)
23:16	CFG	R	0	Configuration (reserved for future use)
15:7	-	R	0	(reserved)
6	UVRP	R	0	Updated VRs (AVR, VR2) present
5:0	REV	R	0	Revision

Unit Present Register (UPR)

Except for instruction and data caches, no optional units are present. Table 4.6 shows the fields of the UPR.

Table 4.6: ParaNut Unit Present Register (UPR)

Bit(s)	Name	Mode	Value	Description
31:24	CUP	R	0	Custom Units Present
23:11	-	R	0	(reserved)
10	TTP	R	0	Tick Timer Present
9	PICP	R	0	Programmable Interrupt Controller Present
8	PMP	R	0	Power Management Present
7	PCUP	R	0	Performance Counters Unit Present
6	DUP	R	0	Debug Unit Present
5	MP	R	0	MAC Present
4	IMP	R	0	Instruction MMU Present
3	DMP	R	0	Data MMU Present
2	ICP	R	1	Instruction Cache Present
1	DCP	R	1	Data Cache Present
0	UPR	R	1	UPR Present

CPU Configuration Register (CPUCFGR)

Table 4.7 lists the fields of the CPUCFGR.

Table 4.7: ParaNut CPU Configuration Register (CPUCFGR)

Bit(s)	Name	Mode	Value	Description
31:24	-	R	0	(reserved)
14	AECSRP	R	0	Arithmetic Exception Control Register (AECR) and Arightmetic Exception Status Register (AESR) present
13	ISRP	R	0	Implementation-Specific Registers (ISR0-7) Present
12	EVBARP	R	0	Exception Vector Base Address Register (EVBAR) Present
11	AVRP	R	0	Architecture Version Register (EVBAR) Present
10	ND	R	0	No Delay-Slot
9	OV64S	R	0	ORVDX64 Supported
8	OF64S	R	0	ORFPX64 Supported
7	OF32S	R	0	ORFPX32 Supported
6	OB64S	R	0	ORBIS64 Supported
5	OB32S	R	1	ORBIS32 Supported
4	CGF	R	0..1	Custom GPR File 0: GPR file has 32 registers 1: GPR file has 16 registers
3:0	NSGF	R	0	Number of Shadow GPR Files

Data/Instruction Cache Configuration Register (ICCFGR/DCCFGR)

Table 4.8 lists the fields of the DCCFGR. Since the ParaNut has a unified instruction/data cache this table also applies to the Instruction Cache Configuration Register (ICCFGR). The DCCFGR is mapped to the ICCFGR.

ParaNut Histogram Control Register (PNHCTRLR)

This register controls the histogram feature of the ParaNut. Table 4.9 shows the fields of the register.

Table 4.8: ParaNut Data Cache Configuration Register (DCCFGR)

Bit(s)	Name	Mode	Value	Description
31:15	-	R	0	(reserved)
14	CBWBRI	R	0	Cache Block Write-Back Register Implemented
13	CBFRI	R	0	Cache Block Flush Register Implemented
12	CBLRI	R	0	Cache Block Lock Register Implemented
11	CBPRI	R	0	Cache Block Prefetch Register Implemented
10	CBIRI	R	0	Cache Block Invalidate Register Implemented
9	CCRI	R	1	Cache Control Register Implemented
8	CWS	R	1	Cache Write Strategy
7	BS	R	0..1	Cache Block Size 0: Cache Block size 16 bytes or less (OR1K: exactly 16) 1: Cache Block size 32 bytes or more (OR1K: exactly 32)
6:3	NCS	R	0..15	Number of Cache Sets (cache blocks per way) 0: DC has one set ... 15: DC has 32768 sets
2:0	NCW	R	0..2	Number of Cache Ways 0: DC has one way (direct-mapped) 1: DC has two ways 2: DC has four ways

Table 4.9: ParaNut Histogram Control Register (PNHCTRLR)

Bit(s)	Name	Mode	Value	Description
31:1	-	R	0	(reserved)
0	HEN	R/W	0..1	Histogram Enable register 0: Disable histogram performance counting 1: Enable histogram performance counting

4.4.4 Interrupts and Exception Handling

Exceptions are handled according to the ParaNut architecture manual [14]. Since synchronisation mechanisms between CePUs and CoPUs are not yet implemented, there is no support for CoPU exceptions. If an exception occurs that is not caused by an instruction, all issued instructions complete before the exception is handled. Support for exceptions listed in table 4.10 exists, though no optional units are implemented yet. Fast context switching is not supported because only one set of exception registers is implemented.

Table 4.10: Supported Exceptions

Exception	ID	Description
Reset	0x1	Caused by hardware reset.
Tick Timer	0x5	Tick Timer interrupt asserted (optional, Tick Timer Unit not implemented yet).
Alignment	0x6	Load/store access to naturally not aligned location.
Illegal Instruction	0x7	Illegal instruction in the instruction stream.
External Interrupt	0x8	External interrupt asserted.
System Call	0xC	System call initiated by software.
Trap	0xE	Caused by the l.trap instruction or by debug unit.

4.4.5 Wishbone Bus Interface (BUSIF)

The bus interface module provides access to a Wishbone B4 compliant Bus [16]. Standard single read/write cycles are employed for direct (uncached) reads and writes. Depending on the configured number of cache banks, cache line refills are either performed using a Standard Block read/write cycle or a registered feedback bus cycle. The registered feedback bus cycle can be used for burst accesses with 4, 8, or 16 words per burst. In all other cases, a standard block cycle is used. Table 4.11 lists the information required by the Wishbone specification rule 2.15 [16] for Wishbone compatible IP cores.

Table 4.11: Wishbone Datasheet for the ParaNut Wishbone bus interface

Description	Specification
General description:	32-bit MASTER interface for CPU IP core
Revision level:	B4
Supported cycles:	MASTER, SINGLE READ/WRITE MASTER, BLOCK READ/WRITE MASTER, INCREMENTING BURST READ/WRITE
Optional ERR_I support:	not supported
Optional RTY_I support:	not supported
Tag support	not supported
Data Port, size:	32 bits
Data Port, granularity:	8 bits
Data Port, maximum operand size:	32 bits
Data transfer ordering:	Big endian and/or little endian
Data transfer sequencing:	Undefined

Table 4.11: Wishbone Datasheet for the ParaNut Wishbone bus interface

Description	Specification
Defined signal names	CLK_I, RST_I, ACK_I, ERR_I, RTY_I, DAT_I CYC_O, STB_O, WE_O, SEL_O, ADR_O, DAT_O, CTI_O, BTE_O

4.4.6 ParaNut Histogram Unit

The ParaNut Histogram Unit is a custom unit that was developed for evaluating the ParaNut VHDL implementation. It was used to generate the results for the detailed performance analysis that is conducted in chapter 6. The unit is able to generate a histogram of the number of clock cycles that is needed for certain events that can be triggered at arbitrary points inside the ParaNut CPU. Additionally, the minimum and maximum number of clock cycles as well as the total number of clock cycles and number of events are registered. This allows for a detailed performance analysis of different aspects of the architecture. The units are read-only and are accessible from the SPR space by the `l.mfspr` instruction. All implemented units and their addresses are listed in table 4.4. Figure 4.6 shows a block diagram of the core components of a histogram unit.

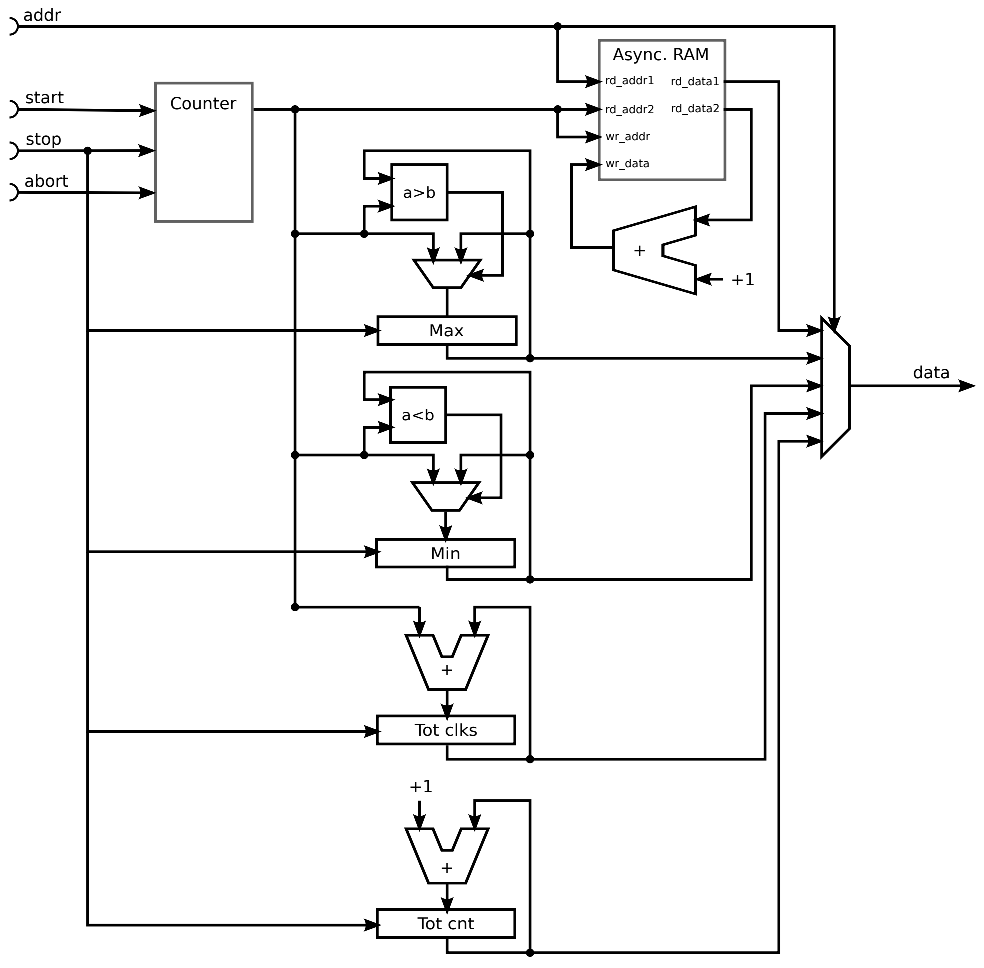

Figure 4.6: Block diagram of a histogram unit

The histogram unit is controlled by the 3 signals “start”, “stop”, and “abort”. The beginning of an event is signaled by start = ’1’, and can be stopped with stop = ’1’. This updates the minimum, maximum, and total number of clock cycles and registered events. Additionally, the bin with the corresponding number of clock cycles is incremented. If an event is not to be counted anymore after it has been started, it can be aborted with abort = ’1’. In the same clock cycle that an event is stopped, the next event can be started.

The unit also has an address input and data output. Addresses 0..59 will address the 60 bins of the unit. Addresses 60..63 address the minimum, maximum, total number of events, and total number of clocks registers, in that order. All data can be read asynchronously on the data output. If the number of clock cycles used for an event exceeds the maximum number of bins, it is put into the last bin (i.e. 60).

5 ParaNut VHDL implementation

In order to be able to evaluate the ParaNut architecture, a synthesizable RTL VHDL model of the ParaNut was made in the context of this work. Since it is the initial implementation of the VHDL model, some of the main conceptual ideas behind it are outlined.

The ParaNut in this work is completely implemented in the VHDL programming language and is targeted at synthesis on FPGA technology. In fact, throughout this work, it was successfully synthesised for a Virtex-5 FPGA from Xilinx. Although only tested with this specific FPGA, the code does not use any target technology specific components. Instead, it is designed to be as cross-platform as possible and therefore uses inference for design elements as much as possible. Besides this, all configuration is done via a central configuration file and allows for a generic design that can be tuned to suit application specific needs.

5.1 General Code Structure

The ParaNut VHDL model is intended to support the same generic configuration mechanism as the ParaNut SystemC reference model. Therefore, it sticks to it as closely as possible. All module names of the original SystemC model have been adopted as well as most signal names, as far as possible. However, the design follows the principles of the structured VHDL design method as proposed by Jiri Gaisler [17]. Therefore, nearly all signals used for connecting entities and registers have been grouped into VHDL records to help readability as well as maintainability. Entity inputs and outputs, except for `clk` and `reset` signals, are grouped into input and output records, respectively. A record type named “registers” is also used for all registers in the module. It is without exception instantiated with the symbol “`r`”. Furthermore, modules utilize the two process design method that is also proposed in the article: A combinational process is used to generate output signals and signals all register contents for the next clock cycle to a second, clocked process. Every module and its associated input/output record is included in a package, so no component declarations are necessary in architecture descriptions of modules. The main library name for all ParaNut modules is `paranut`.

The ParaNut VHDL model consists of the same functional modules as the SystemC model and their names have been adopted as well. The main entity of a module is named just like the package name, but with a preceding small letter “m”. E.g., if a module is named “exu”, then its main entity is called “mexu”. Modules that interface with the MEMU over a read or write port additionally must include the “memu_lib” package. It lists type definitions for read and write ports and supporting functions useful when working with the MEMU.

5.2 Configuration Interface

Since the ParaNut design is meant to be scalable, a central configuration mechanism is provided with the VHDL model. It resides in the file `paranut_config.vhd`. The following subsections show the aspects that can be configured for each module within that file. It is embedded in a package with the name `paranut_config` and is included in all modules that can be configured by this file.

5.2.1 Simulation Configuration Options

The configuration file contains some options that do only affect simulation. They are mainly useful for debugging purposes and generate additional debug output. The options are listed in table 5.1. Any option can either be set to `true` to activate or `false` to deactivate it.

Table 5.1: Simulation Configuration Options

Symbol	Type	Description
CFG_DBG_INSN_TRACE	boolean	This option enables/disables generation of an instruction trace for all CPU cores.
CFG_DBG_LSU_TRACE	boolean	This option enables/disables debug output for data accesses to the MEMU.
CFG_DBG_BUS_TRACE	boolean	This option enables/disables debug output for accesses to main memory and I/O addresses.
CFG_DBG_TRAM_TRACE	boolean	This option enables/disables debug output for accesses to the cache tag RAM internal to the MEMU.
CFG_DBG_BRAM_TRACE	boolean	This option enables/disables debug output for accesses to the cache bank RAM internal to the MEMU.

5.2.2 General Configuration Options

The configuration options listed in table 5.2 affect both simulation and synthesis.

Table 5.2: General Configuration Options

Symbol	Type	Description
CFG_NUT_CPU_CORES_LD	natural	The number of CPU cores that will be generated in the design. 0: 1 CPU core 1: 2 CPU cores ...
CFG_NUT_MEM_SIZE	natural	The size of main memory in the system in bytes. This has to be set for both simulation as well as synthesis to the correct size. However, this may not be set too high for simulation, as some tools don't behave well when simulating huge amounts of memory.

Table 5.2: General Configuration Options

Symbol	Type	Description
CFG_NUT_LITTLE_ENDIAN	boolean	Defines the byte ordering of the bus interface. true: Little endian byte ordering false: Big endian byte ordering
CFG_NUT_HISTOGRAM	boolean	Enables or disables the generation of histogram units.

5.2.3 IFU Configuration Options

Table 5.3 lists configuration options for the IFU.

Table 5.3: IFU Configuration Options

Symbol	Type	Description
CFG_IFU_IBUF_SIZE	natural (2..4)	The size of the instruction buffer. 2: 4 entries 3: 8 entries 4: 16 entires

5.2.4 LSU Configuration Options

Table 5.4 lists the configuration options for the LSU.

Table 5.4: LSU Configuration Options

Symbol	Type	Description
CFG_LSU_SIMPLE	boolean	Generate a simple LSU without store buffer
CFG_LSU_WBUF_SIZE	natural (2..4)	The size of the store buffer (only effective if CFG_LSU_SIMPLE is false). 2: 4 entries 3: 8 entries 4: 16 entires

5.2.5 MEMU Configuration Options

Configuration options listed in table 5.5 affect the size of the cache.

Table 5.5: MEMU Configuration Options

Symbol	Type	Description
CFG_MEMU_CACHE_BANKS_-LD	natural (1..4)	The number of cache banks. Every bank has a data width of 32 bits. 1: Cache has 2 banks ... 4: Cache has 16 banks
CFG_MEMU_CACHE_SETS_LD	natural (1..15)	The number of cache sets. 1: Cache has two sets ... 15: Cache has 32768 sets
CFG_MEMU_CACHE_WAYS_LD	natural (0..2)	The number of cache ways. 0: Cache is direct-mapped 1: Cache is two-way set associative 2: Cache is four-way set associative
CFG_MEMU_CACHE_-REPLACE_LRU	natural (0..1)	Selection method for cache line replacement. 0: Cache uses random replacement (LFSR-based) 1: Cache uses LRU replacement
CFG_MEMU_ARBITER_-METHOD	natural (-1..15)	Arbitration method for arbiter module inside MEMU. -1: Random replacement (LFSR-based) 0..15: Round-robin arbitration that switches every $2^{0..15}$ clocks

5.2.6 EXU Configuration Options

The configuration options listed in table 5.6 can be used to choose between different types of implementations of modules in the EXU.

Table 5.6: EXU Configuration Options

Symbol	Type	Description
CFG_EXU_SHIFT_IMPL	natural (0..2)	Controls implementation of the shift module. 0: Implement a serial shifter (smallest area, slow) 1: Implement a generic shifter inferred by synthesis tools. 2: Implement a barrel shifter (more area, fast)

Table 5.6: EXU Configuration Options

Symbol	Type	Description
CFG_EXU_MUL_PIPE_STAGES	natural (1..5)	Number of pipeline stages of the embedded multiplier module. 1: Infer multiplier with 1 pipeline stage . . . 5: Infer multiplier with 5 pipeline stages

6 Evaluation

This chapter shows the results of the evaluation that was performed with the presented ParaNut implementation. The evaluation focused on three distinct aspects with the following set of experiments:

- Benchmarks: CoreMark, Dhrystone, simple merge sort
- Detailed performance analysis using histogram units.
- FPGA resource usage and timing results for different configurations of the ParaNut

An xc5vlx110t-ff1136-1 Virtex-5 FPGA was used to evaluate the ParaNut VHDL implementation. It features 17,280 slices, each containing four look-up tables (LUTs) and flip-flops for a total of 69,120 slice LUTs and flip-flops. It has 666 KiB of block RAM distributed over 148 block ram cells with a capacity of 36 kbits each. 48 DSP slices can be used for multiplication of 25-bit by 18-bit operands with a 48-bit result. Synthesis was performed using the Xilinx ISE tools 14.5. The following XST synthesis optimization options were used:

- -opt_mode Speed
- -opt_level 1

6.1 Benchmarks

6.1.1 Evaluation Platform

For the evaluation of the *ParaNut* on FPGA hardware, a peripheral system for the ParaNut processor was necessary. This system is provided by the ORPSoC project from the OpenCores project. Figure 6.1 shows the components of the system-on-chip that was built around the ParaNut and used throughout evaluation.

The SoC is centered around a Wishbone bus arbiter with an accompanying byte-wide bus. All components shown are connected through a single bus system. Both the ParaNut processor and the debug unit have a Wishbone master interface and can initiate read and write access to the slaves. The debug unit is used to remotely connect to the SoC via a JTAG interface and load program data into DDR2 memory. For this task, the GNU Project debugger (GDB) was used. At present, no other debugging functions like breakpoints, single-stepping, etc. are supported because of the debug unit missing in the ParaNut VHDL implementation. For getting printouts on a terminal via serial connection, a 16550 compatible UART is used. The GPIO module can additionally provide input and output possibilities like switches or LEDs. Since the ParaNut does not yet implement a tick timer unit, an external timer core is used for measuring time in benchmarks. It has a granularity of 1 clock cycle. Although access to the timer is via the bus and therefore is not as fast as for an internal tick timer unit, it is sufficient for the demands of the benchmark time measurements.

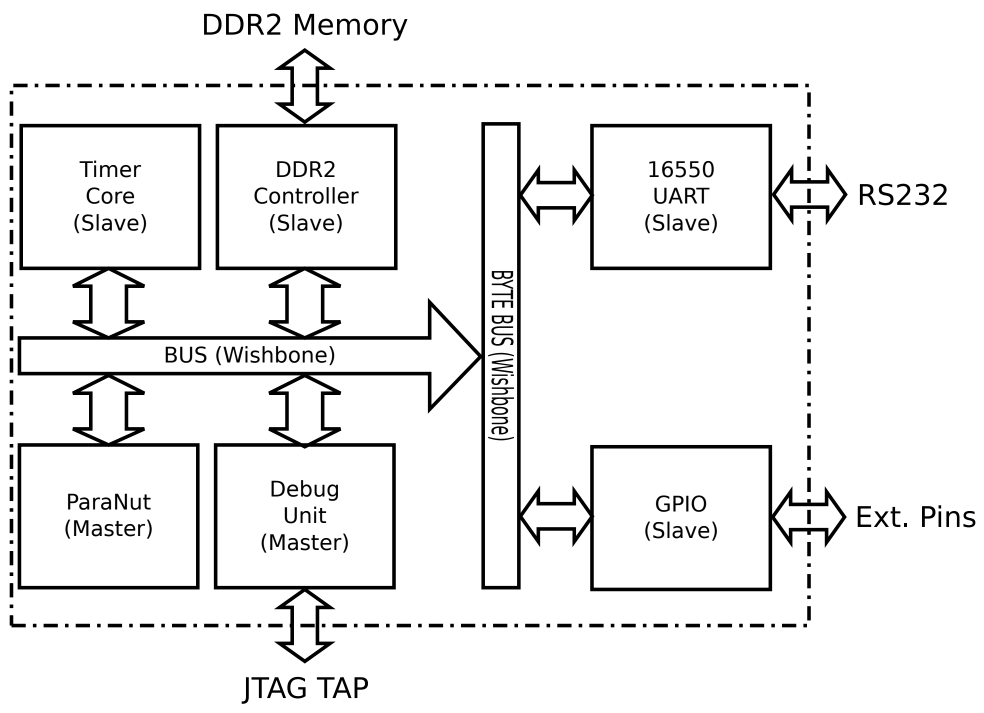

Figure 6.1: Block diagram of the system-on-chip used for evaluating the ParaNut architecture.

The system-on-chip was successfully synthesised for the Xilinx Virtex-5 FPGA that was introduced in section 6.3 using the same set of synthesis tools. The build process was handled by the ORPSoC makefile based build system which was slightly modified for use with VHDL source files. The resulting FPGA bitfile was then put into operation on a Digilent XUPV5 board, also known as ML509, running with a bus frequency of 50 MHz. The DDR2 RAM interface is clocked at 266 MHz. For all benchmarks, the ParaNut was configured as shown in table 6.1 unless otherwise noted. For experiments where multiple CPUs were used, 1, 2, 4, and 8 CPUs were synthesized for a clock speed of 25 MHz. Based on the same SoC, the OR1200 implementation of the OR1K architecture is compared to the ParaNut for the benchmarks. As far as possible, the OR1200 was configured similarly to the ParaNut. For the size of the caches a configuration of 16 KiB for both instruction and data cache was chosen, with a line size of 16 Bytes. The OR1200 has separate instruction and data bus access, but this should not be of consequence for the benchmark results once the program has been loaded into the cache. The implemented cache sizes should be big enough to hold the complete text code segments of the prgrams in the cache.

6.1.2 Compiler Options

All benchmark programs were compiled with GCC version 4.5.1-or32-1.0rc4 of the OpenRISC GNU toolchain utilizing the Newlib C library (linker flags: "-mnewlib"). Optimization level was set to "`-O3`" and compiler flags corresponding to CPU capabilities were set to "`-mhard-mul -msoft-div -msoft-float`".

6.1.3 Dhrystone

The *Dhrystone-2.1* benchmark [18] which can be obtained from the Fresh Open Source Software Archive [http://fossies.org] was used in this work. The number of runs was set to 400,000.

Table 6.1: ParaNut benchmark configurations

Parameter	Value
CPU cores	1..8
Cache size	32KB
Cache sets	512
Cache line size	16 Bytes (4 banks)
Cache associativity	4 ways
Cache replacement strategy	LRU
Instruction buffer size	4
Write buffer size	4
Shift implementation	Barrel shifter
Multiplier pipeline stages	3
MEMU arbitration	7

Based on the evaluation platform that was described in the last section, the results for the Dhrystone benchmark are as shown in table 6.2.

Table 6.2: Dhrystone benchmark results

Processor	Dhrystones per second
ParaNut	25,667
ParaNut (no store buffer)	23,073
OR1200	37,821

For the Dhrystone benchmark, the use of a store buffer increases performance by about 7.5%. However, if a small design with low resource usage is the primary design goal, then the LSU can be omitted to lower resource usage and potentially improve timing. Here, the OR1200, which has a 5-stage pipeline that can execute most instructions in 1 clock cycle [19] is about 47% faster than the ParaNut.

6.1.4 CoreMark

The *CoreMark* benchmark from the Embedded Microprocessor Benchmark Consortium (EEMBC) is targeted at evaluating CPUs designed for embedded devices [20]. It supports operation with multiple cores through the use of pthreads, fork, or sockets by default. These methods are not yet supported by the current VHDL implementation of the ParaNut presented in this work, because of missing synchronisation between CPU cores and the lack of threading libraries and operating system support. Since the CoreMark allows for operation with only using the stack segment as memory location for objects, it is possible to run multiple parallel executions of the same CoreMark program, if every core uses its own separate stack. Therefore, the C runtime initialisation code of the OpenRISC GNU toolchain has been modified so that separate stacks for different CPU cores are initialized. This is done by reading the CPU core identifier from the PNCPUID CPU register and adding a fixed (negative) value to the stack pointer. Of course, this bears the risk that a stack of one CPU will grow large enough to overwrite the stack of another. An offset of 32 KiB was added to the stack of each CPU that should prevent overlapping of stacks for the programs tested. With the modifications in effect, it was possible to run multiple “copies“ of the CoreMark in parallel at the same time. Although this does not allow the CoreMark to run in parallel, the results can be used to show the correct operation and the performance of the MEMU with multiple CPU cores. The results for runs with multiple

CPUs were calculated by adding all of the single execution times for each CPU und dividing them through the number of CPUs. The measurements were performed with 1 to 8 cores for the ParaNut, which is the maximum number of cores that could fit on the FPGA. Cache size was set to 64 KiB. Since the 8 core ParaNut had to be synthesised for a bus clock frequency of 25 MHz to avoid timing errors, all other processors were synthesized for the same clock frequency in order to get comparable results. The number of runs was set to 500 so that a correct operation was validated by the CoreMark program.

Table 6.3: CoreMark benchmark results

Processor	**CPU cores**	**Iterations/sec per core**	**Core-Mark/MHz per core**
ParaNut	1	19.99	0.80
	2	19.98	0.80
	4	19.62	0.78
	8	19.12	0.76
ParaNut (no store buffer)	1	18.48	0.74
	2	18.47	0.74
	4	18.23	0.73
	8	18.00	0.72
OR1200	1	32.08	1.28

The results show no significant negative performance impact for the ParaNut when running with up to 8 CPU cores in parallel. The use of a write buffer increases performance by about 3.2%. While the OR1200 is about 60% faster than the ParaNut single core, it does not support multiple CPU cores.

6.1.5 Merge Sort

A simple merge sort program has been created that can use the parallel capabilities of the ParaNut and use multiple cores to achieve a single objective. The algorithm is based on an idea by John von Neumann [21] which follows the principle of divide and conquer. Figure 6.2 shows how the algorithm was adapted to work with the ParaNut VHDL implementation presented in this work.

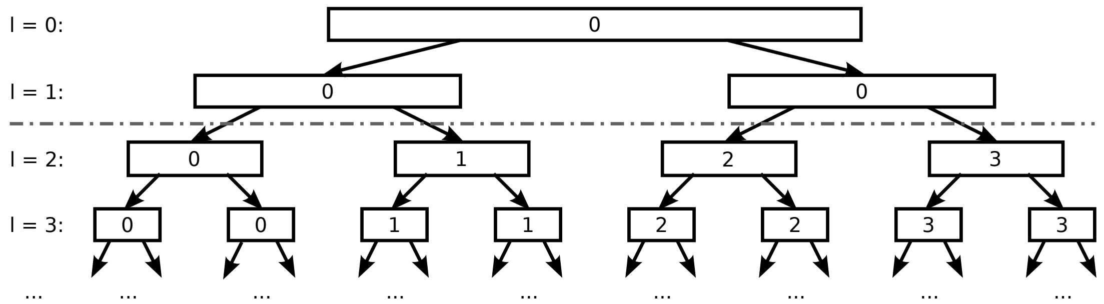

Figure 6.2: Basic principle of the merge sort implementation for ParaNut.

In the figure, an exemplary number of 4 CPUs is used to sort the array. Based on a recursive algorithm the source array is divided into four equal slices that are then processed by each of the 4 CPU cores. The number in each slice is the number of the CPU that is going to process the slice. In the division phase, every core has to traverse the tree recursively and look for his own slice to sort. For every level of recursion l the size of the remaining slice is halfed.

When a CPU reaches level $l = \log_2(n)$, where n is the number of CPUs (i.e. $n = 4 \Rightarrow l = 2$), only the CPU with the same ID as the slice will continue traversing the remaining sub-tree and finally sort and merge it. This is indicated by the dotted line in the figure. Only when all CPUs have finished sorting and merging their sub-tree, the controlling CPU, i.e. the CPU with ID 0, sorts and merges the remaining slices from $l = \log_2(n) - 1$ up to $l = 0$. A simple synchronisation mechanism is used for reporting to the controlling CPU. Since only CPU0 will sort the remaining slices, a linear speedup is not to be expected. The source code for the program can be seen in listing C.1. Table 6.4 shows the results for a randomly generated array of 524,288 32-bit integers (2 MiB). The array will therefore not completely fit into a cache of 32 KiB size so that an increased number of cache misses is to be expected. The times were again measured for a bus clock frequency of 25 MHz. Cache hit rates were measured using the histogram feature of the ParaNut VHDL implementation (see section 6.2.5).

Table 6.4: Merge sort benchmark results first run

Processor	**CPU cores**	**Elapsed time/sec**	**Speedup**	**Data read hit rate**	**Data write hit rate**
ParaNut	1	45.886	1	95.00%	94.29%
	2	28.460	1.61	93.84%	94.81%
	4	21.312	2.15	92.26%	93.20%
	8	22.858	2.01	89.02%	91.39%
OR1200	1	31.454	-	-	-

The OR1200 is about 45% faster than the single core ParaNut. The relative speedup for the ParaNut decreases for an increasing number of cores and the absolute speedup for eight cores is even lower than for four. The increasing number of cores seems to produce an increasing number of cache misses as multiple CPUs simultaneously request data from different addresses that are spread evenly across the whole range of the array. The small cache size in comparison to the array size seems to limit the maximum speedup here. An additional run for the ParaNut with a cache size of 256 KiB was made. Additionally, the array size was reduced to 128 KiB. The results are shown in table 6.5.

Table 6.5: Merge sort benchmark results second run

Processor	**CPU cores**	**Elapsed time/sec**	**Speedup**	**Data read hit rate**	**Data write hit rate**
ParaNut	1	2.270	1	99.98%	99.98%
	2	1.200	1.89	99.93%	99.96%
	4	0.764	2.97	99.84%	99.94%
	8	0.640	3.54	99.26%	99.89%

Hit rates obviously benefit from the increased cache size and reduced array size. However, the speedup does not seem perfect yet, and cache misses are still present. The more CPU cores, the more accesses to addresses with the same index address but a different tag address may happen at the same time. A 4-way set associative cache is not able to hold more than 4 entries for the same index address which may be a bottleneck if more than 4 CPUs are used. The next section will have a more deeper look into what can cause performance bottlenecks.

6.2 Detailed Performance Analysis

The section consists of a series of experiments measuring performance for different events inside of the ParaNut architecture using the histogram feature of the ParaNut VHDL implementation. The same evaluation platform and benchmark programs as in the previous section are used. The ParaNut was configured with a cache size of 16 KiB, 4 ways, and 16 bytes cache line size. Other parameters are given in the relative sections. For all measurements, if more than 1 CPU is involved, average values have been calculated as the arithmetic mean across all CPUs.

6.2.1 Instruction Throughput

Instruction execution times have been measured for different classes of instructions. Table 6.6 shows the results for 500 runs through CoreMark with 1 CPU core and an LSU with 4 entry store buffer and without store buffer. Execution times are measured in clock cycles from the point when an instruction is issued (beginning in the execute stage) to its last clock cycle in the execution stage when the next instruction is about to be issued in the next clock cycle. The category for “other” instructions includes instructions such as NOPs and also branches that are not taken. Instruction fetch delays are measured beginning from a memory read request of the IFU to memory until the instruction is about to arrive in the next clock cycle. Because read ports are pipelined they acknowledge in the clock cycle before data actually arrives.

Table 6.6: Dhrystone instruction execution times in number of clock cycles

Event	Min	Avg	Max	Count	% of total time
ALU instructions	2	2.00	2	158,400,927	47.63
Shift instructions	2	2.00	2	47,200,398	14.19
Multiply instructions	5	5.00	5	400,002	0.30
Load instructions	4	4.00	13	26,800,030	16.12
Store instructions	3	3.04	9	22,800,028	10.40
Jump/Branch instructions	2	2.00	2	30,800,184	9.26
Other instructions	1	1.00	1	14,000,122	2.11
All instructions	1	2.21	13	300,401,691	100.00
Instruction fetch	1	2.04	163	300,401,691	-

As can be seen from the measurements, except for load and store instructions, execution times of instructions are constant. They are also in line with the times proposed in the EXU architecture description in section 4.4.1. Store instructions require a minimum of 3 clock cycles when using a LSU with a store buffer. Load instructions complete in 4 clock cycles at least. The total average CPI across all instructions is 2.21. The minimum instruction fetch delay is 1 clock cycle and occurs on incidental hits in the bus interface when the cache is cold and the bus interface is reading cache lines from main memory to the cache. However, the number of bus interface hits was 97 compared to the total number of 300,401,691 instruction fetches so their ratio is virtually zero. When the cache is hot, the average instruction fetch delay of 2.04 clock cycles reflects very well the minimum time for a cache hit of 2 clock cycles. The average CPI for all instructions lies slightly above the average number of clock cycles for instruction fetches which indicates that the IFU can sustain a constant instruction stream for the EXU.

The same measurements have been repeated for the CoreMark and merge sort programs with 1 to 8 CPU cores, in order to examine performance for load and store operations with an increased workload for the MEMU. Only delays for load operations, store operations, all operations, and instruction fetches are shown, because the minimum, average, and maximum number of clock cycles for all other categories remain constant. The results are shown in table 6.7 and 6.8 for both programs. The cache for the merge sort program is 256 KiB and array size is 128 KiB.

Table 6.7: CoreMark instruction execution times in number of clock cycles

Number of cores	**Min**	**Avg**	**Max**	**Count**	**% of total time**
Load instructions					
1	4	4.00	43	31,929,065	24.84
2	3	4.00	54	63,858,130	24.84
4	3	4.07	198	127,716,260	25.17
8	3	5.04	1983	255,432,520	30.29
Store instructions					
1	3	3.00	43	8,498,133	4.96
2	3	3.00	43	16,996,266	4.96
4	3	3.01	259	33,992,532	4.95
8	3	3.56	2050	67,985,064	5.57
All instructions					
1	1	2.28	43	225,355,337	100.00
2	1	2.28	54	450,710,674	100.00
4	1	2.29	259	901,420,862	100.00
8	1	2.50	2050	1,802,839,826	100.00
Instruction fetch					
1	1	2.04	72	225,355,337	-
2	1	2.04	71	450,710,674	-
4	1	2.08	256	901,420,862	-
8	1	2.26	1733	1,802,839,826	-

Table 6.8: Merge sort instruction execution times in number of clock cycles

Number of cores	Min	Avg	Max	Count	% of total time
Load instructions					
1	3	4.00	155	2,088,491	18.08
2	3	4.01	162	2,088,788	18.09
4	3	4.06	575	2,088,713	18.11
8	3	4.15	877	2,089,665	17.16
Store instructions					
1	3	3.00	113	1,638,392	10.64
2	3	3.03	208	1,638,425	10.73
4	3	3.29	602	1,638,611	11.50
8	3	5.42	1181	1,639,463	17.59
All instructions					
1	1	2.19	155	21,095,682	100.00
2	1	2.19	208	21,097,188	100.00
4	1	2.22	602	21,096,831	100.00
8	1	2.39	1881	21,103,442	100.00
Instruction fetch					
1	1	2.04	130	21,095,682	-
2	1	2.05	201	21,097,188	-
4	1	2.06	403	21,096,831	-
8	1	2.10	907	21,103,442	-

In overall, the average instruction throughput seems to remain constant across all tests with a little more delay for 8 cores. However, the maximum delays show an increase of up to 500% when comparing 8 to 1 core CPUs. This is to be expected because shared resources like the bus interface will be demanded more frequently and a port may have to wait many bus cycles before it is finally granted the bus interface. What can also be seen is that for eight cores the average delay for store operations increases by 81% compared to the one core configuration, which reflects the lower priority for write ports. The relative amount of clock cycles spent for store instructions compared to all instructions therefore rises by 7%. The average load and instruction fetch delays from one to eight CPU cores only increase by 3.75% and 2.94%, respectively.

6.2.2 LSU Load/Store Performance

In order to further examine the influence of a store buffer on the performance of load and store instructions, their execution times were compared for different store buffer sizes. Table 6.9 shows the average execution times in number of clock cycles for 500 runs through CoreMark varying the number of CPU cores. Table 6.10 shows the average execution times for loads and stores for the merge sort program.

As can be seen, a store buffer generally improves performance not only for the minimum but also for the average delay of store operations. However, the biggest improvement can be seen when comparing the 0 and 4 entry buffers. Delays for buffer sizes greater than 4 do not to improve that much. Surprisingly, latency for load instructions seems to be slightly increased. However, load times are not as badly influenced as is the benefit for stores. The results for the

Table 6.9: Average load/store execution times for CoreMark in number of clock cycles

Instruction Class	**Store buffer entries**	**1 core**	**2 cores**	**4 cores**	**8 cores**
LOAD	0	4.00	4.00	4.15	5.92
	4	4.00	4.00	4.15	6.45
	8	4.00	4.00	4.18	5.56
	16	4.00	4.00	4.18	5.56
STORE	0	5.72	5.77	6.39	10.14
	4	3.00	3.00	3.17	4.41
	8	3.00	3.00	3.12	3.56
	16	3.00	3.00	3.07	3.48

Table 6.10: Average load/store execution times for merge sort in number of clock cycles

Instruction Class	**Store buffer entries**	**1 core**	**2 cores**	**4 cores**	**8 cores**
LOAD	0	4.54	6.26	8.86	19.60
	4	4.56	7.11	13.10	31.78
	8	4.56	7.23	12.85	30.81
	16	4.56	7.21	12.39	30.61
STORE	0	7.11	12.83	24.29	53.46
	4	3.01	4.62	10.34	32.18
	8	3.00	4.40	8.98	29.49
	16	3.00	4.05	7.83	28.82

merge sort program show that delays for stores without a write buffer are increasing much more with the number of CPU cores when more cache data read and write misses are to be expected (32 KiB cache vs 2 MiB array size). During loads, store buffer partial hits that must forward bytes from the store buffer but still access the MEMU to fetch the remaining bytes accounted for a maximum of 0.8% of all loads seen in all benchmarks. The fraction for total store buffer hits that could forward a data word immediately did not exceed 1.1% for all read operations. However, with an increasing number of CPU cores, the number of hits in the store buffer was higher, probably because data words stayed longer in the store buffer due to higher demands to the shared resources of the MEMU.

6.2.3 IFU Instruction Fetch Performance

Different sizes of the instruction fetch buffer in the IFU have been examined for influence on average instruction fetch read delays. The times have been measured for 4 to 16 instruction buffer entries with a varying number of CPU cores using the CoreMark program. Instruction fetch delays are measured from the beggining of a read request of the IFU until the read port acknowledges. The results are shown in table 6.11.

The measurements indicate, that there is no significant reduction in average instruction fetch delay for an increased number of instruction buffer entries. The maximum increase is 1.9% for the 16 entry buffer compared to the 4 entry buffer. Therefore, it could be worth investigating an IFU without store buffer in order to save additional slices.

Table 6.11: Average instruction fetch delays for CoreMark in number of clock cycles

Instruction buffer entries	1 core	2 cores	4 cores	8 cores
4	2.04	2.04	2.11	2.24
8	2.01	2.01	2.07	2.24
16	2.01	2.01	2.07	2.21

6.2.4 Bus Interface Performance

The bus interface is responsible for reading and writing cache lines from and to main memory and therefore has a significant influence on cache miss performance. Since access times to the external DDR2 memory are not known, the delay times for reading and writing a cache line to and from memory are determined. Time is measured for the full Wishbone bus cycle during reading or writing a full cache line. Cache line sizes of 16, 32, and 64 bytes show the performance of main memory and bus system for 4-, 8-, and 16-word bursts respectively. They employ an incrementing burst read/write cycle for the Wishbone bus interface which is also supported by the bus interface of the DDR2 memory controller. Table 6.12 shows the results for both reads and writes.

Table 6.12: Bus interface read delay to main memory in number of clock cycles

Burst count (words)	read			write		
	min	avg	max	min	avg	max
4	7	36	93	9	39	93
8	11	41	96	13	44	96
16	19	49	105	21	53	105

As far as the results show, burst access to the DDR2 memory controller seems to be correctly implemented. Reads have a minimum offset of 3 clock cycles and then take 1 extra clock cycle for every word to be read. The offset for writes is 5 clock cycles. The maximum delays seem to be constant across reads and writes, increasing by a constant offset of 3 and 12 clock cycles for 8 and 16 word bursts respectively.

6.2.5 Cache Performance

This section compares cache performance of the MEMU for a varying number of CPU cores. Time was measured from the beginning of a read or write request to a port until it is acknowleged. Cache read hit and miss delays for instruction and data ports as well as cache write hit and miss delays for data ports are examined using the CoreMark and merge sort programs from before. Here, cache size was 32 KiB and the size of the array for the merge sort program was set to 2 MiB. Tables 6.13 through 6.15 show the delays in number of clock cycles varying the number of CPU cores. For benchmark runs where multiple CPUs were involved, the minimum and maximum delays are the best and worst case results across all CPUs. *Note:* Hit and miss rates for reads may not amount to a total of 100%. The difference is the percentage for incidental bus interface hits.

For cache read hits, the minimum and average delay times do not increase much with the number of CPU cores. Only for write hits, the average access time is increasing more strongly than for

Table 6.13: Cache delay for instruction reads in number of clock cycles

CPU cores	Hit rate	min	avg	max	Miss rate	min	avg	max
CoreMark								
1	99.99%	2	2.05	5	<0.01%	7	13.28	75
2	99.99%	2	2.05	8	<0.01%	8	13.86	75
4	99.91%	2	2.10	14	0.06%	3	27.15	533
8	99.85%	2	2.30	23	0.09%	3	97.73	2,064
Merge sort								
1	99.99%	2	2.07	8	<0.01%	9	27.67	234
2	99.99%	2	2.09	10	<0.01%	3	45.21	559
4	99.99%	2	2.16	18	<0.01%	4	249.62	3,122
8	99.99%	2	2.23	20	<0.01%	3	555.21	6,179

Table 6.14: Cache delay for data reads in number of clock cycles

CPU cores	Hit rate	min	avg	max	Miss rate	min	avg	max
CoreMark								
1	99.99%	2	2.00	5	<0.01%	43	43.00	43
2	99.99%	2	2.00	5	<0.01%	15	39.75	15
4	99.54%	2	2.07	12	0.42%	3	30.91	729
8	97.11%	2	2.33	21	2.17%	3	132.10	2,151
Merge sort								
1	94.66%	2	2.01	5	5.33%	8	18.07	197
2	93.66%	2	2.02	7	6.12%	3	80.01	548
4	92.02%	2	2.10	11	7.18%	4	173.06	6,479
8	88.44%	2	2.19	14	9.12%	3	393.93	14,956

Table 6.15: Cache delay for data writes in number of clock cycles

CPU cores	Hit rate	min	avg	max	Miss rate	min	avg	max
CoreMark								
1	99.99%	3	3.42	9	<0.01%	26	47.72	103
2	99.99%	3	3.49	46	<0.01%	24	44.75	104
4	99.71%	3	4.29	513	0.29%	24	75.72	1,012
8	98.73%	3	10.06	1,013	1.27%	24	213.75	3,129
Merge sort								
1	97.01%	3	3.55	143	2.99%	26	42.67	311
2	94.84%	3	7.24	395	5.16%	24	159.10	952
4	93.00%	3	13.81	698	7.00%	24	325.73	8,231
8	91.75%	3	28.60	1,780	9.25%	24	547.98	10,129

read hits. This demonstrates again the influence of different priorities of read and write ports. The overall minimum access time for cache read hits is two clock cycles and 3 for write hits which is the theoretical minimum delay. However, the average and maximum miss delays significantly increase with the number of CPU cores. This is for both instruction reads as well as data reads and writes. This becomes clear when looking at the delay times for data cache misses for the merge sort program. Since the size of the array to be sorted is much greater than the cache size, cache miss rates are increasing significantly with an increase of the number of CPU cores. Additionally, even with only one CPU, miss rates are high to begin with. Since program code size is relatively small, miss rates for instruction fetches are not significant for both CoreMark and merge sort. The overall minimum read miss delay can be as low as 3 clock cycles because of read ports watching out for incidental bus interface hits. The delay is much higher for write ports that cannot benefit from this.

6.3 ParaNut Resource Usage and Timing Results

This section is going to show resource usage and timing results for different configurations of the ParaNut processor. Table 6.16 shows the configuration that was used to generate area and timing reports, unless otherwise noted. Parameters that have a range argument (e.g. 1..16) in the table will be varied in the different experiments and further be specified in the respective subsections.

Table 6.16: ParaNut resource usage and timing results configurations

Parameter	**Value**
CPU cores	1..16
Cache size	4..32 KiB
Cache sets	64..1024
Cache line size	16..64 Bytes (4..16 banks)
Cache associativity	1..4 ways
Cache replacement strategy	LRU
Instruction buffer size	4
Write buffer size	4
Shift implementation	Barrel shifter
Multiplier pipeline stages	3
MEMU arbitration	7

The numbers that are extracted from the synthesis reports for documenting resource usage are:

- Number of occupied slices
- Number of slice flip flops
- Number of 4 input Look-Up-Tables
- Number of block RAM cells

From the timing reports the following items were extracted:

- Minimum period
- Minimum input arrival time before clock
- Maximum output required time after clock
- Maximum combinational path delay

Block RAM usage is documented only for the experiments in section 6.3.4, where resource usage depending on cache size is examined.

6.3.1 Modules Relative Resource Usage

First, the overall distribution of slices relative to the modules of the ParaNut is examined. Figure 6.3 shows the slice usage of all the main modules of the ParaNut VHDL implementation in comparison. The numbers in parentheses show the total ratio of slices that is used on the FPGA.

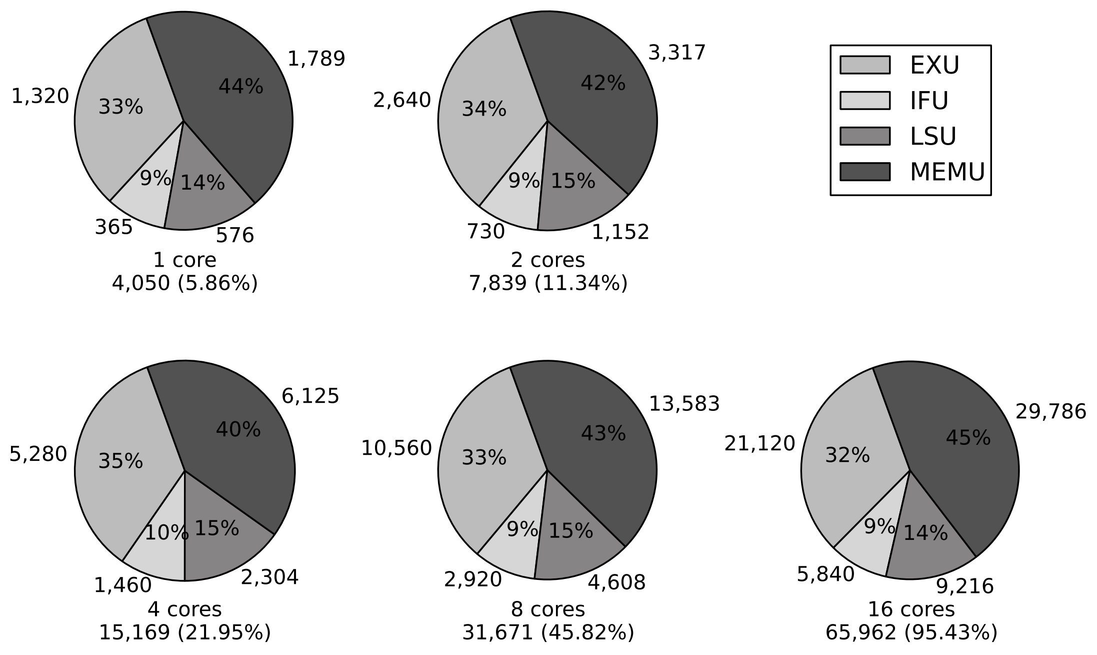

Figure 6.3: Percentaged slice usage for main ParaNut components.

While the proportions do not seem to change with the number of cores, the MEMU takes most of the total area into account with about 43%. Next in line is the EXU with about 33% of the total area. LSU and IFU share the last of the remaining 24% of area where they constitute 15% and 9% respectively. In order to save area, components like the LSU could be omitted from the design. The influence of the store buffer of the LSU on performance has been examined in section 6.2.2. An implementation without IFU could also reduce area consumption and has yet to be implemented.

Most of the area is consumed by the MEMU. Figure 6.4 shows the resource usage of components of the MEMU in comparison. In overall, the proportions do not seem to change for the MEMU as well. The spike for slice usage of the interconnect in the diagram for 2 cores is not in line with the rest of the diagrams. This may be due to different outcomes during optimisation of the synthesised netlist by the synthesis tool. For every CPU core, the number of read ports and write ports is doubled. This suggests that the number of slices is doubled as well, and slice usage for the other components seem to behave equally. The interconnect includes resources that are used for routing and multiplexing addresses and data between all of the components, including the crossbar switch that is used to route any of the cache banks to read ports, write ports, and the bus interface. The arbiter is a mealy based combinatorial circuit that has to convey grants to every read and write port and the bus interface. Hence, the number of read and write ports seems to have direct influence on area consumption of the MEMU. In the future, the implementation of CePUs with a capability of 1 that do not have a read port for fetching instructions could help in reducing slice usage.

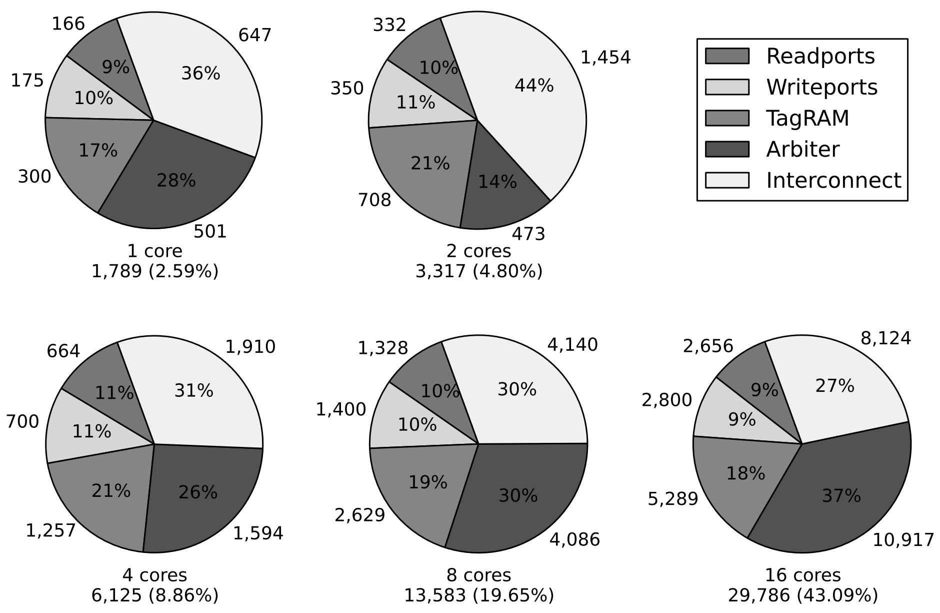

Figure 6.4: Percentaged slice usage for MEMU components.

6.3.2 Area and Timing For Different Numbers of CPUs

Figures 6.5 and 6.6 show area and timing results for different configurations of the ParaNut VHDL implementation varying the number of CPU cores. The parameters are as follows:

- CPU cores: 1, 2, 4, 8, 16
- Cache size: 16 KiB
- Cache sets: 256
- Cache line size: 16 Bytes
- Cache associativity: 4-way

As can be seen in the figures, slice usage appears to scale linearly with the number of CPU cores. The average increase in slices for a doubling of CPU cores is 97%. This is in line with the tendency that can be seen from the previous measurements. However, if CoPUs are implemented that support a lower capability, a lower increase in slice usage is to be expected.

The minimum period increases by an average of 40% with the number of CPU cores. The critical path that designates the minimum period is identified ranging from the MEMU tag RAM hit logic to a read port which then requests a bank from the arbiter based on a tag RAM hit. A write port can concurrently request a bank from the arbiter and if it does not interfere with the read port bank request, the write port additionally requests a linelock from the arbiter. Based on the outcome of the returned grant signal, the next contents of the write port state machine registers are determined. As the number of CPU cores grows, so does the number of read and write ports, and with it the the number of gates that has to be passed for the combinational logic that determines grant signals. Since the write port requests another grant signal based on the outcome of a previous requested grant signal, the path leads 2 times through the combinational

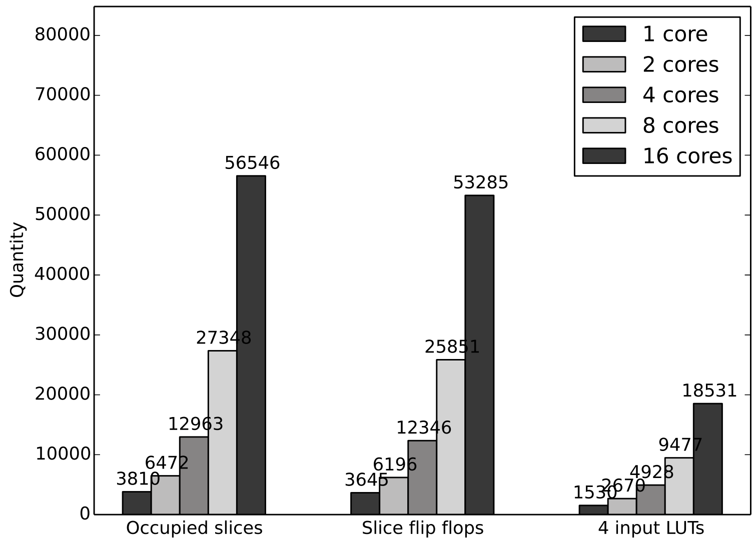

Figure 6.5: Slice usage for different number of CPU cores.

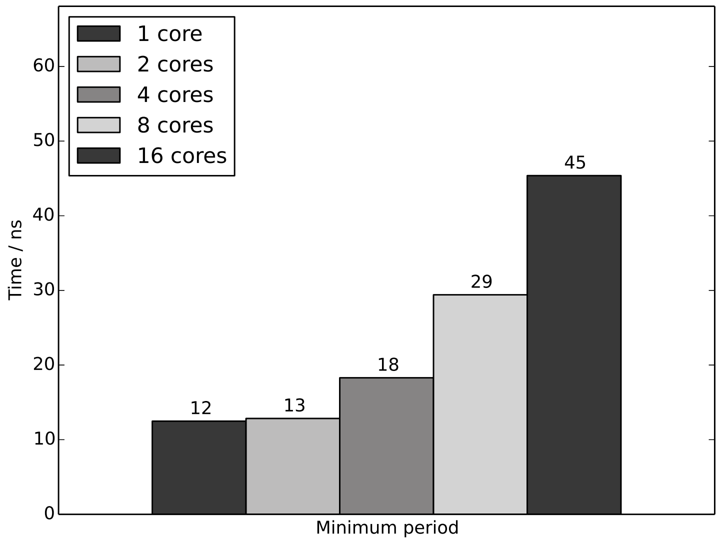

Figure 6.6: Minimum period for different number of CPU cores.

logic of the arbiter in one clock cycle. In order to improve timing, breaking the chain at the point where the first grant is given looks promising in terms of improved timing, and even could help in reducing slice usage. Another approach would be to implement the arbiter as a moore type state machine, but this would probably increase the minimum delay for cache read hits from 2 to 3 clock cycles. To catch up on the lost clock cycle, requests to tag and bank RAM could be issued simultaneously by read ports, so that in the case of a cache hit, data is immediately available in the next clock cycle. However, this would have to be carefully planned, as this would have consequences for all parts of the MEMU which is a carefully balanced structure.

The minimum input required time is 2.204 ns for all configurations depending on a path that ranges from the acknowledge signal of the Wishbone bus to a register in the bus interface which signals to the read ports that valid input data from the bus arrived. It can be used by the read ports to read data that is not yet in the cache (bus interface hit).

The maximum output required time is 2.694 ns for a path originating from the bus interface that multiplexes output data to the Wishbone bus based on its current state register. Last, there is no combinational path for the ParaNut processor.

6.3.3 Area and Timing for Different Cache Associativity

For the results shown in figures 6.7 and 6.8, the number of cache ways has been varied. In order to keep cache size constant, the number of cache sets is halved with a doubling of cache associativity. The results are shown for 1 to 16 CPUs. Only slice usage and the minimum period are shown. The parameters are:

- CPU cores: 1, 2, 4, 8, 16
- Cache size: 16 KiB
- Cache sets: 1024, 512, 256
- Cache line size: 16 Bytes
- Cache associativity: 1-, 2-, 4-way

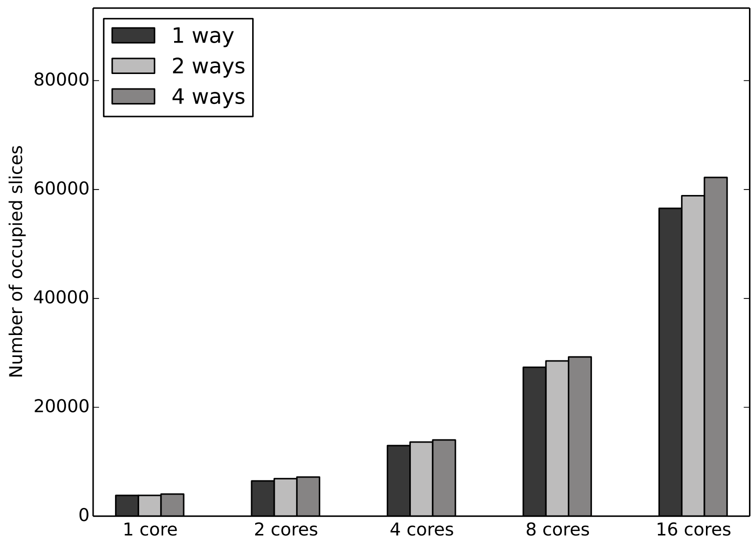

Figure 6.7: Slice usage for different cache associativity.

The results show an average increase of 4.17% in slice usage for increased associativity while there seems to be no significant impact on the minimum period.

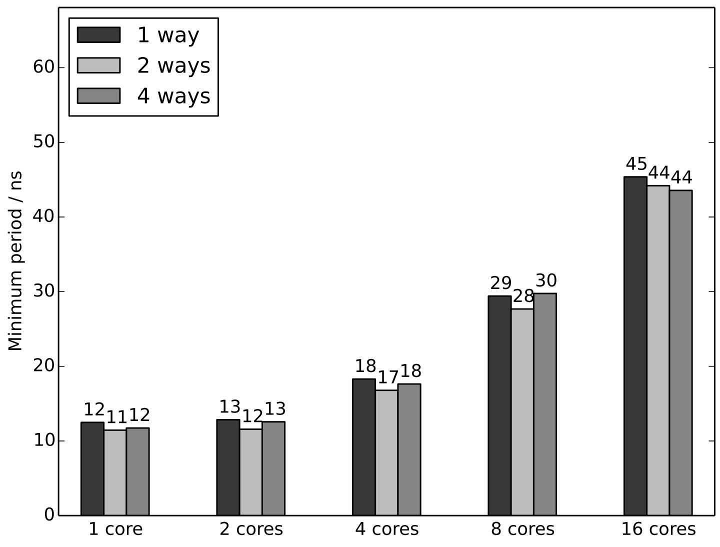

Figure 6.8: Minimum period for different cache associativites.

6.3.4 Area and Timing for Different Cache Sizes

Figures 6.9 and 6.10 show results for different cache size configurations by varying the number of cache sets. The measurements have been made for 1 to 16 CPU cores. Additionally, figure 6.11 shows block RAM usage and its distribution on tag RAM and bank RAM. The parameters are:

- CPU cores: 1, 2, 4, 8, 16
- Cache size: 8, 16, 32, 64, 128, 256 KiB
- Cache sets: 64, 128, 256, 512, 1024, 2048
- Cache line size: 16 Bytes
- Cache associativity: 4-way

As can be seen in the figures, increasing cache size does not show any significant impact on slice usage. The highest average increase for the minimum period can be seen for the 2 CPU configuration with 3.82% and the lowest increase is for the 16 CPU configuration with 0.50%.

Block RAM usage is dependent on the number of cache sets, cache ways, cache banks, and number of CPU cores. Doubling the number of cache sets effectively doubles the amount of memory (and therefore the number of addresses) needed for both tags and banks. For the caches from 4 KiB to 16 KiB, no additional block RAM cells are needed for both tag and bank RAM for a given number of CPU cores, since the effectively needed memory is smaller than can fit into a single block RAM cell. A single block RAM cell for the Virtex-5 FPGA can hold up to 4.5 KiB of memory. A single bank of a 16 KiB cache that has 4 banks is 4 KiB in size, so up to this point only one block RAM cell per bank will be used. The same applies to the number of cache ways, but for the tag RAM, instead of the number of addresses, the number of data bits to be stored is doubled. If implemented, there is also a small overhead for LRU information that needs to be stored along with the tag address as well as valid and dirty bits.

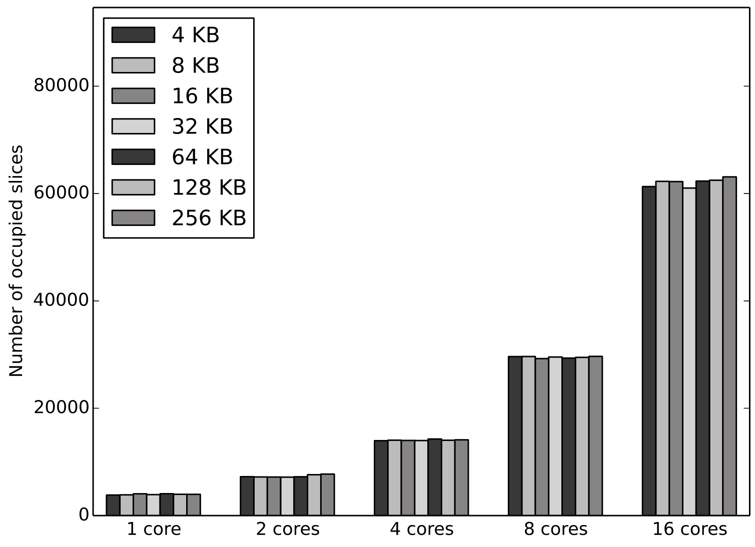

Figure 6.9: Slice usage for different cache sizes.

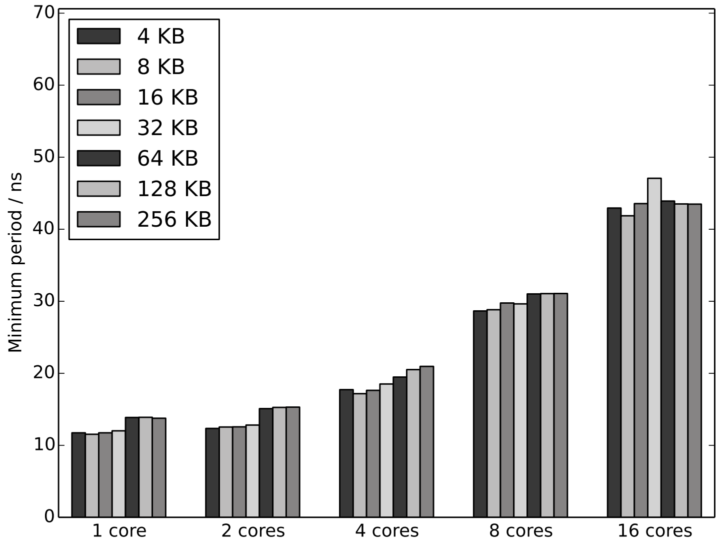

Figure 6.10: Minimum period for different cache sizes.

The number of cache banks does not affect tag RAM size, but only bank RAM size. It is the other way round for the number of CPU cores, which doubles the amount of tag RAM needed, since every CPU core has its own port to the tag RAM.

(*Note: The number of bits needed for storing the tag address is actually reduced by one bit for every doubling of the number of cache sets as well as cache banks, but this will rarely influence the amount of block RAM needed.*)

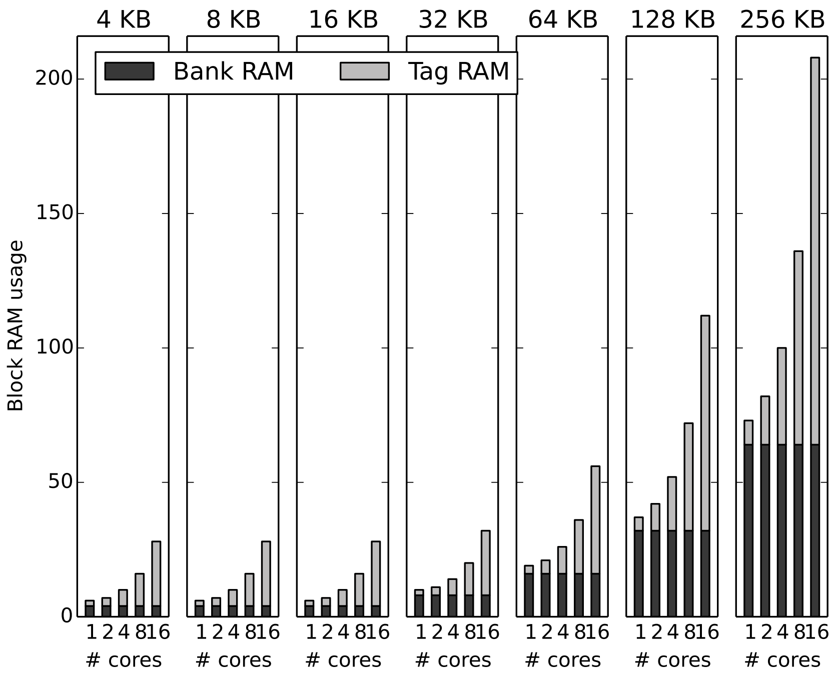

Figure 6.11: Block RAM usage for different cache sizes.

6.3.5 Area and Timing For Different Number of Cache Banks

The impact of varying cache line size by altering the number of cache banks is shown in figures 6.12 and 6.13. For a doubling of the number of banks, the number of cache lines is halved to keep cache size constant. All parameters are shown for 1 to 16 CPU cores. The parameters are:

- CPU cores: 1, 2, 4, 8, 16
- Cache size: 16 KiB
- Cache sets: 256, 128, 64
- Cache line size: 16, 32, 64 Bytes
- Cache associativity: 4-way

Increasing cache line size shows an average increase of 28% in slice usage. The minimum period seems to increase more with a higher number of CPU cores. The relative increase per doubling of banks is 3% for 2 cores and 7.7% for 16 cores. By increasing the number of banks, more logic is needed for the crossbar switch of the MEMU that routes read and write ports as well as the bus interface to the bank RAM ports. The critical path is the same as for the path described in section 6.3.2. However, as can be seen from this example, the length of the chain of logic gates is also increased by the number of cache bank RAM ports. Here, the same suggestions as in the mentioned section apply.

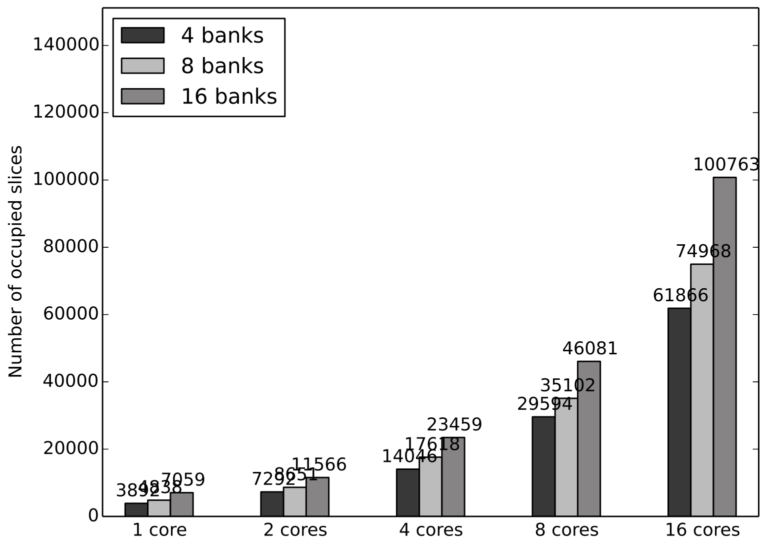

Figure 6.12: Slice usage for different number of cache banks.

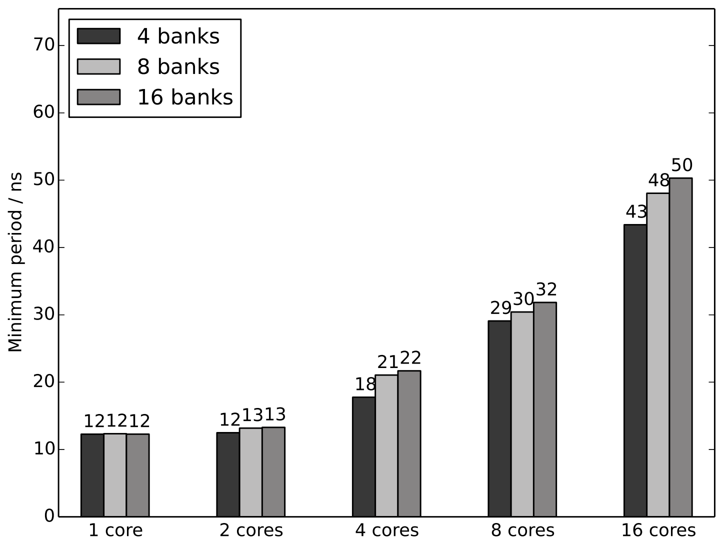

Figure 6.13: Minimum period for different number of cache banks.

6.3.6 EXU Resource Usage and Timing Results

The EXU has three different implementations for the shift logic. An implementation without hardware multiplier, which is not mandatory by the OR1K architecture, is also compared. All EXUs with hardware multiplier had a 3 stage multiplier pipeline. Resource usage and timing results are shown in figures 6.14 and 6.15.

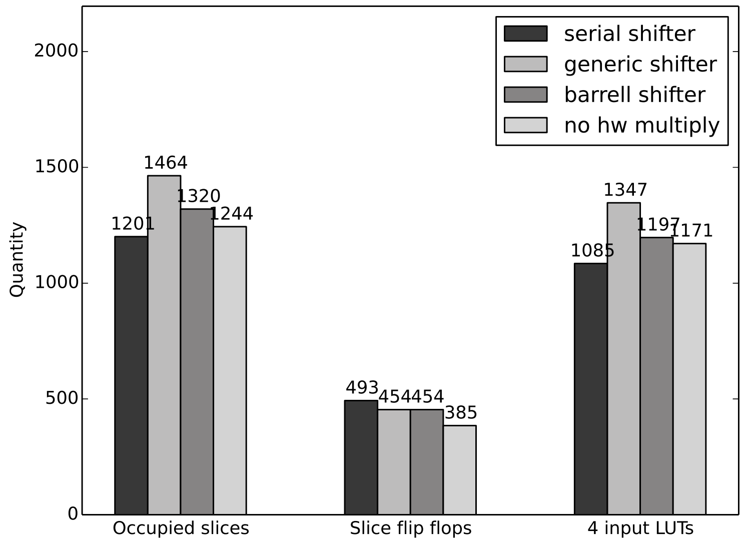

Figure 6.14: Slices for Different Configurations of the EXU.

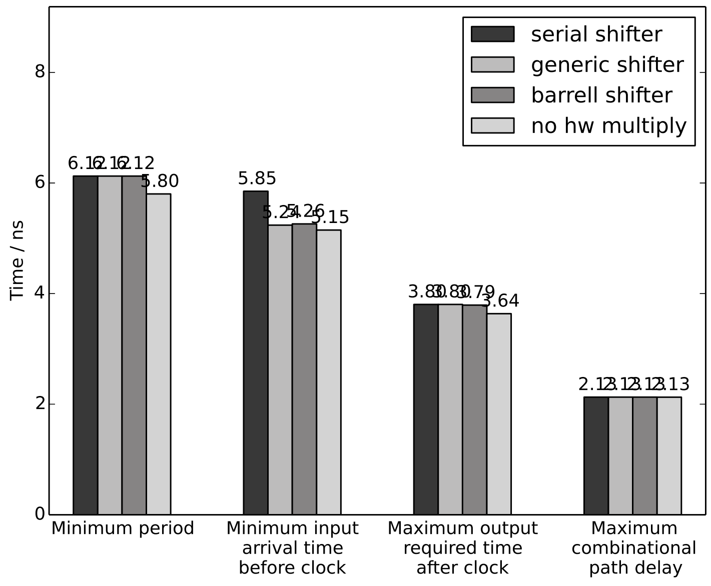

Figure 6.15: Timing results for Different Configurations of the EXU.

The EXU with serial shift implementation uses a little more registers than the the other two. For the number of occupied slices, the generic shifter which is inferred by the synthesis tool has the worst value. In all cases, the minimum period allows for a clock frequency of 163.40 MHz. Maximum output required time as well as combinational path delays are all equal. The minimum input arrival time is slightly increased for the serial shifter. The generic shift implementation seems not to be a good choice, because it does not provide faster timing and has higher resource usage than the barrel shifter. About 6% of slices can be saved with an implementation without a hardware multiplier. The hardware multiplier uses 3 additional DSP48 slices. Implementations of CePUs with a capability of 2 could save area by omitting logic for interrupt and exception handling as well as privileged instructions. A CePU with a capability of 1 would additionally lose the portion that is used for decoding instructions.

Critical paths are the same for all implementations with a hardware multiplier. Here, the minimum period is limited by the multiplier pipeline registers. In case of the 1-stage implementation it is 7.537 ns (132.679 MHz). For the EXU without a hardware multiplier, the path leads from the decode-stage registers through the ALU to the zero test comparator that then sets the input to the flag bit of the supervision register (SR). The minimum input arrival time is constrained by a path in the instruction decode stage, where the multiplier input operands are determined by decoding the instruction register from the IFU, reading the register file operands, and finally storing them in the first multiplier pipeline register. Maximum output required time and combinational path delay are both determined by a path signaling to the IFU that the next instruction can be fetched. In the first case it originates from the instruction decode stage pipeline registers. For the combinational path delay the debug unit stall signal that stalls the execution of all instructions is the source.

6.3.7 IFU Resource Usage and Timing Results

This section examines resource usage and timing results for different sizes of the instruction fetch buffer internal to the IFU. Figures 6.16 and 6.17 show the results.

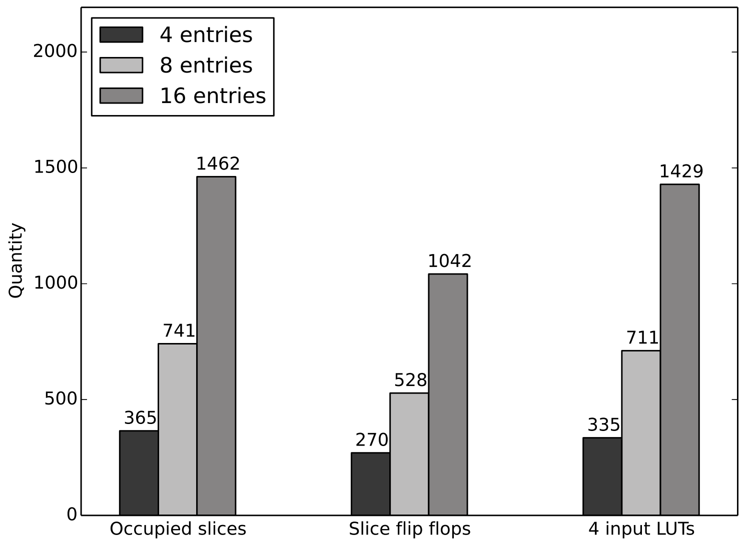

Figure 6.16: Slice usage for different IFU instruction buffer sizes.

As can be seen in the figures, resource usage scales linearly with a doubling of buffer size. All timing values are affected by an average increase of 24%. They can increase an otherwise non-critical path for the ParaNut that originates from the tag RAM hit logic of the MEMU. Based on a cache hit, the IFU read port sends a request to read a bank to the arbiter. If the request in granted, the read port acknowledges to the IFU which then calculates the next instruction fetch address and places it in a register. As the results in section 6.2.3 showed an instruction buffer

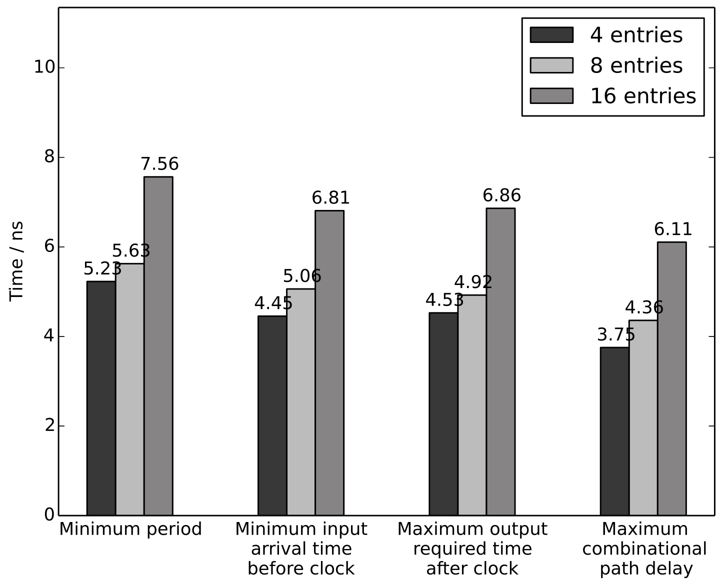

Figure 6.17: Timing results for different IFU instruction buffer sizes.

does not necessarily increase the performance of instruction fetches. However, an implementation without instruction buffer has yet to be realised and evaluated.

6.3.8 LSU Resource Usage and Timing Results

This section examines the influence of the LSU store buffer size on resource usage and timing results for the LSU. Figures 6.18 and 6.19 show the results for a store buffer with 0 to 16 entries.

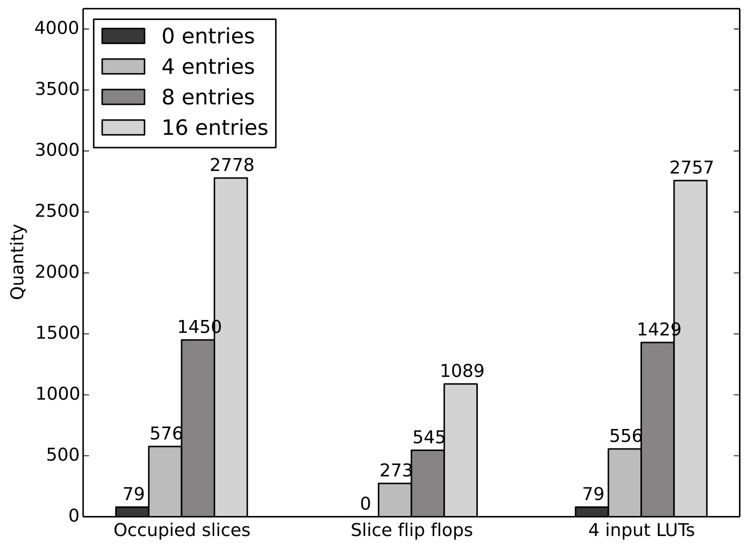

Figure 6.18: Slice usage for different sizes of the LSU store buffer.

Doubling the number of store buffer entries increases slice usage by an average of 121%. The LSU without store buffer does not use any registers and only a few LUTs that are mainly needed

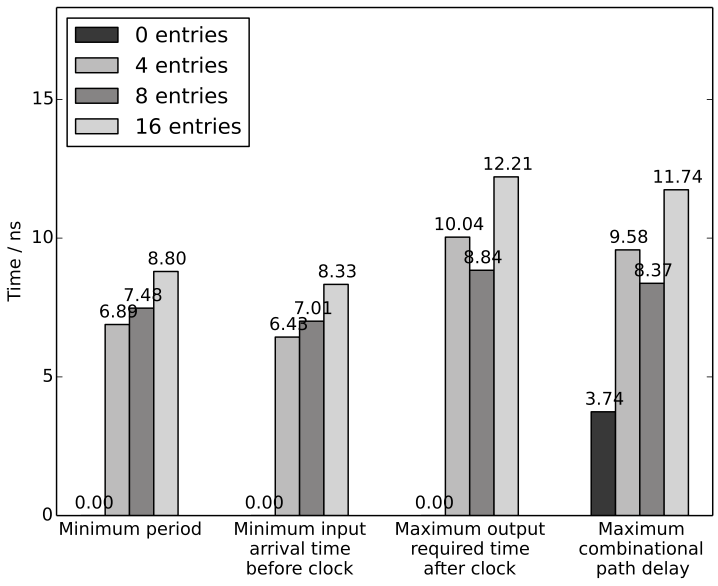

Figure 6.19: Timing results for different sizes of the LSU store buffer.

for formatting and sign-extending data words. Timing values are increased by an average of 13.5% per doubling of store buffer entries. The critical path is specified as originating from an address in the store buffer registers leading through a comparison with the hit logic for reads and writes which finally determines the contents of the write buffer registers for the next clock cycle. The use of a store buffer and its impact on performance of load and store operations was examined in section 6.2.2. The results showed, that a store buffer generally increased performance for store operations but did not yield significant improvements for greater buffer sizes.

7 Conclusion

In the context of this work, a functional VHDL implementation of a ParaNut processor was implemented. It was then successfully embedded into a system-on-chip that enabled execution of binary programs on the processor on FPGA hardware. A set of benchmark programs was then used to validate the functionality and evaluate the performance of the implemented ParaNut processor. In a second step, the implemented processor was then supplied with performance profiling hardware. This allowed a detailed analysis of the performance of core components of the ParaNut architecture at critical points in the design. Instruction throughput, delays for load and store operations as well as cache hits and misses were analysed in the context of different workloads and configurations. Furthermore, the influence of different instruction and store buffer sizes on overall performance was examined. The evaluation could be conducted with an implementation of up to 8 CPU cores.

The next step was to evaluate the ParaNut in terms of resource usage and timing for the FPGA that was used in the functional and performance evaluation. This was done for different configurations of the ParaNut and allowed to identify what the components are that use the most resources. Moreover, critical paths for the system were identified. Based on the results, suggestions were made for reducing area consumption and improving timing.

Finally, the ParaNut processor that was implemented in the context of this work provides a starting point for further development on the ParaNut architecture,

8 Next Steps

This chapter gives an outlook on the possible future enhancements to the ParaNut VHDL implementation and the ParaNut architecture in general.

8.1 Improving Timing and Minimizing Resource Usage

Chapter 6 showed, where the critical paths for different modules of the ParaNut are and what are the modules with the highest resource usage. Based on this information, improvements to the implementation can be made. Implementing CoPUs with reduced capability can also help in improving both goals. The results also show that FPGA resource usage is increasing linearly with the number of CePUs. CoPUs with a capability of 2 could save area that is released by omitting exception processing units and privileged instructions in the execution unit. A CoPU with capability 1 could considerably reduce resource usage by omitting isntruction fetch unit and its read port interface. Since the number of read and write ports also influences the timing of the MEMU, the maximum frequency for the ParaNut VHDL implementation could be increased as well.

8.2 Implementation of CoPUs and Modes

CoPUs have not been implemented yet for both SystemC and VHDL model. CoPUs with different capabilities are an essential building block of the ParaNut architecture and offer great potential in helping to reduce area consumption and improve timing. This is beneficial for implementations that need to fit on FPGAs with very limited resources. Moreover, in order to properly support the different ParaNut CPU modes, synchronisation mechanisms must be implemented. This is especially important for the linked and threaded modes. Since the unique SIMD concept of the ParaNut architecture is the main motivation for its design, this would be a very important asset to the ParaNut VHDL implementation.

8.3 Verification of Exception Handling

Only a subset of exceptions are implemented in the ParaNut SystemC and VHDL models. Handling of exceptions has only been tested in simulation for system calls, traps, alignment errors, and illegal operations exceptions. Their correct operation in hardware remains to be verified.

8.4 Additional Units

The OR1k architecture specifies a set optional units that provide standard features for modern CPUs. Many of them have not yet been implemented but would widen the field of application for the ParaNut processor.

A port of the Linux kernel exists for the OR1K architecture. In order to be able to run Linux on the ParaNut, a memory management unit (MMU) would be necessary. The OR1k architecture specifies that translation lookaside buffer (TLB) misses can be handled by logic in hardware or cause an exception so that the TLB reload is handled in software. The latter approach would be preferred from an economical hardware resource usage point of view. This would merely require implementing the TLB and the logic required for comparing tag information and checking for violation against protection information. Page faults also generate MMU exceptions. MMU exceptions require the effective address of the operation that caused the exception to be saved into the EEAR register. Since the ParaNut has a store buffer and the CPU may already have completed the instruction before the address is actually written to memory, MMU exceptions would require the processor to retrieve not only the address, but also the program counter and state of the supervisor register for the store instruction that caused the exception.

The current implementation of the ParaNut processor allows for uploading programs to main memory through a JTAG based debug adapter and the GNU debugger (GDB) from the OpenRISC toolchain. While this is sufficient for testing the hardware, software developers need to be able to debug software running on the target platform. To this end, support for the optional debug unit would have to be implemented to enable basic debugging through software breakpoints.

Timing measurements for the benchmarks have been conducted using an external timer core that can only be accessed over the bus. While this is sufficient for measuring the execution times of benchmarks that run several seconds, it may be desirable for real time applications to have a high precision tick timer unit on a per-CPU basis. Scheduling of tasks for an operating system would also require an independent clock source for every CPU that can additionally generate interrupts. Implementing a programmable interrupt controller would be mandatory. Support from the software side is already given through the OpenRISC support library that is included in the OpenRISC newlib toolchain.

8.5 Supporting Additional Platforms

The ParaNut has been successfully run embedded on a system-on-chip with the ORPSoC as a peripheral platform. Support for other platforms would further enhance the range of application of the ParaNut. Actually, the GRLIB from Aeroflex Gaisler was considered as an alternative platform for the ParaNut to be evaluated on for this work. The GRLIB features a considerable amount of IP cores that are based on the AMBA on-chip bus. This would require implementing a bus interface for the ParaNut that is AMBA compliant. Alternatively, a Wishbone to AMBA bridge can be used that is available from the OpenCores project. Moreover, the GRLIB build system natively supports a broad range of simulation and synthesis tools from different FPGA manufacturers which can help in bringing the ParaNut to new platforms.

A ORPSoC + ParaNut HOWTO

This chapter is intended to guide the reader through the process of setting up a working environment for the ParaNut. The digital media provided with this work includes pre-built FPGA bitfiles and software. In case the reader is interested in the underlying environment for e.g. customising their own SoC or adding support for a new board, continue reading section A.4. In any case all readers should first read section A.2 in order to determine whether and what additional steps are required to get the ParaNut VHDL implementation working on an FPGA.

A.1 Development Tools

This section gives a complete overview of the tools involved in generating bitfiles, running the ParaNut on hardware, and executing programs on the evaluation platform.

A.1.1 ORPSoC

ORPSoC is the "OpenRISC Reference Platform System On Chip". It provides a peripheral system for OpenRISC compatible processors and a framework for simulating as well as synthesizing systems for FPGAs. At the time of this writing, there are 2 versions of the project, namely "ORPSoCv2" and "ORPSoCv3", which has just been released shortly before the completion of this work. Although ORPSoCv2 is no longer actively developed and superseded by ORPSoCv3, this HOWTO is based on the former version and may provide a starting point for porting to the latter. Instructions for downloading, installing, and operating both versions can be found on the OpenCores website at [http://opencores.org/or1k/ORPSoC] . Some modifications have been made to the platform which are already included in the ORPSoC version found on the digital media:

- Added support for the Digilent XUPV5-board (ML509).
- Modification of hardware synthesis build scripts to better cope with VHDL source files
- Addition of the ParaNut and a small timer core with support libraries

A.1.2 OpenRISC GNU tool chain

Compiling C/C++/Assembler programs as well as downloading the resulting binaries to the ParaNut SoC running on the FPGA board can be done with the OpenRISC GNU tool chain. Throughout this work, the bare"=metal version based on the Newlib library was used. The tools are commonly prefixed "`or32-elf-`". A comprehensive source of information can be found on the OpenCores website at [http://opencores.org/or1k/OpenRISC_GNU_tool_chain] where instructions can be found on how to obtain pre-built binaries as well as compiling from source.

After having installed the toolchain, there is one thing to do when adding support for a new board: Edit one of the existing board support library files in newlib-1.x.0/libgloss/or32, e.g. ml501.S. This file defines a few symbols that help Newlib to set up the C runtime environment for compiled programs. This is what the file looks like after adapting it to the design that is used on the ML509 board in this work:

```
/*
 * Define symbols to be used during startup - file is linked at compile time
 *
 */
.global _board_mem_base
.global _board_mem_size
.global _board_clk_freq

_board_mem_base:        .long   0x0
_board_mem_size:        .long   0x10000000 /* 256MB */

_board_clk_freq:        .long   66666666

/* Peripheral information - Set base to 0 if not present*/
.global _board_uart_base
.global _board_uart_baud
.global _board_uart_IRQ

_board_uart_base:       .long   0x90000000
_board_uart_baud:       .long   115200
_board_uart_IRQ:        .long   2
```

Here, only _board_mem_size was changed to 256MB, which is the size of the ML509's DDR2 memory module. It may not be larger than the actual memory size because it is used to set up the stack pointer which is automatically set to the top of the RAM (_board_mem_size - _board_mem_base) during C runtime initialisation.
_board_clk_freq actually refers to the clock speed at which the Wishbone BUS/CPU is running, not the used clock input to the FPGA. Because the UART module's clock input is connected to the Wishbone BUS' clock and its internal baudrate generator needs to be set up to the desired baud rate (here: 115200) it is important that this variable is set correctly for getting sane output via UART. The UART's base address is kept unchanged. Save this file to something with the name of the board name in it, e.g. ml509.S.

The next step will be to create an object file that can then be linked into the executable of the program:

```
$ or32-elf-as -o ml509.o ml509.S
$ or32-elf-ar -q libboard.a ml509.o
```

The resulting archive is then placed in a new folder under install_path/or32-elf/lib/boards/*boardname*. Every time the compiler/linker is invoked with -mnewlib -mboard=ml509 linker flags, the symbols in that archive will be used.

A.1.3 or_debug_proxy

The missing link between GDB and the USB debug adapter that will later physically connect to the target is "or_debug_proxy" It is intented for use with the OpenRISC USB"=JTAG Debugger only. The project can be checked out from the OpenCores SVN repository:

```
$ svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/or_debug_proxy
```

A README file is included that lists all required dependencies for compilation and steps on how to build and install.

A.2 Prerequisites

In order to be able to run software on the ParaNut processor some requirements must be met. These are mainly the hardware and non-free software which cannot be distributed with the digital media provided with this work. Be sure that the following requirements are met when trying out the ParaNut. On every item, an explanation is given and hints to possible solutions are provided in case the requirements are not met:

A.2.1 XUPV5-LX110T FPGA (ML509) board

Of course the ParaNut should be able to run on other FPGAs/boards, too, but this will require some additional work. Some of the necessary steps to do that are outlined in section A.4 but they cannot cover the full process because it is very specific to the actual vendor/board used.

A.2.2 Linux operating system/Cygwin Windows and libraries

The tools used to create FPGA bitfiles and software for both the ParaNut as well as debugging require some sort of Linux operating system and libraries. While the Xilinx ISE tools are available natively for Windows, at least some virtual machine running Linux or Cygwin are necessary in order to run the GDB stub `or_debug_proxy` and the OpenRISC NewLib toolchain programs. For a complete documentation please see the README file in the `or_debug_proxy` and `ToolchainOr32Bare` directories on the digital media.

A.2.3 OpenRISC GNU toolchain

The OpenRISC GNU toolchain that is used to compile programs for the ParaNut must be obtained from the OpenCores website at [http://opencores.org/or1k/OpenRISC_GNU_tool_-chain] . There are different versions of the toolchain, so be sure to choose the right version for your operating system. Follow the instructions on the site to install the toolchain. After having installed the toolchain, copy the file `paranut/sw/newlib/crt0.o` from the CD to the `/opt/or32-elf/lib/` folder. (*Note*: `"/opt/` may be a different folder for your installation). This is the object file for the C runtime initialisation code that is getting linked to a program when using the Newlib library. It is ensures that different memory locations for the stack of each CPU will be used for a multi-core ParaNut. A single-core ParaNut will also work without the modified C runtime.

A.2.4 OpenRISC USB-JTAG Debugger

Downloading software to the ParaNut SoC running on the FPGA board is done via a JTAG debug interface. Presently, the only hardware JTAG debug adapter working with the debug application or_debug_proxy is the OpenRISC USB"=JTAG Debugger[22], which is unlikely to change. The newer version of the ORPSoCv3 uses the “Advanced Debug System” which works with a broader range of JTAG debug adapters but this would first require porting the ParaNut ORPSoCv2 project to ORPSoCv3. Moreover, a suitable cable with fly-leads to connect from the adapter to the expansion header pins on the FPGA board is required. The debugger also contains a USB-serial-converter which can be used to get printouts from the UART.

A.2.5 Xilinx ISE tools with full license

Since the ML509 board features a Xilinx Virtex-5 XC5VLX110T FPGA, which is not part of the “Webpack” license, a 30-day evaluation or full license is required for the Xilinx ISE tools. However, the ML501 board, which is also supported by ORPSoCv2, does not require a full license and it should be very easy to port to.

A.3 Getting Started

If all of the requirements from the previous section are fulfilled there are just a few steps left before running the ParaNut on the board.

A.3.1 Connecting the hardware

JTAG Debuger Interface

The OpenRISC USB-JTAG Debugger is connected to headers J4 and J7 on the ML509 board as listed in table A.1.

Table A.1: JTAG connection to the ML509 board

JTAG pin	ML509 board pin
3. JTAG TDO	HDR2_6 (header J4)
9. JTAG TDI	HDR2_8 (header J4)
5. JTAG TMS	HDR2_10 (header J4)
1. JTAG TCK	HDR2_12 (header J4)
4. VCCIO JTAG	Any pin of left column of J4
2. GND	Any pin of header J7

All pins of the left column of header J4 are tied to ground. Connect one of them to the GND pin on the USB debugger. All pins of header J7, which is to the right of J4, are connected to 2.5V DC. Connect the VCCIO pin of the debugger to any of the pins on J7.

UART

A serial terminal can be connected either by using the UART embedded in the OpenRISC USB-JTAG Debugger, or by using a serial cable. For the USB debugger, connect the fly-leads from the debugger according to table A.2

Table A.2: UART connection to the ML509 board

RS232 pin	ML509 board pin
7. UART RX	HDR2_2 (header J4)
8. UART TX	HDR2_4 (header J4)

When using a serial cable, simply connect it to the serial port connector P3.

The host PC must use a serial terminal application in order to receive data from the UART. The settings for the serial connection are:

- Baudrate: 115200
- Parity: none
- Bits: 8
- Stopbits: 1
- Flow control: none

Figure A.1 shows a picture of the set up hardware platform.

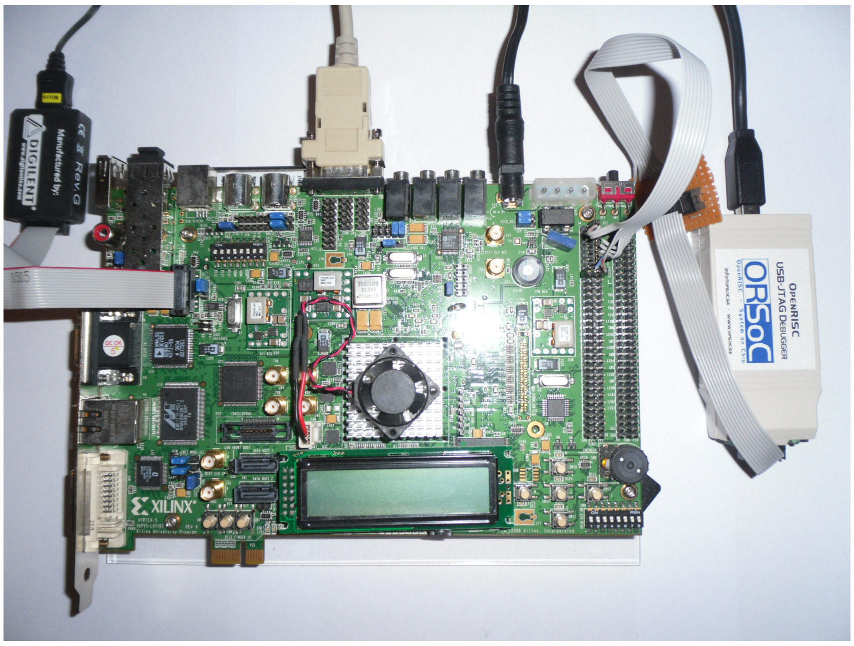

Figure A.1: The ML509 board with the JTAG debugger and serial cable connected.

A.3.2 Configuring The Hardware

This section assumes that the tools and software provided with the CD are used. First, copy the contents of the CD to a local folder on the hard drive e.g.:

```
$ cp -r /cdroot/orpsocv2 ~/destination-folder
$ cd ~/destination-folder
```

It is assumed that as of now all subsequent commands are relative to the destination folder.

The `paranut/bitfiles` directory contains pre-built bitfiles for the ML509 board. The directory includes 4 different bitfiles with ParaNut processors that have 1 to 8 cores and are clocked at 25 MHz. A bitfile can be loaded onto the FPGA using the Xilinx Impact tool.

A.3.3 Downloading Programs

Now that the FPGA is configured, download a program to the ParaNut. This task is accomplished by GDB and involves two pieces of software. For the following steps, it is assumed that the OpenRISC GNU toolchain is installed as described in section A.2. Add the paths for the Newlib toolchain and the `or_debug_proxy` application to your `PATH` environment variable:

```
$ export PATH="${PATH}:/opt/or32-elf/bin"
$ export PATH="${PATH}:/cdroot/or_debug_proxy"
```

Or, if the changes should be permanent across logins, simply add the two lines above to your `~/.bashrc` and start a new terminal.

First, we need to provide a GDB stub that allows GDB to connect to the hardware. It is advised to open up a new terminal for the following command as the current terminal will be occupied after executing:

```
$ or_debug_proxy -r 5000 -b
```

If there are no errors, connect to the board via GDB from another terminal and download a program:

```
$ or32-elf-gdb -ex "target remote :5000" -ex "load" paranut/sw/bin/hello_newlib
```

Be sure to start a serial port terminal to be able to get "printf" output. The serial port uses a baud rate of 115200, 8 bits, 1 stopbit, no parity and no flow control. A press on the reset button on the board should get the processor to start executing the program. However, it is advised that the bifile is simply re-download to the FPGA as this will also purge the cache (handling of cache flushing is not yet implemented in the hardware and may lead to unexpected results). The folder `paranut/sw/bin/` contains further precompiled binaries that are ready to execute on the ParaNut.

A.4 Customising the SoC

If the reader wants to try out a different configuration of the ParaNut other than the provided ones, do so by creating a new board build in the `boards` path. This is achieved by simply copying over one of the existing board builds to a new directory:

```
$ cp -r orpsocv2/boards/xilinx/ml509_paranut orpsocv2/boards/xilinx/
    ml509_paranut_custom
```

Inside of each board build folder is a local overlay of RTL modules located in `board-build/rtl`. Modules which are placed in it take precedence over the same modules also found in the global RTL module path `orpsocv2/rtl`. In the new custom build path edit the file `rtl/vhdl/include/paranut_config.vhd`. It defines a number of constants that control certain aspects of the ParaNut SoC. Debug options do not affect sythesis and can be completely ignored. Where options are not self-explanatory, comments are used to explain their function. For a full explanation on every config item, see section 5.2. If porting to a different board special care should be taken about the following two constants:

CFG_NUT_MEM_SIZE
This has to be set to the size of the RAM. E.g., this can be the size of the onboard DDR RAM or the size of generated on-chip Block RAM when using the Wishbone RAM module.

CFG_NUT_LITTLE_ENDIAN
This parameter must be set to reflect the endianess of the peripheral system so that correct byte select signals for the Wishbone bus are generated by the CPU. This can normally be left as-is.

All other parameters may be set freely within their designated range but keep in mind that setting values too high may result in a design that does not fit onto the target FPGA or fails to meet timing requirements. E.g., doubling the number of CPU cores may easily double the number of slices used. Cache memory is normally implemented as Block RAM if the given target FPGA technology supports this and by incrementing any of the parameters `CFG_MEMU_CACHE_BANKS_LD`, `CFG_MEMU_CACHE_SETS_LD`, or `CFG_MEMU_CACHE_WAYS_LD` by 1, Block RAM usage is usually doubled, among some additional slice overhead. In the example above, a 16KB 4-way associative cache with a line size of 16 bytes (4 words) will be generated (cache size is $2^{2+8+2+2}$).

It's time to finally make a bitfile for the design. If the design should be built from ground up starting with XST synthesis, first clean the synthesis working directory. In case of errors that do not seem to make much sense during synthesis, also clean that directory:

```
$ make -C boards/xilinx/ml509_paranut_custom/syn/xst/run clean
$ make -C boards/xilinx/ml509_paranut_custom/backend/par/run orpsoc.bit
```

B ParaNut VHDL simulation

A testbench for the ParaNut VHDL implementation has been created in the process of this work. It was used to validate the correct operation of the ParaNut. In order to be able to use the testbench you need to either have the Xilinx ISE tools or GHDL installed. The testbench can execute programs that are compiled using the or32-elf-gcc compiler. However, before the program can be used in the VHDL testbench, a VHDL representation of the memory content of the program has to be created. This is achieved by the ParaNut SystemC testbench. It can be obtained from the website of the Efficient Embedded Systems workgroup of the University of Applied Sciences Augsburg [http://ees.informatik.hs-augsburg.de/paranut/index.html.]

The testbench needs at least SystemC-2.2 installed in order to compile. When compiled, execute the testbench with the program to be simulated:

```
$ make
$ ./paranut_tb -v /path/to/program.elf
```

The "-v" switch will generate the VHDL dump. The file will be named `program_mem_dump.vhd` and created in the current folder.

The ParaNut VHDL testbench is located in the `paranut/rtl/vhdl/tb/paranut` folder. The testbench will pick up the file `prog_mem.vhd` as program to be simulated. Copy the file that was generated by the SystemC testbench to the VHDL testbench folder:

```
$ cd paranut/project/rtl/vhdl/tb/paranut
$ cp ../../../../systemc/program_mem_content.vhd .
```

The testbench can be run with either the Xilinx ISIM or GHDL /vhrefhttp://gna.org/projects/ghdl/ VHDL simulators. However, it should be noted that GHDL may have trouble executing testbenches with programs that are big in size. The GUI target version for GHDL needs the GTK-Wave [http://gtkwave.sourceforge.net/] wave viewer program.

The testbench uses make targets to run the simulation. There are two make targets for ISIM. One is for running the command-line version of ISIM, the other for running the GUI version:

For the command line version type:

```
$ make isim-cl
```

For the GUI version type:

```
$ make isim-gui
```

GHDL has the same kind of make targets.

For the command line version type:

```
$ make ghdl-cl
```

For the GUI version type:

```
$ make ghdl-gui
```

C Merge Sort program listing

Listing C.1: merge_sort.c

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

#include "paranut.h"
#include "paranut_hist.h"
#include "counter.h"

#define CTR_BA 0xf0000000
#define BUS_FREQ_HZ 25000000

// SIZE may be only a power of two!
#define SIZE 0x8000
#define MAX_NUMBER 10000

int a[SIZE];
int tmp[SIZE];
int sorted;
unsigned int start_time, stop_time;

void do_msort(int n0, int n1, int m)
{
    int i0, i1, j;

    i0 = n0;
    i1 = m+1;
    j = n0;
    while (i0 <= m && i1 <= n1) {
        if (a[i0] <= a[i1]) tmp[j++] = a[i0++];
        else                tmp[j++] = a[i1++];
    }
    while (i0 <= m)  tmp[j++] = a[i0++];
    while (i1 <= n1) tmp[j++] = a[i1++];
    for (j = n0; j <= n1; j++) a[j] = tmp[j];
}

void do_divide (int n0, int n1, int c, int l, int cpu_id, int n_cpus)
{
    int m;

    if (n1 > n0) {
        m = (n0 + n1) / 2;

        if (l >= n_cpus) {
            // Begin sorting for each CPU.
            if (c == cpu_id) {
                // Only sort assigned slice.
                do_divide(n0, m, c, 2*l, cpu_id, n_cpus);
                do_divide(m+1, n1, c, 2*l, cpu_id, n_cpus);
                do_msort(n0, n1, m);
                if (cpu_id != 0 && l == n_cpus)
                    // CPU finished sorting its sub-tree.
                    pn_sync_set(cpu_id);
```

```c
            } else {
                return;
            }
        } else {
            do_divide(n0, m, 2*c, 2*l, cpu_id, n_cpus);
            do_divide(m+1, n1, 2*c+1, 2*l, cpu_id, n_cpus);
            if (cpu_id == 0) {
                // CPU 0 has to wait here for each CPU to be finished before
                // sorting the remaining array.
                wait_for_cpu(2*c+1);
                do_msort(n0, n1, m);
            }
        }
    }
}

int real_main(int cpu_id)
{
    int i;

    if (cpu_id == 0) {
        counter_init(BUS_FREQ_HZ);
        sorted = 1;
        printf("\nGenerating random array of size %d...\n", SIZE);
        for (i = 0; i < SIZE; i++) {
            a[i] = rand() % MAX_NUMBER;
            if (i > 1 && a[i-1] > a[i])
                sorted = 0;
        }
        if (sorted)
            printf("Random array generation failed.\n");
        else
            printf("Random array generated.\n");
        printf("Running merge sort with %d CPU(s)...\n", pn_get_ncpus());
        counter_reset(CTR_BA, 0);
        counter_start(CTR_BA, 0);
        counter_set_cnt_div(CTR_BA, 0, 2);
        start_time = counter_get_msecs(CTR_BA, 0);
        pn_sync_set(0);
    }
    // Wait for random array to be generated by CPU 0...
    wait_for_cpu(0);
    if (cpu_id == 0) {
        pn_sync_unset(0);
    }
    pn_hist_enable();

    // Begin sorting...
    do_divide (0, SIZE-1, 0, 1, cpu_id, pn_get_ncpus());

    pn_hist_disable();

    if (cpu_id == 0) {
        stop_time = counter_get_msecs(CTR_BA, 0);
        printf("Finished sorting...");
        // Check if array is actually sorted.
        sorted = 1;
        for (i = 1; i < SIZE; i++) {
            if (a[i-1] > a[i])
                sorted = 0;
        }
        if (sorted)
            printf(" Correct operation verified.\n");
        else
            printf(" Correct operation could not be verified.\n");
        printf("Time elapsed: %d ms\n", stop_time - start_time);
    }

    if (cpu_id == 0) {
        pn_sync_clear();
    }

    // Print histogram sequentially for every CPU
    if (cpu_id != 0) {
        wait_for_cpu(0);
        wait_for_cpu(cpu_id-1);
    }
    pn_stats_collect();
    //pn_stats_print();
    // Last one to print global stats...
    if (cpu_id == pn_get_ncpus()-1) {
        pn_global_stats_collect();
        pn_global_stats_print();
    }
    pn_sync_set(cpu_id);
    // Do not leave before all is done...
    wait_for_cpu(pn_get_ncpus()-1);

    return sorted-1;
}

int main ()
{
    return real_main(pn_get_cpuid());
}
```

List of Abbreviations

CPI Cycles per instruction

CPU Central processing unit

DLP Data level parallelism

FPGA Field-programmable gate array

ILP Instruction level parallelism

IP core Intellectual property core

ISA Instruction set architecture

ORPSoC OpenRISC Reference Platform SoC

RISC Reduced instruction set computer

SIMD Single instruction, multiple data

SoC System-on-Chip

TLP Thread level parallelism

VLSI Very-large-scale integration

VHDL Very-high-speed integrated circuit hardware description language

Bibliography

[1] David A. Patterson John L. Hennessy. Computer Architecture - A Quantitative Approach, 4th edition. Morgan Kaufmann Publishers. ISBN: 978-0123704900, 2007.

[2] Xilinx. Microblaze Processor Reference Guide - Embedded Development Kit - EDK 14.5. URL: URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/mb_ref_guide.pdf, 2013.

[3] Sun Microsystems. OpenSPARC Overview. URL: URL: http://www.oracle.com/technetwork/systems/opensparc/index.html, 2013.

[4] Paul Hartke Thomas Thatcher. OpenSPARC T1 on Xilinx FPGAs – Updates, RAMP Retreat – August 2008, Stanford. URL: http://www.oracle.com/technetwork/systems/opensparc/06-ramp-2008-08-final-1530358.pdf, 2008.

[5] Aeroflex Gaisler. LEON3 Processor. URL: http://www.gaisler.com/index.php/products/processors/leon3, 2006.

[6] Catovic E. Gaisler J. Multi-Core Processor Based on LEON3-FT IP Core. DASIA 2006 - Data Systems in Aerospace, Proceedings of the conference held 22-25 May, 2006 in Berlin, Germany. Edited by L. Ouwehand. ESA SP-630. European Space Agency, 2006. Published on CDROM., p.76.1 URL: http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2006ESASP.630E..76G&link_type=GIF, 2006.

[7] Parallella. Parallella Reference Manual. URL: http://www.parallella.org/wp-content/uploads/2013/01/parallella_gen1_reference.pdf, 2013.

[8] Adapteva. Epiphany Architecture Reference. URL: http://www.adapteva.com/wp-content/uploads/2012/12/epiphany_arch_reference_3.12.12.18.pdf, 2013.

[9] OpenCores. OpenCores Website. URL: http://opencores.org/, 2013.

[10] Julius Baxter. Open Hardware Description License Version 1.0. URL: http://juliusbaxter.net/ohdl/ohdl.txt, 2012.

[11] Julius Baxter. Open Hardware Description License. URL: http://juliusbaxter.net/ohdl, 2012.

[12] OpenCores. OpenRISC 1000 Architecture Manual. URL: http://opencores.org/ocsvn/openrisc/openrisc/trunk/docs/openrisc-arch-1.0-rev0.pdf, 2012.

[13] OpenCores. OpenRISC GNU tool chain. URL: http://opencores.org/or1k/OpenRISC_GNU_tool_chain, 2013.

[14] Gundolf Kiefer Hochschule Augsburg. The *ParaNut* Processor - Architecture Description and Reference Manual. URL: http://ees.informatik.hs-augsburg.de/paranut/index.html, 2012.

[15] Gundolf Kiefer Hochschule Augsburg. Forschungsforum Informatik am 21. Juli 2011. Tag der Forschung - Der ParaNut-Prozessor. URL: `http://ees.informatik.hs-augsburg.de/paranut/index.html`, 2011.

[16] OpenCores. Wishbone B4 - WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. URL: `http://cdn.opencores.org/downloads/wbspec_b4.pdf`, 2010.

[17] Jiri Gaisler. A structured VHDL design method. URL: `www.gaisler.com/doc/vhdl2proc.pdf`.

[18] Reinhold P. Weicker. "DHRYSTONE" Benchmark Program. URL: `http://fossies.org/linux/privat/old/dhrystone-2.1.tar.gz/`, 1988.

[19] OpenCores. OpenRISC 1200 IP Core Specification. URL: `http://opencores.org/ocsvn/openrisc/openrisc/trunk/or1200/doc/openrisc1200_spec.pdf`, 2012.

[20] EEMBC. CoreMark - CPU Core Benchmarking. URL: `http://www.eembc.org/coremark/about.php`, 2009.

[21] Donald E. Knuth. The Art of Computer Programming. Volume 3: Sorting and Searching. Third Edition, section 5.2.4. Addison-Wesley, ISBN 0-201-89685-0, 1997.

[22] ORSoC. USB-JTAG Debugger. URL: `http://www.orsoc.se/?page_id=99`, 2013.