Hochschule Augsburg University of Applied Sciences

Fakultät für Informatik

Masterarbeit

Studiengang
Master Informatik

Alexander Bahle
Matr.-Nr.: 955273

Optimization of the ParaNut softcore processor for FPGA systems

Prüfer: Prof. Dr. Gundolf Kiefer

Abgabedatum: 02.11.2020

Verfasser der Arbeit:
Alexander Bahle
Von-Osten-Straße 18
86199 Augsburg
alex.bahle@web.de

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

For some time a trend towards application specific processors is noticeable in most computing fields. Special architectures to accelerate Artificial Intelligences (AIs) or cryptography in Internet of Things (IoT) systems are common and widely adopted. Nonetheless general purpose processors are used in most of these systems as well because they have mature toolchains and the ability to be easily software-programmable. Offering these advantages while still being able to be catered towards a specific application is the aim of customizable processors. Modern field-programmable gate arrays (FPGAs) provide the necessary flexibility to develop systems with a specialized hardware and customizable softcore processors with low cost even for low volume production.

The *ParaNut* architecture describes a processor architecture that is open source, customizable, and highly scalable using the RISC-V instruction set architecture (ISA). Beyond the functions of other general purpose processor the *ParaNut* architecture offers special implementations for data-level and thread-level parallelism in an effort to gap the bridge between these forms of parallelism.

1.2 Purpose of this work

The goal of this work is the optimization of the *ParaNut* processor SystemC and VHDL model in order to achieve higher clock speeds. A single core *ParaNut* should be able to run at 100 MHz on the current generation of FPGAs used at the University of Applied Sciences Augsburg (UASA). Using systematic optimization strategies, like the implementation of constant paths, should be used to also improve the speed of multi-core systems up to 100 MHz. The secondary target is reducing the FPGA resources usage of the processor in all or some configurations. The timing and resource report of the FPGA toolchain and a well-chosen set of software are used to validate the goals on a theoretical and practical level. The results of selected benchmarks can be compared to previous implementations and further confirm the optimizations.

1.3 Structure of This Work

The following chapter 2 ”Fundamentals” will first introduce the RISC-V instruction set architecture and then describe the internal structure of the *ParaNut* architecture. The chapter 3 ”State of the Art” gives an overview of techniques used in this work to enhance the processors performance and resource usage. After that the chapter 4 ”Status Quo” presents the status quo of the different components of the *ParaNut* implementation at the beginning of this work that is ought to be improved upon. The chapter 5 ”Optimization” then lists and describes the actions and changes applied to the components. It is structured in the same manner as the chapter before to easily reference and compare the improvements.
The chapter 6 ”Evaluation” presents the accumulated results of this work and evaluates them against previously collected findings. Lastly the chapter 7 ”Conclusion” offers the summary of this work.

2 Fundamentals

2.1 The RISC-V Instruction Set Architecture

RISC-V (pronounced ”risk-five”) is a free and open instruction set architecture (ISA) that is being developed since 2010. The RISC-V Foundation, an association of more than 235 organizations from universities to well-known companies, has been responsible for the continuous development of the RISC-V instruction set architecture. The architecture uses a modular structure, which is based on a standard instruction set with 32, 64 or 128 bit instruction and address width. In addition, any number and combination of instruction set extensions can be supported by a RISC-V processor. The detailed descriptions of the ISA and the open-source model allows to develop processors using the common resources whilst the processor itself can use a different licensing model. The table 2.1 shows all RISC-V instruction sets and extensions ratified by the RISC-V Foundation to date [Waterman u. a. 2019].

Name	Description
RV32I	Base integer instruction set with 32 bit address & register size
RV64I	Base integer instruction set with 64 bit address & register size
RV128I	Base integer instruction set with 128 bit address & register size
Zicsr	Instructions that operate on the control and status registers (CSRs)
”M” Extension	Extension for integer multiplication and division
”A” Extension	Extension for atomic instructions
”F” Extension	Extension for single-precision floating point instructions
”D” Extension	Extension for double-precision floating point instructions
”Q” Extension	Extension for quad-precision floating point instructions

Table 2.1: List of all ratified RISC-V instruction sets and extensions

2.2 The ParaNut Processor Architecture

The *ParaNut* architecture is developed by the working group Efficient Embedded Systems headed by Gundolf Kiefer at the University of Applied Sciences Augsburg. The goal of the project is to develop an open source scalable softcore processor for FPGAs. By focusing on parallelization on thread and data level, smaller and less complex cores can be used. Starting with the OpenRISC instruction set for both, a SystemC model and a VHDL implementation, of the architecture a change to the RISC-V instruction set was carried out in 2019 [Kiefer u. a. 2015a]. The architecture was presented at the Embedded World Conference in 2015 [Kiefer u. a. 2015b] and in 2020 [Bahle u. a. 2020].
The structure of a *ParaNut* processor is shown in figure 2.1. The processor consists of a *Central Processing Unit* (CePU) and an arbitrary number of *Co-Processing Units* (CoPU), which are all connected to the central *Memory Unit* (MemU). The MemU contains the cache shared by all CPUs and several measures for synchronization between the processing units.

A CPU consists of the *Execution Unit* (ExU), which contains the registers (*general* and *special*), the *Arithmetic Logic Unit* (ALU) and the control unit for the CPU. Connected to this is the *Instruction Fetch Unit* (IFU), which reads instructions via its own connection to the MemU and stores them in a small configurable buffer. The *Load Store Unit* (LSU), also connected to the ExU, is responsible for read and write data access and features a write buffer.
Another special feature of the *ParaNut* architecture are the operating modes of the CPUs, which are shown in figure 2.2.

Mode 0: The CPU is inactive

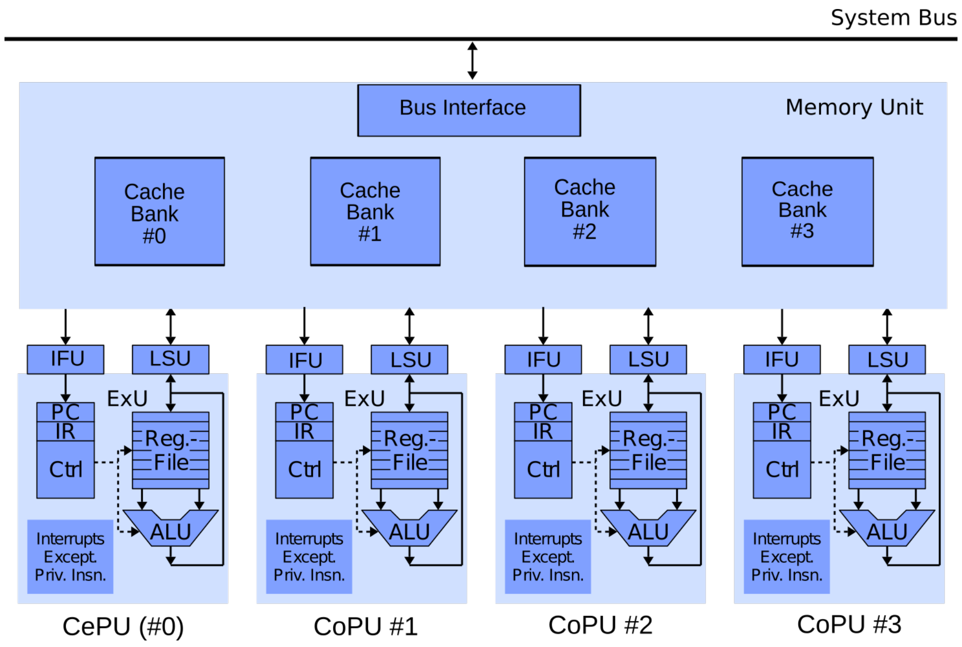

Figure 2.1: Block diagram of a *ParaNut* processor with four cores
source: [Kiefer u. a. 2015a]

Mode 1: The CPU itself does not load instructions via the IFU, but executes the instruction stream of the CePU.

Mode 2: The CPU independently loads commands via the IFU and executes them. Exceptions are handled by the CePU.

Mode 3: Only the CePU runs in this mode. Mode 2 with additional exception handling and management functions

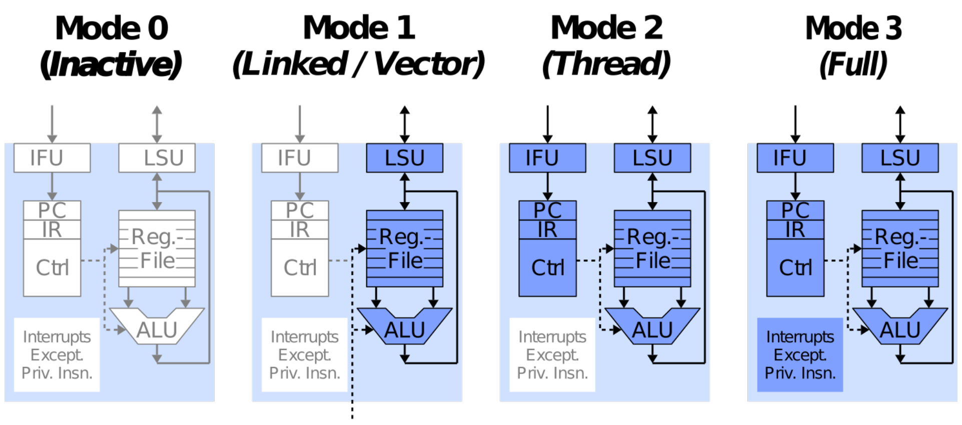

Figure 2.2: Operating modes of a CPU in the *ParaNut* processor
Source: [Kiefer u. a. 2015a]

A deeper look into the inner workings of the MemU with a focus on components that are important for this work is following in chapter 4.

3 State of the Art

This chapter contains the state of the art methods used to optimize the *ParaNut* architecture implementation.

3.1 Timing Closure Techniques

The Timing Closure describes the process by that an FPGA or ASIC design is changed to improve performance and meeting timing objectives. The brunt of the work is often handled by the Electronic Design Automation (EDA) software used (e.g. Xilinx Vivado) that can get beneficial input from the designer to further improve timing behavior.

The possible clock speed in a hardware design can be formulated as seen in formula 3.1. t_{cycle} represents the clock period, $t_{combDelay}$ the combinatorial logics longest path delay, t_{setup} the setup time of the receiving storage element, and t_{skew} the time offset between one storage element and another storage element because of routing delays. [Kahng u. a. 2011]

$$t_{cycle} \geq t_{combDelay} + t_{setup} + t_{skew} \tag{3.1}$$

Modern EDA software for FPGAs contains the functionality to estimate the values in formula 3.1 based on a Static Timing Analysis (STA) that propagates Actual Arrival Times (AATs) and Required Arrival Times (RATs) to every cell, pin or net inside the design to identify timing problems. So called *timing violations* are easily diagnosed and the path or paths can be visualized to localize the parts of the design that caused them. These paths violating the defined timing goals are called *critical paths*. Modifying these is one way to improve the performance of a design. [Kahng u. a. 2011]

In an FPGA or ASIC design the $t_{combDelay}$ includes not only the *logic delay* by the gates or cells themselves but also the *routing delay* between them. For slow designs the *routing delay* may not play a big role but for high performance designs it can be significant. The *place and route* step in the EDA design process will try to optimize this delay to a certain point and try to meet the timing requirements set by the designer. The *routing delays* on an FPGA are also dependent on the amount of logic cells used by the design, because of apparent dependencies when more cells are used and less room for optimizations is available. For optimal timing results Xilinx for example suggested for older generations of FPGAs a 60/40 rule where 60% of the timing budget can be used up in logic and 40% should be saved to be used for routing the design. [Whatcott u. Xilinx Inc. 2008]

If a path is too long because of its $t_{combDelay}$ one can add registers at fitting points in the combinatorial logic and thus cutting the path into two smaller paths. If the cut is exactly in the middle the added delay through the registers may be mitigated by enabling to run the logic at double the speed possible before. Although that might not be feasible in every case because of dependencies to surrounding logic or if additional routing delays are necessary. By cutting the path multiple times a pipeline can be implemented which may improve the throughput while in ideal cases the delay stays the same. [Patterson u. Hennessy 2014]

The easiest way to hit timing closure targets is to define timing targets as early as possible in the design phase of the hardware and design every component to meet the timing target. Using only *Moore machines* as finite-state machine (FSM) ensures that paths will not grow in unexpected ways especially when connected and used in combination with other *Moore machines*. The opposite can be said for using Mealy machines which require great planning and foresight to ensure the timing targets in combination with other FSMs can still be met and thus should be used only sparsely. [Simpson 2015]

3.2 Bus Structures

There are different ways to connect multiple processing elements (PEs) with each other and required resources like memory. Each have unique properties that determine the speed of the system and its scalability.

3.2.1 Crossbar

Originating from early telephony and circuit switching the term crossbar switch describes a network of where every input M can be connected to each output N resulting in a matrix of $M * N$ cross-points. Each cross-point switch connects one of the inputs to one of the outputs. This enables a non-blocking operation where one connection does not prevent another connection on different inputs and outputs in the network. This can lead to a situation where every input and every output can be used concurrently if there are no overlapping connection pairs. [Scudder u. Reynolds 1939] Figure 3.1 is a schematic of a Crossbar with 4 inputs (M) and 4 outputs (N) resulting in 16 cross-points. Every cross-point contains a switch which can be implemented depending on the target system from physical switches to tri-state buffers or FPGA logic. To automate the switching and arbitrate conflicting accesses each of the switches is controlled by a signal generated by a switching logic block that collects the requests of all inputs. The function of the switching block is arbitrary, for example it could be fully blocking and only allow one access per transmission or time slice.

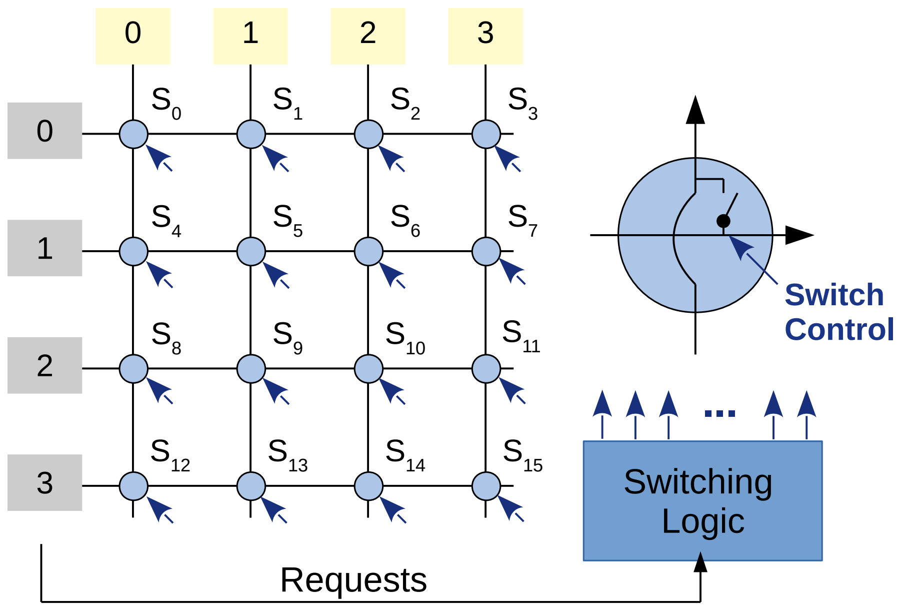

Figure 3.1: Schematic of 4 input (grey) 4 output (yellow) Crossbar

Transferred to a digital bus on a FPGA the crossbar offers an easy implementation and arbitration strategy. Each output has a multiplexer with M inputs and the arbitration can select one of the possible inputs based on given prioritization or other strategies. The drawbacks of using this bus structure include the at least linear growth of required resources for adding a participant. On FPGA systems a crossbar can also quickly lead to long paths because of the amount of signal routes between the inputs and outputs.

3.2.2 Omega network

The Omega network has, in contrast to the Crossbar, a multistage interconnection network. A balanced network with N inputs and N outputs contains $N/2$ switches at each of the log_2N stages. [Lawrie 1975] The switches themselves are as simple as possible and feature 2 inputs and 2 outputs. They can either pass the inputs directly to the outputs or cross them. By connecting them in a *faro* or *perfect shuffle* each input can reach every output setting the switches in each stage to either pass or cross the inputs. [Diaconis u. a. 1983] The necessary switch position to successfully route through the network can be determined by defining a destination address for the input and using either the Destination-tag or XOR-tag routing method described in the following sections.

In a simple implementation where every switch in one stage has the same switching state the Omega networks are highly blocking which means only one input can be routed to one output per time slice. Considering the reduced amount of resources ($(N/2) * log_2N$) a sublinear growth should be possible with this bus structure.

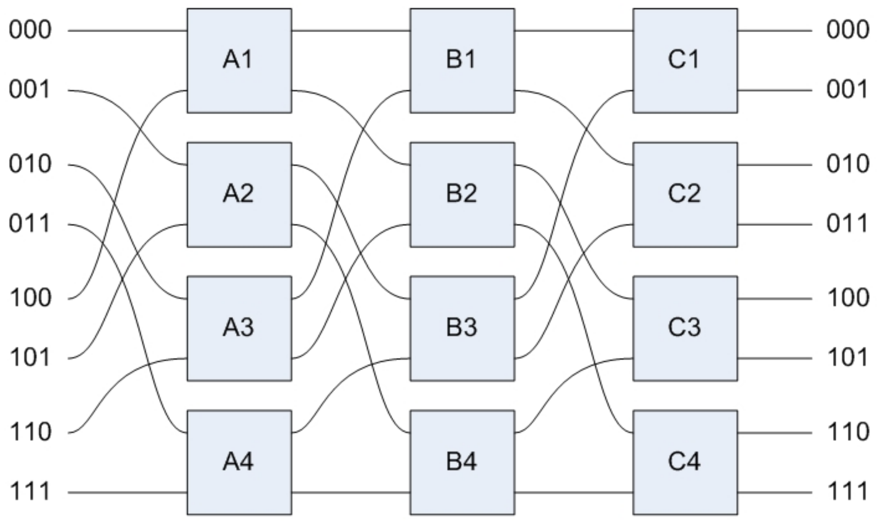

Figure 3.2: A graph of an 8 PE omega network
Source: Wikimedia, "Creative Commons" by Bjmyers17, License: CC BY-SA 3.0

Destination-tag routing

The routing is solely dependent on the destination output and can be defined as a number with log_2N bits. For example in the 8x8 network depicted in figure 3.2 to get to output 5, in binary 101, at the A stage the lower output has to be selected, in the B stage the upper output and in the C stage the lower output again regardless of the input port the routing starts from. Therefore the output address can be used for routing by taking the most significant bit for the first stage, the next bit for the second stage and so on to select the output (0 meaning top output and 1 meaning bottom output). However this method requires that each switch knows which input requires routing and thus is more suitable for message based communication than for a hardware bus.

XOR-tag routing

For the XOR-tag routing the input address and output address are XORed. The resulting tag contains a specific bit combination coding the specific switch setting for this communication pair. Looking at the example network in figure 3.2 input 3 (011 binary) to output 6 (110 binary) would yield the tag $011 \oplus 110 = 101$. As with Destination-tag routing the most significant bit is used in stage A to either pass through the inputs (0) or cross the inputs (1). Thus the XOR-tag 101 would result in crossing stage A, passing through in stage B and lastly crossing stage C to reach the selected output 6 from input 3. Assuming that the access to the outputs is fully blocking and each switch in a stage is switching the same the hardware can be simpler and is more suitable for the hardware bus communication.

4 Status Quo

In this chapter the status quo of the *ParaNut* processor before this work is described. The focus is on parts that are changed and optimized in the following chapter 5. In the first section the Memory Unit (MemU) and its many sub modules like the Arbiter and inner workings are described, showing the bottlenecks and things that could be improved. The next section covers the Execution Unit (EXU) and the opportunities with the multiplication function and the linked mode synchronization. The Load Store Unit (LSU) is covered in the last section, showing a critical path originating from this module.

Before this work the system clock frequency for the *ParaNut* processor did not scale well with the number of cores. Starting at 50MHz for a single core, the clock frequency needed to be decreased roughly 20% down to 40MHz for 2 cores and another 20% down to 33MHz for 4 cores and finally at 8 cores only 20MHz are possible on a current generation FPGA board. One of the several reasons is the combinatorial logic inside the Memory Unit (MemU) that arbitrates the accesses of the CPUs to the shared cache. [Bahle u. a. 2020]

4.1 Memory Unit (MemU)

The Memory Unit (MemU) plays a vital role in the performance of a *ParaNut* processor, especially in multi core systems. The complex mechanisms to ensure high performance during parallel execution (threaded and SIMD) are also responsible for a large part (about 30%) of the FPGA resources the processor needs. Furthermore most of the MemUs submodules are either Mealy machines (Readport/Writeport) or combinatorial logic (Arbiter) which leads to critical timing paths and affects the possible clock speed.
Figure 4.1 is a schematic of the MemU and its core components without any connection shown. This example features the Readports and Writeports for 2 cores with a capability >= 2 and 2 Cache Banks. The Cache Tagram, Bus Interface and the Arbiter are present regardless of configuration.

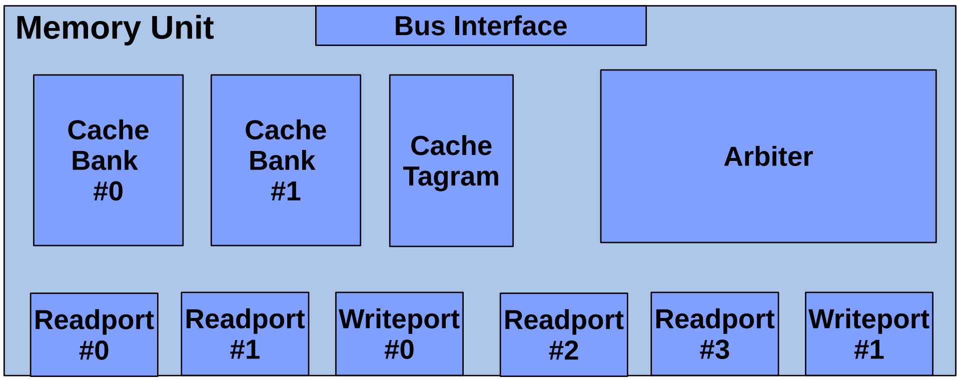

Figure 4.1: Memory Unit blockdiagram for 2 cores and 2 Banks

4.1.1 Cache Tagram

The Cache Tagram is optimized to serve multiple cores in parallel. This is achieved by replicating the Tagram and thus allowing independent access to the cache tag information. Figure 4.2 depicts the address and control signal flow towards the Cache Tagram distributed to the two internal Tagrams. Notice that the last Tagram also has to serve the BusIf increasing the interconnects size. The memory is ought to be fully mapped to block RAM cells but fails to do so prior to this work, resulting in a much higher resource usage.

Cache tags are organized as shown in table 4.1 in a cache tag line for a cache associativity of 4. The last few bits contain the least recently used information if that feature is enabled.

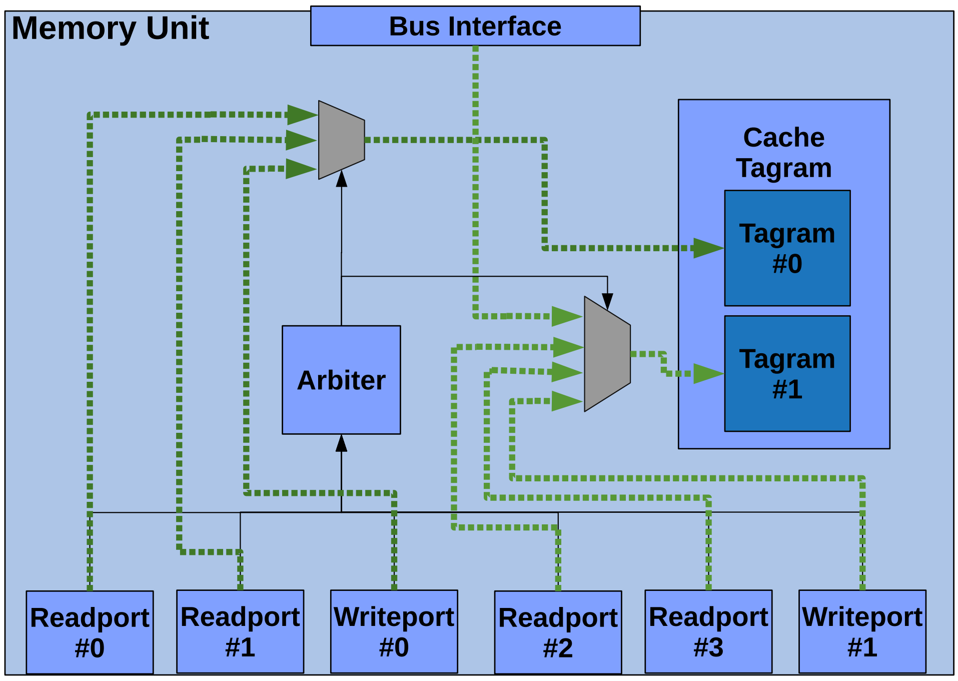

Figure 4.2: Memory Unit Cache Tagram address and control flow for 2 cores. Green: Address and Tagram control, Black: Arbiter signals

Bits	**4**	**4**	**76**	**(n)**
Desc.	valid(0 to 3)	dirty(0 to 3)	taddr(19)(0 to 3)	(lru)

Table 4.1: 4-way associative Tagram line example with 84+n bit

The output towards the Readports and Writeports is only a single tag entry. For an associativity of 2 and 4 every entry inside the cache tag line output of the block RAM needs to be checked for a hit and the output selected accordingly. Since the cache entry information is used in the receiving machines to either request permission to read or write the Bankram through the Arbiter the dependencies could be eliminated to remove the possibility of creating long paths for higher associativity in multi core systems because of growing routing delays.

4.1.2 Cache Bankram

The Bankram is fully mapped to block RAM and can be configured to use as many concurrent ports as the FPGA hardware allows. For the current generation this leads to two read and write ports. For a single core *ParaNut* one port pair is used by the only EXU and the other by the BusIf. In multi core configurations the even numbered EXUs share one port pair and the odd numbered EXUs together with the BusIf the other one resulting in lower latencies and less resources overall.

Reads are guaranteed to be handled in a single cycle. This guarantee can be utilized in the arbitration logic. Writes on the other hand are only single cycle for full word writes, half-word and byte writes require the Writeport to read the memory first and then combine the data before writing a whole word. This significantly hampers the performance of software operating on byte or half-word data. Furthermore this adds unnecessary logic to the Writeport. A better solution would be to utilize the byte enable inputs of the block RAM cells to select the bytes that need to be written. [Xilinx 2019]

The challenges of routing address and write data to and read data from the Bankram to the Readports, Writeports and BusIf in multi core systems will be described in section 4.1.3 but should be mentioned here.

4.1.3 Port to Cache Interconnect

Figure 4.3 shows the data flow design of the MemU for 2 cores and 2 banks. Each core requires a Readport and Writeport to read and write data from and to the cache banks and one Readport to read instructions. Since the cache banks are connected to these ports through 2 Crossbar switches (write and read). The write Crossbar has N (number of cores) inputs and B (number of banks) outputs, resulting in $N * B$ endpoints growing linearly with the amount of cores or amount of banks. The read Crossbar on the other hand has B (number of banks) inputs and $3 * N$ outputs, leading to $3 * N * B$ endpoints and thus greater, but still linear, growth compared to the write Crossbar. The data flow towards the Writeports is needed for writing bytes and half-words to the cache, which requires to read the word from the bank and then combining it with the bytes that are written inside the Writeport before writing a full word to the bank.

The figure 4.3 only shows the data flow but the address flow is equally complex and also requires crossbar switches to connect all components with each other. Also the data flow to and from the BusIf was omitted. The BusIf has B data read inputs and B data write outputs to parallelize the accesses to the cache as much as possible.

4.1.4 Arbiter

For the Arbiter one can divide it into smaller parts based on the subject that needs to be protected from conflicting access. Each part has its unique set of inputs and outputs and a different arbitration strategy. Most of these strategies rely on a prioritization on core level. The way the current core with the highest priority is determined is configurable at compile or synthesis time by setting the `CFG_MEMU_ARBITER_METHOD`

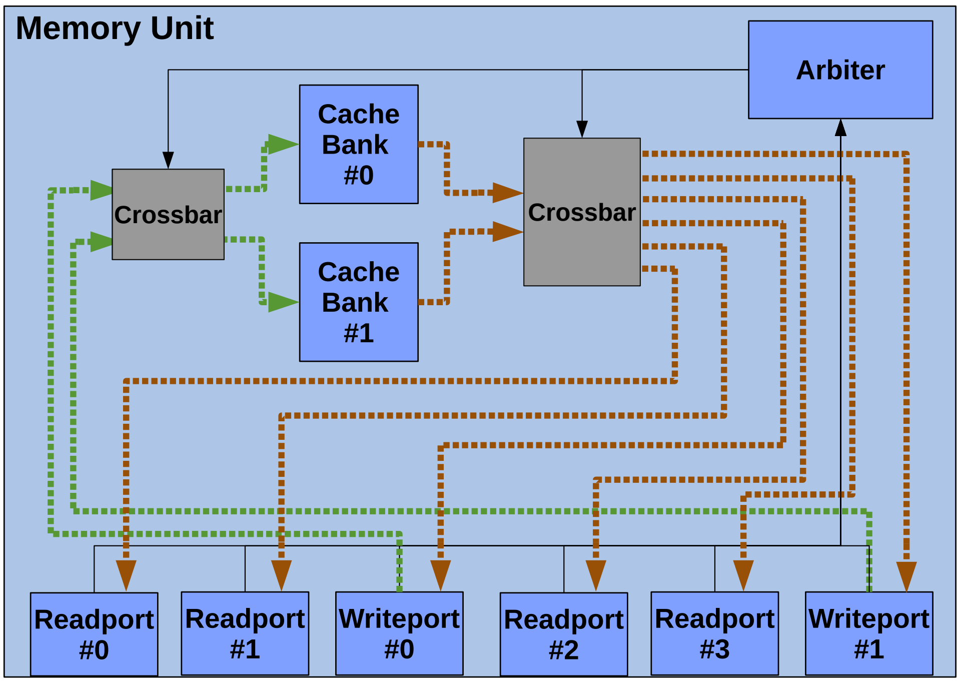

Figure 4.3: Memory Unit read write port to cache data flow for 2 cores and 2 banks. Green: Write data, Red: Read data, Black: Arbiter signals

configuration option. A round-robin option where each core has the priority for a defined amount of clocks before switching to the next is the default. The other possible option is a pseudo-random selection based on a linear-feedback shift register (LSFR).

To describe the arbitration its inputs and outputs variables are defined here and then used in the following sections:

N Number of cores in the *ParaNut* processor.

R Number of Readports in the *ParaNut* processor. Increases by 1 for each Mode 1 capable core and by 2 for each Mode 2 capable core.

W Number of Writeports in *ParaNut* processor. Is the same as N.

C Is the current core with the highest priority. Determined by an arbitrary algorithm.

Bus Interface (BusIf)

In the current implementation of the *ParaNut* architecture there is a single Bus Interface (BusIf) that handles accesses to the main system bus. Every Readport or Writeport can request and use it to either read or write single words or handle the cache related accesses to replace, flush or invalidate cache lines. Since these accesses often take a multitude of clock cycles and only one access is possible at a time the use of the BusIf needs to be arbitrated.

The arbitration function has $R + W$ request inputs and $R + W$ grant outputs one from/to each Readport and Writeport. The BusIf arbiter first checks if there are previous grants that are still needed (request and grant of the Readport or Writeport are set) and does not change the grant signals until the request is no longer set. If there is no grant set each core checks if there are any requests from its Readport or Writeports based on the following fixed prioritization:

1. The data Readport connected to the cores LSU.
2. The instruction Readport connected to the cores IFU (not present for cores with a capability of 1).
3. The data Writeport connected to the cores LSU.

Starting at the core C with highest priority the first core that has a request gets the BusIf grant in the same cycle.

Cache Linelock

The Linelock implements a mutex for a single cache line (cache tag and all data banks) between all writers to the cache. The writers include all Writeports and the BusIf. This is necessary because the cache tags and banks are not necessarily written in the same cycle, which could lead to unwanted behavior. Requests can take an arbitrary amount of cycles to complete and only one access per cache line is possible.

The arbitration function has $W + 1$ request and address inputs and $W + 1$ grant outputs from/to each Writeport and the single BusIf. Previous grants that are still requested remain granted until the request is released. During this time the address of the requesting input is not allowed to change. The current policy excludes Writeports from concurrent Linelocks even if the address is different to save area and reduce path size (not every address input has to be checked against each other) but allows the Writeports to acquire a Linelock concurrently with the BusIf if the addresses don't reference the same cache line. This means that the BusIf request is granted if its addressed cache line is different from all other granted addressed cache lines. Since the amount of inputs grows with the amount of cores this is a potentially problematic path for higher numbers of cores. For the Writeports the request is granted, starting on core priority at the core C, if no other Writeport is granted and its addressed cache line differs from the BusIf (only one address compare).

Cache Tag

The access to the cache tags is arbitrated differently for read an write accesses. Only one writer is possible at a time and a write excludes all readers. Because of the per core replicated tagram a read access on different cores are never conflicting and only reads originating from the same core need to be arbitrated.

The arbitration function has $R + W + 1$ read request inputs and $R + W + 1$ read grant outputs from/to each Readport, Writeport and the BusIf. Moreover there are $W + 1$ write request inputs and $W + 1$ write grant outputs from/to each Writeport and the BusIf. Previous grants that are still requested remain granted until the request is released. A pending write request excludes new read requests from being granted to give the writers priority and to prevent write starvation. For writers the BusIf has the highest priority followed by all Writeports by current core priority C. Read requests are handled on a per core basis and are ordered by following fixed prioritization:

0. (The BusIf read, only necessary for the last core which shares the Tagram port with the BusIf)
1. The data Readport connected to the cores LSU.
2. The instruction Readport connected to the cores IFU (not present for cores with a capability of 1).
3. The data Writeport connected to the cores LSU.

The read arbitration is constant because of the constant amount of inputs per core (either 3 or 4 for the last core). The write arbitration however is growing linearly with the amount of cores and could possibly be critical for systems with many cores.

Bankram

The arbitration for the Bankram is completely independent per bank and will be described as such in this section. As mentioned in section 4.1.2 the Bankram can be configured to use multiple block RAM ports to increase performance which is also reflected in the Bankram arbitration by allowing concurrent readers per port. However only one writer is possible per bank regardless of the number of ports.

The arbitration function has $R + W + 1$ read request inputs and $R + W + 1$ read grant outputs from/to each Readport, Writeport and the BusIf per bank. Moreover there are $W + 1$ write request inputs and $W + 1$ write grant outputs from/to each Writeport and the BusIf per bank. Previous grants that are still requested remain granted until the request is released. Read requests from the BusIf have the highest priority the rest of the read requests are prioritized by core priority C and then on the following fixed prioritization per core:

1. The data Readport connected to the cores LSU.
2. The instruction Readport connected to the cores IFU (not present for cores with a capability of 1).
3. The data Writeport connected to the cores LSU.

Some more logic takes care of granting multiple read requests per port if the address that is requested is the same for the read accesses. For the write requests again the BusIf has the highest priority. The other write requests originating from the Writeports are prioritized by core priority C.

Since these grants are also used as the select signals for the previously described port to cache crossbar interconnect (section 4.1.3) the Bankram arbitration is part of the critical path and grows quickly with the amount of cores in a *ParaNut* processor.

The usage of multiple block ram ports and splitting the possible inputs is also not reflected properly in hardware. Table 4.2 shows the index used to for example select the input address for a Bankram port correlates with the possible select values for the Bankram with a single port. In a configuration with 2 Bankram ports, shown in table 4.3, the Readport and Writeports are assigned to a Bankram port by their suffix with the formula $suffix \bmod 2$. With that the port index does no longer match the select values on both ports. This leads to the hardware shown in Figure 4.4 with `wiadr[0]_i_1` being the multiplexer for the address to the first port of the first bank and `wiadr[0]_i_2` being the multiplexer for the address to the second port of the first bank. Even though the select value going to `wiadr[0]_i_1` can only be 0, 2 or 4 it has 6 inputs. This wastes resources and can lead to longer paths.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Port0 (Select Value)	RP0 (0)	RP1 (1)	RP2 (2)	RP3 (3)	WP0 (4)	WP1 (5)	BusIf (6)

Table 4.2: Single port Bankram inputs for 2 cores - Select value is equal to index

Index	**0**	**1**	**2**	**3**
Port0 (Select Value)	RP0 (0)	RP2 (2)	WP0 (4)	
Port1 (Select Value)	RP1 (1)	RP3 (3)	WP1 (5)	BusIf (6)

Table 4.3: Dual port Bankram inputs for 2 cores - Select value is not equal to index

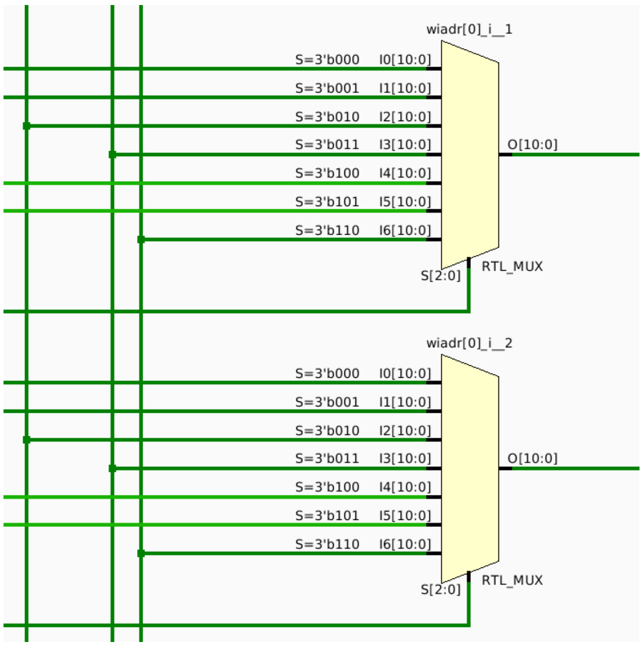

Figure 4.4: Input address select for a dual port Bankram with 2 cores

4.2 Execution Unit (EXU)

The Execution Unit (EXU) was already pretty optimized from the previous change to the RISC-V instruction set architecture and only needed a few improvements. The *MExtension* submodule, responsible for hardware accelerated multiplication and division operations, contained a long path in the multiplication function. The High-level synthesis (HLS) of this SystemC module required a carefully catered and structured module to get the correct 64bit multiplication results while still utilizing the performance of the FPGAs DSP slices. Figure 4.5 shows the multiplication function. The two 32 bit operands are coming in from EXU registers into 4 DSP (marked red) slices executing 16 bit multiplications each. The 4 32 bit results are then combined by adding them in the orange marked section into a 64 bit multiplication result from which either the top or bottom 32 bit are saved in a register before it will be written to a general purpose register (GPR) in the EXU. This of course uses more resources than would be required if the full 25x18 bit multiplier and adders in the 7-series FPGAs DSP slices would be used by implementing the *MExtension* in VHDL but is currently the only working solution for using the HLS. [Xilinx 2018]

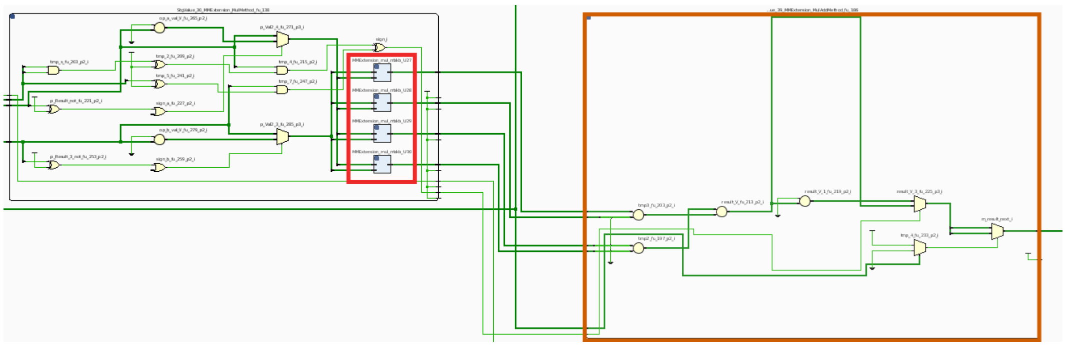

Figure 4.5: MExtension multiplication path, Red: DSP slices, Orange: Add and select function

In multi core systems another performance issue is the daisy chain used to synchronize the instruction execution inside between the EXUs when running in linked mode (Mode 1). Figure 4.6 shows the schematic of the daisy chain for 4 cores. The last of the EXUs, EXU3 in this case, `sync_i` input is fixed to 1 and every other EXU has the `sync_o` output of the previous EXU as input. Internally the EXUs can either pass through their `sync_i` or a value of 0 using a multiplexer. The multiplexer is controlled by a small bit of logic and only break the chain if the EXU is running in the linked mode. During linked execution the multiplexer will output 0 until the current instruction is finished and the EXU is ready for the next instruction. This means that until every EXU finished the EXU0 (CePU) will not instruct its Instruction Fetch Unit (IFU) to supply the next instruction, even if itself has already handled the current one. The signal from the EXU0 to every other EXU is used to activate the EXUs again since they otherwise would not know when the next instruction is ready.

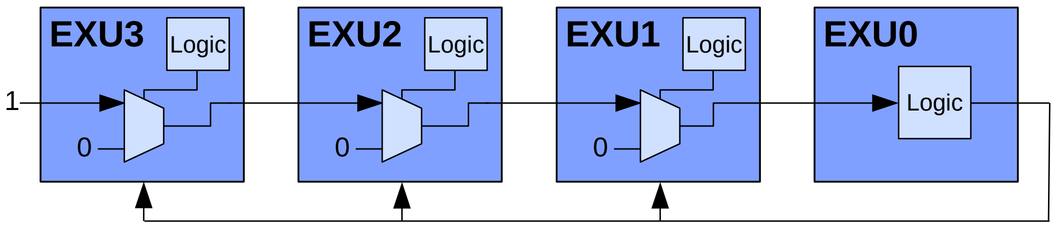

Figure 4.6: Schematic of the linked mode synchronization daisy chain for 4 EXUs

4.3 Load Store Unit (LSU)

The Load Store Unit (LSU) plays an important role in the performance of the *ParaNut* processor. It features a write buffer configurable in size. Each data access requested by an EXU is first entering a LSU before it will be relayed to the Readport or Writeport inside the MemU. Since some of the control signals going to the MemU are combinatorial they are often part of the critical paths.

One of the problematic paths is linked to the write buffer hit detection. On a `rd` signal coming registered from the EXU the LSU checks all entries in the write buffer for a potential hit by comparing the incoming address with the address and the valid bits saved in the entries of the buffer. On a hit the LSU only sends a read request to the Readport if the valid bits do not match the requested bits (`bsel`). Since the Readport itself is also a Mealy machine it will start requesting an access to either the Cache Tagram or the BusIf depending on the address. If the request is granted in the same cycle the state of the Readport is changed.

Another place for improvement is the implementation of the load reserved and store conditional instructions. On a load reserved the address that is read needs to be saved inside the Writeport to assess if a store conditional at a later point in time is successful or not. To do that the write buffer needs to be flushed first and then the current address is routed to the Writeport. Listing 1 shows the interface towards the Writeport and the many outputs depending on input signals. For the special cache invalidate, write back, and flush instructions another special case is needed. Again the Writeport itself is also a Mealy machine and will send requests to the Arbiter or act directly on inputs from the LSU and these outputs are thus problematic.

Lastly even though great care was taken in previous works to generate optimized VHDL code through the HLS some parts of the LSU have still room for improvements. The generation of the alignment error output (`align_err`) which is used in the EXU to evaluate load and store address misalignment exceptions for example or the generation of the half-word and byte read outputs which where unnecessarily duplicated.

```
// Generate MEMU write port signals ...
wp_adr = lres_scond & rd ? adr_var : wbuf_adr[0].read ();
wp_data = wbuf_data[0].read ();
wp_bsel = wbuf_valid[0].read ();
if (wbuf_new == 0 && ((cache_writeback == 1 || cache_invalidate == 1) ||
↪  (lres_scond && !rd))) {
  wp_adr = adr_var; // set address
  wp_data = wdata_var;
  wp_bsel = bsel;
  wp_wr = lres_scond.read (); // cannot write now
  wp_writeback = cache_writeback.read ();
  wp_invalidate = cache_invalidate.read ();
  wp_direct = 0;
  ack = wp_ack.read ();
} else {
  wp_wr = wbuf_dirty0.read ();
  wp_writeback = 0;
  wp_invalidate = 0;
  wp_direct = !dcache_enable | !AdrIsCached (wbuf_adr[0].read ());
}
```

Listing 1: lsu.cpp, LSU to Writeport interface before

5 Optimization

This chapter describes the measures and changes to the *ParaNut* processor model and hardware to improve its performance and resource usage. It is structured in the same way the previous chapter 4 starting with the Memory Unit (MemU) and all of the sub modules that were optimized. The big updates to the arbitration logic inside the Arbiter for example. The succeeding section focuses on the Execution Unit (EXU) and the changes needed to remove long timing paths there. In the last section the Load Store Unit (LSU) is completely refactored and presented.

5.1 Memory Unit (MemU)

5.1.1 Cache Tagram - Write in one cylce

After some investigation the block RAM used for the Tagram offers the possibility to have asymmetric read and write widths. [Xilinx 2016] For caches with enabled associativity this enables a writer to update a single cache tag, while still retaining the ability to read a full cache tag line containing 2 or 4 tags. This removes the necessity to read the cache tag line to first before updating the information of a single tag. This saves a clock cycle from each cache write that requires to update the cache tag (validating, invalidating, setting the dirty bit) and simplifies the Writeport and BusIf state machines.

To enable the write of a single tag entry the information is split into distinct entries as seen in table 5.1. Each 21 bit tag entry featuring its valid and dirty bits and the associated address are grouped together. The block RAM for this configuration would be configured to have an 84 bit wide read port with a number of cache sets (CS) bit address and a 21 bit wide write port with an $CS + 2$ bit address to distinctively address the different cache ways. The least recently used information can no longer be stored in the same block RAM and will therefore require more block RAM cells.

Bits	**21**	**21**	**21**	**21**
Desc.	v|d|taddr(19)	v|d|taddr(19)	v|d|taddr(19)	v|d|taddr(19)

Table 5.1: 4-way associative Tagram line example with 84 bit

Implementing this structure required the implementation of a new inference architecture inside the `mem_inferred.vhd`. Listing 2 shows the entity declaration for the new `mem_sync_read_wider_dp_inferred` having independent read and write ports with configurable address and data width. The way it is set up the write port data width (`DWIDTHW`) needs to be smaller or equal to the read port data width (`DWIDTHR`) and should also be a divider of the width. If both data widths and address widths are set equal the memory will be infered like a normal block RAM with symmetric read and write ports. Thus no change is needed for configurations with no associativity.

```vhdl
entity mem_sync_read_wider_dp_inferred is
        generic (
                DWIDTHW : integer := 21;
                AWIDTHW : integer := 10;
                DWIDTHR : integer := 16;
                AWIDTHR : integer := 84
        );
        port (
                clk   : in std_logic;
                we    : in std_logic;
                waddr : in std_logic_vector(AWIDTHW - 1 downto 0);
                raddr : in std_logic_vector(AWIDTHR - 1 downto 0);
                wdata : in std_logic_vector(DWIDTHW - 1 downto 0);
                rdata : out std_logic_vector(DWIDTHR - 1 downto 0)
        );
end mem_sync_read_wider_dp_inferred;
```

Listing 2: mem_inferred.vhd, Entity for inference of block RAM with a wider read than write port

The SystemC model of the Tagram was also changed to represent this hardware behavior by simply reading and replacing the tag in one cycle instead of implementing a similar asymmetrical memory.

5.1.2 Cache Bankram - Use of byte enable signals

As described in section 4.1.2 the Bankram is not able to write half-word or bytes in a single cycle. The Writeport needs to read the current value from the bank and combine the data before writing a full word back to the bank. This sequence can be eliminated by utilizing the byte enable bits of the block RAM cells present in the Xilinx FPGAs. "The byte-write enable feature of the block RAM allows writing eight-bit (one byte) portions of incoming data." [Xilinx 2019]

The implementation of this feature is handled by adding a byte-write enable port `we` to the `mem_sync_sp_inferred` and `mem_sync_true_dp_inferred` entities in the `mem_inferred.vhd`. Since the Writeport already has the information which bytes are being written through its `byte_sel` input from the LSU this information only needs to be routed to the byte-write enable ports. This does not only lead to byte and half-word writes needing less clock cycles and a simplified Writeport state machine, but also eliminates the necessity of bank data to be routed to the Writeports. This data path is visible in Figure 4.3 and upon removal reduces the read Crossbar endpoints to $2 * N * B$ from $3 * N * B$.

5.1.3 Port to Cache Interconnect

The port to memory interconnect was not changed for this work because of time constraints. As mentioned in section 5.1.2 the improvements to the Bankram did reduce the size of the data read Crossbar. A *ParaNut* with an implementation of an Omega Network as described in section 3.2.2 could probably help with scalability for systems with more cores.

Figure 5.1 shows a possible solution for using an Omega Network to route the address output from all Readport and Writeports in a 2 core system to 4 Bankrams with one port. Since this configuration is not symmetrical and there are more inputs than outputs, some of the switches can be simplified (pass-through) or removed completely. This is represented by the red marked connections and switches. The figure 5.2 below displays the simplified Omega-Network with only 6 switches necessary to route every input to every output. In contrast to the current Crossbar implementation this would be highly blocking and a different arbitration scheme would be necessary to accommodate this change.

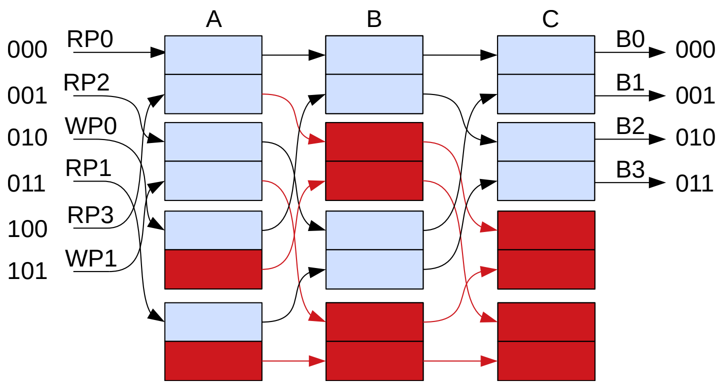

Figure 5.1: Address routing from 2 cores to 4 Bankrams through an Omega Network

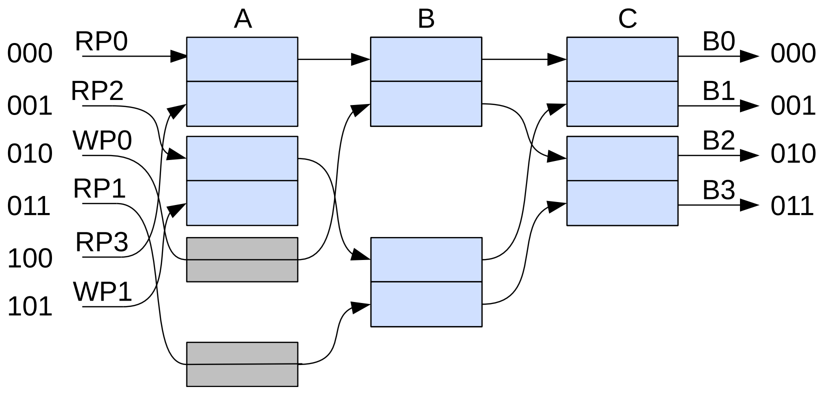

Figure 5.2: Address routing from 2 cores to 4 Bankrams through an Omega Network - simplified

To keep the same amount of parallelism the Omega Network could only use one output and be replicated per Bankram, the same way the Crossbar switches are currently set up for each Bankram. Figure 5.3 shows the necessary paths and switches for this setup.

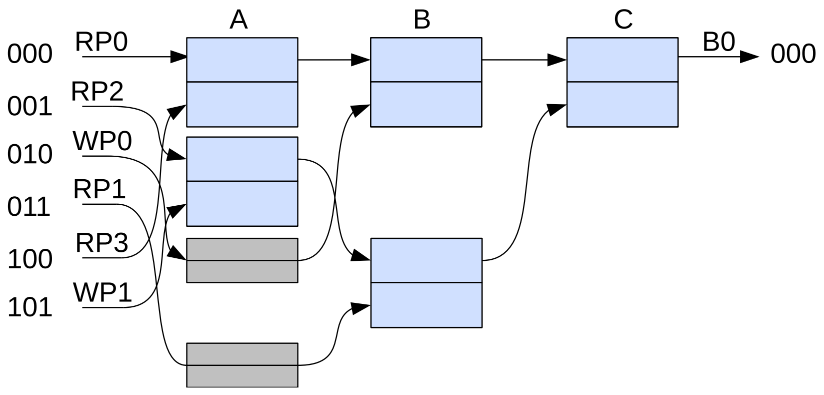

Figure 5.3: Address routing from 2 cores to 1 Bankram through an Omega Network - simplified

In a same manner the data interconnects to and from the banks could be implemented. The only difference being the amount of inputs and outputs. For the data write interconnect for example at 4 cores and 4 Bankrams only 2 stages with 2 switches would be necessary to connect all Writeport data outputs to all bank data inputs. Or for improved parallelism 2 stages and 3 switches if there is an Omega Network per bank. Figure 5.4 shows the proposed write data architecture using an Omega Network.

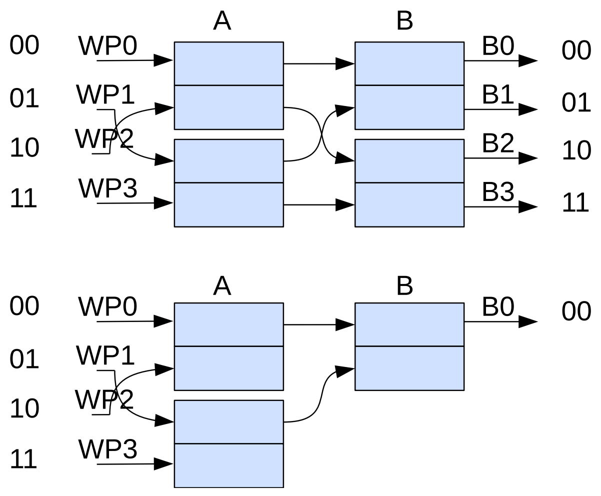

Figure 5.4: Write data routing from 4 cores to either 4 Bankrams or 1 Bankram through an Omega Network

5.1.4 Arbiter

In an effort to generate constant paths inside the Arbiter a new entity was defined called `selector`. This configurable component is designed to have a fast input and a slow input. Listing 3 shows the interface of the `selector` and the configuration options. The fast inputs are prefixed with a `f_` and the slow inputs with a `s_`. The data inputs are not necessary for all use cases and will be optimized out if `DWIDTH` is set to 0 and the ports left unconnected. The select input is usually the index of one of the Readports or Writeports and the select valid is equivalent to a request by that Port. The `prio` input is the current core number with priority defined in 4.1.4. It is used to determine if the fast input has priority if the `prio` is less than the fixed configured `FAST_INDEX` or if in turn the slow input has priority.

```vhdl
entity selector is
  generic (
    DWIDTH         : integer := 1; -- Width of the data that gets arbitrated
    SEL_MAX        : integer := 1; -- Maximum value for sel (usually
    ↪  RPORTS+WPORTS)
    FAST_INDEX     : integer := 0  -- If prio < FAST_INDEX the fast inputs have
    ↪  priority else the slow inputs
  );
  port (
           clk            : in std_logic;
           reset          : in std_logic;
           f_dat_in       : in std_logic_vector(DWIDTH-1 downto 0);
           f_sel_in       : in integer range 0 to SEL_MAX;
           f_sel_valid_in : in std_logic;
           s_dat_in       : in std_logic_vector(DWIDTH-1 downto 0);
           s_sel_in       : in integer range 0 to SEL_MAX;
           s_sel_valid_in : in std_logic;
           prio           : in unsigned(MAX(0, CFG_NUT_CPU_CORES_LD-1) downto 0);
           dat_out        : out std_logic_vector(DWIDTH-1 downto 0);
           sel_out        : out integer range 0 to SEL_MAX;
           sel_valid_out  : out std_logic
  );
end selector;
```

Listing 3: marbiter.vhd, `selector` entity for constant arbitration in the Arbiter

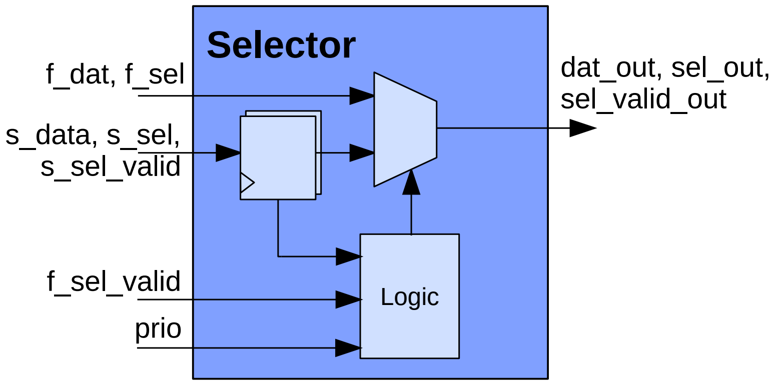

Figure 5.5: Schematic of the selector used in the Arbiter

Bus Interface (BusIf)

The BusIf Arbitration is changed to use the `selector` to determine the Readport or Writeport that can use the BusIf to execute a command. Because a single command is 72 Bits (3 Command + 1 Nolinelock + 4 Byte-Select + 32 Address + 32 Write Data) the data ports of the selector are not used to save resources. To reduce the amount of `selector` stages needed for each core one port is selected based on the fixed priority defined in 4.1.4. The Figure 5.6 is a schematic of the selector stages for 4 cores. The Readport and Writeports are sorted as follows:

- N Data Readports with select index 0 to $N - 1$
- N Instruction Readports with select index N to $(2 * N) - 1$
- N Data Writeports with select index $(2 * N)$ to $(3 * N) - 1$

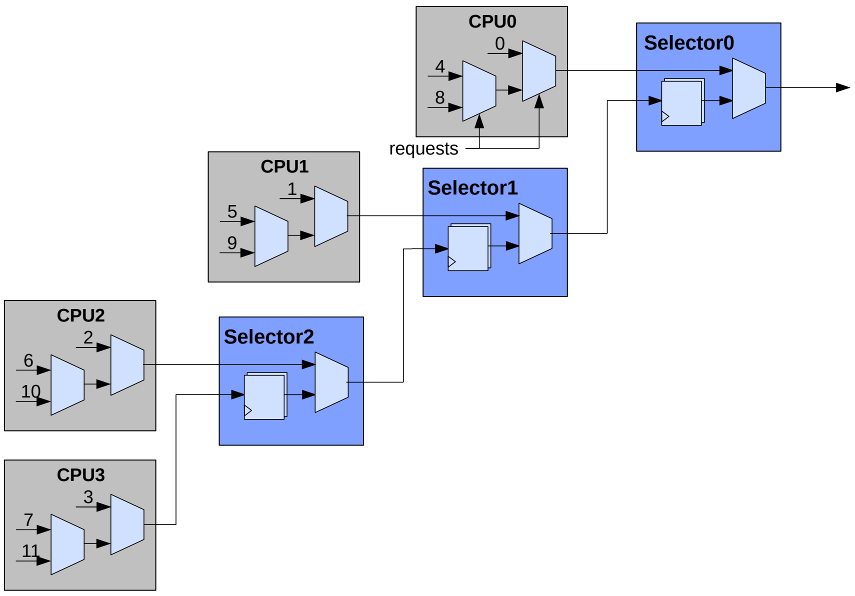

Figure 5.6: Schematic of the BusIf arbitration selector stages for 4 cores

The resulting output of Selector0 is used to set the BusIf grant signal and to select the output in a multiplexer multiplexing the command inputs from all Readports and Writeports. This configuration allows the CPU0 (CePU) to receive a grant in the same cycle the request was set. Other CPUs have a delay of 1 to N-1 cycles through the selector stages.

Cache Linelock

The Cache Linelock Arbitration also uses the `selector` component to determine the Writeport that acquires the Linelock. The BusIf Linelock is arbitrated independent of the `selector` stages and still depends on all address inputs of the Writeports. For up to 8 cores this seems to be fine but could be problematic for systems with more cores.

No data is required by this arbitration, so the data inputs of the `selector` are not used. Before entering the `selector` the requested Writeport address is compared to the BusIf address and the associated select valid input only set if different cache lines are addressed. Other than that the same rules apply as defined in section 4.1.4. Figure 5.7 shows the schematic of the Linelock arbitration selector stages for 4 cores.

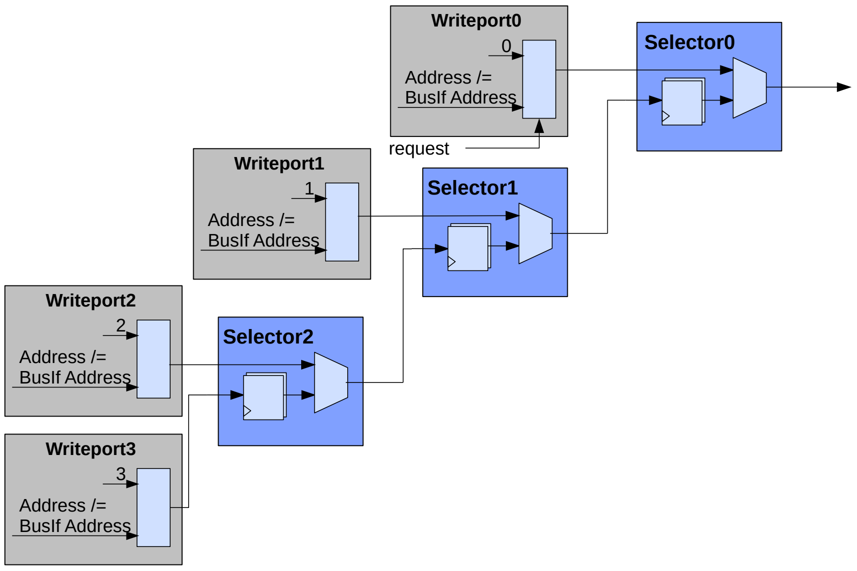

Figure 5.7: Schematic of the Linelock arbitration selector stages for 4 cores

The option of removing the Linelock was explored but deemed not possible at this stage. The snooping of write addresses for example is dependent on the Linelock and important for implementing the *load reserved* and *store conditional* instructions for synchronization between threads.

Another idea was to add a locking mechanism directly to the Tagram by expanding a cache tag line to contain a single lock bit or id field. A writer would set this field during its tag read and unset it after the data write. Since the Cache Tagram is replicated for all CPUs the tag setting of this field would require a tag write lock. This was not yet implemented and tested but could probably further improve the scalability of a multi core *ParaNut* processor.

Cache Tagram

The Tagram arbitration was not changed directly in the Arbiter because of the already independent read arbitration from replicated Tagrams. The write arbitration on the other hand is still growing but was not deemed in need of a change for this work. Nonetheless the way the Readport and Writeports react to the read grant were changed. Both were waiting in their idle state until they got a grant. Making this state transition independent and waiting in the following state does improve the clock speed while not adding clock cycles to the overall read or write operations the ports execute.

Cache Bankram

The Bankram arbitration was changed to use the `selector` for determining the entity that gets access to the bank. The other changes to the Bankram outlined in section 5.1.2 together with the fact that every access to the Bankram now only takes one cycle simplified the arbitration process. To further increase the performance the inputs to the arbitration function are sorted differently to remove unnecessary paths for selecting input addresses to the banks for configurations with more than one Bankram port as described in section 4.1.4.

The Bankram arbitration uses the `selector` data inputs and outputs to select the input address as a byproduct of the arbitration instead of the big Crossbar multiplexers at the end as shown in figure 4.4. Nonetheless the inputs are sorted differently to have the same correlation of index to select value. Table 5.2 shows that the index is now equal to the possible select values. This is achieved by adding a port dimension to the input arrays.

Index	**0**	**1**	**2**	**3**
Port0 (Select Value)	RP0 (0)	RP2 (1)	WP0 (2)	
Port1 (Select Value)	RP1 (0)	RP3 (1)	WP1 (2)	BusIf (3)

Table 5.2: Dual port Bankram inputs for 2 cores now - Select value is equal to index

Looking at figure 5.8 the input address towards the Bankrams port is either coming directly from one of the selectors (marked in red) or directly from a CPUs priority selection (marked in orange). Since the amount of Readport and Writeports per CPU is constant the path is constant for an arbitrary amount of cores. The blue marked lines are the inputs for Bankram 0 port 0 coming from the fixed priority select of CPU 0 (orange) and Bankram 0 port 1 coming from the selector to arbitrate the BusIf and CPU 1 inputs (red).

Problems with the changes

All of the changes to the Arbiter above proved to be perfect for increasing the clock frequency of the *ParaNut* in all configurations. The increased amount of clock cycles is a trade of that should be addressed in future versions. But unfortunately the tests performed for the evaluation in chapter 6 showed that this only worked for up to 4 cores. At 8 cores the long propagation time through 7 stages at the BusIf and Linelock arbitration for the last two cores lead to race condition errors where a grant was dangling and could be granted to the same core with varying side effects

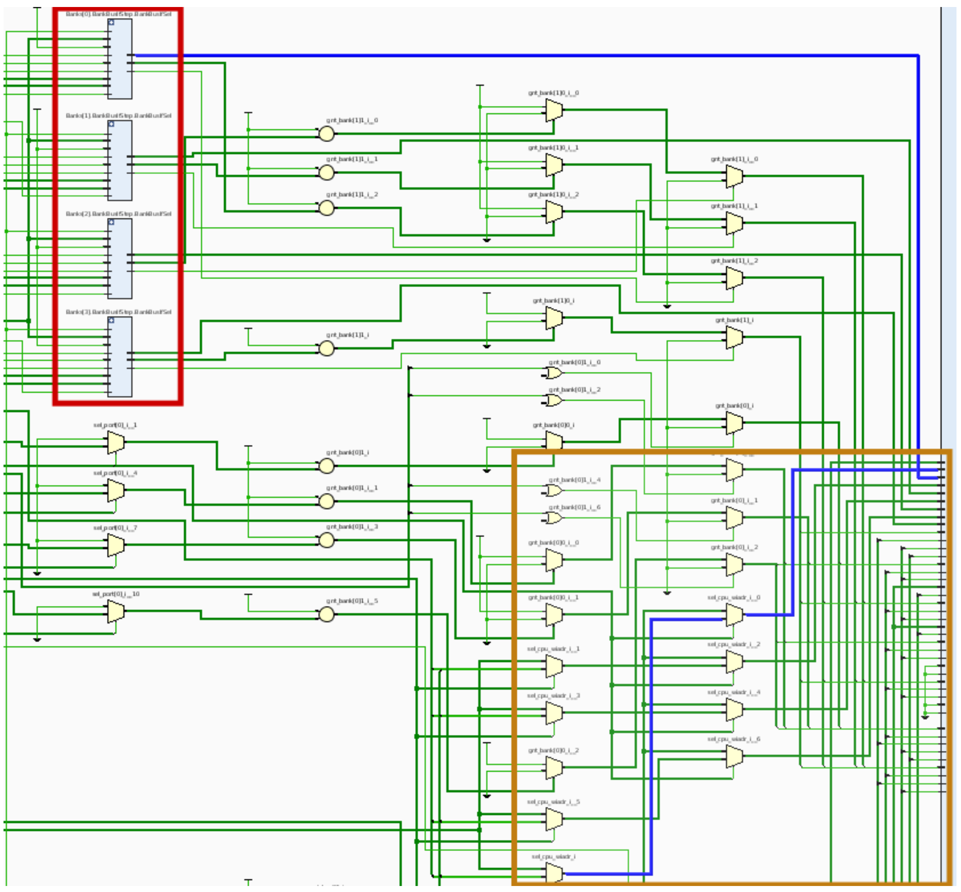

Figure 5.8: Optimized input address select for dual port Bankrams with 2 cores

(exceptions, reads/writes to wrong addresses). These errors where hard to identify and could not be fixed in time for this work.

One possible, but not yet tested or implemented, solution for this problem could be to only use one selector with the CePU as fast input and all other CoPUs use the shared slow input. This would remove the race conditions for more than 8 cores and the long propagation time but would probably introduce dependencies between the CoPUs that could lead to longer timing paths for systems with more than 8 cores. Another idea would be to add a configuration option to the selector that disables the registers. In a same way as the proposition above only the first selector for the CePU would feature registers and all others would be combinatorial.

5.2 Execution Unit (EXU)

After some improvements in the MemU problems where showing with the RISC-V M-Extension implementation for hardware multiplication and division of integers.

As mentioned in 4.2 the path from the general purpose registers of the EXU through the DSP slices and a 64 Bit adder up to the multiplication result register was too long to reach 100 MHz, mainly for routing delays to and from the DSP slices. Since the multiplication instructions already required a state change in the EXU they were executing in 3 cycles, even though the result of the multiplication would be ready after 2 cycles. An additional interim step in the M-Extension saving the results of the DSP slices in registers and adding them in the next cycle to produce the final result and saving it directly in the GPR did thus not lead to a slower execution of the instructions, but reduced the path length considerably. Figure 5.9 shows the elaborated M-Extension after the HLS. The new registers between the `MMExtension_MulMethod` and the `MMExtension_MulAddMethod` are marked in red.

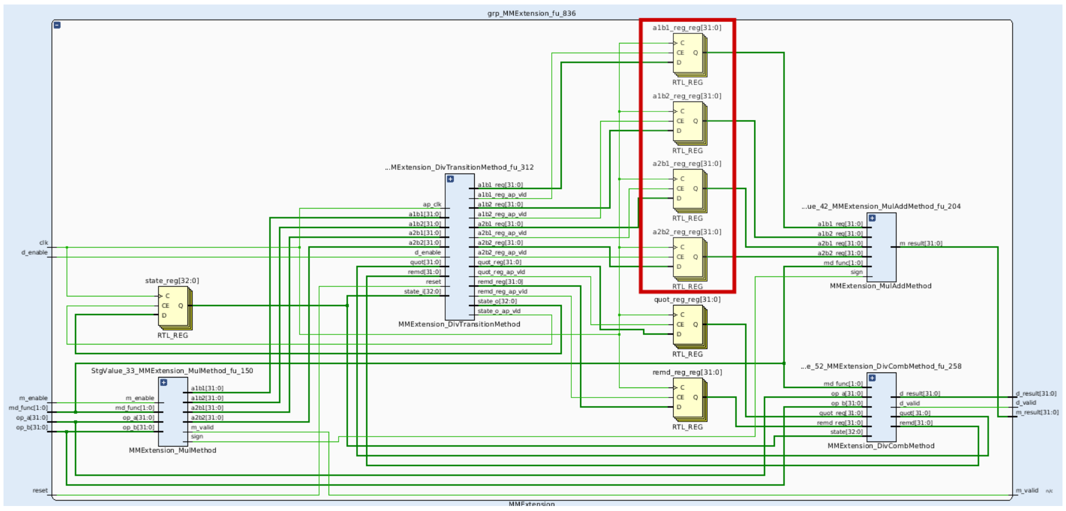

Figure 5.9: M-Extension with the added registers (red)

The synchronization daisy chain mentioned in section 4.2 did not impact the performance in the tested configurations for up to 8 cores where the paths in the MemU had a greater impact. Nonetheless the issue persists and may affect the timing closure for systems with more cores or after more optimizations to the MemU. A proposed solution would be to either add a register before the input to the CePU (EXU0) or inside the chain after a number of EXUs. Figure 5.10 shows the daisy chain for 4 cores with an added register between EXU2 and EXU1 to reduce the path length. The added clock cycle would influence the clocks per instruction (CPI) metric when running in linked mode but may be necessary to run multi core system at higher clock frequencies.

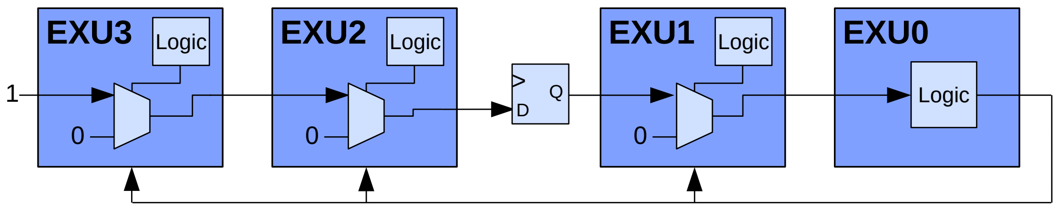

Figure 5.10: Schematic of the linked mode synchronization daisy chain for 4 EXUs with added register

5.3 Load Store Unit (LSU)

To eliminate the problems presented in section 4.3 the LSU had to be changed considerably. So a whole refactor of the LSU was started to not only increase the performance, but also the readability and adaptability of the code. The write buffer registers which where declared as independent arrays of signals are replaced with a single array of a newly defined `struct SWbufEntry` displayed in Listing 4. The `struct` shown in Listing 5 contains all the data for a single write buffer entry and implements all the necessary functions for using it as a signal inside the SystemC model. It was made sure that the structure would be fully synthesizable without any side effects. The new data field `special` inside the write buffer entry contains two bits for the special cache operations to invalidate, write back or flush a cache line. This enables these instructions to be buffered alongside the normal write operations, freeing up the processor to continue execution if the following code does not depend on the cache operation. A `fence` instruction after the cache instruction can be used to get back to strictly ordered execution by waiting for the write buffer to be emptied.

```
// Registers before - multiple arrays...
sc_signal<TWord> wbuf_adr[CFG_LSU_WBUF_SIZE];
sc_signal<TWord> wbuf_data[CFG_LSU_WBUF_SIZE];
sc_signal<sc_uint<4> > wbuf_valid[CFG_LSU_WBUF_SIZE];

// Registers after - only one array...
sc_signal<SWbufEntry>  wbuf[CFG_LSU_WBUF_SIZE];
```

Listing 4: lsu.h, LSU write buffer registers before and after

The restructure also includes the change to make the outputs toward the Writeport only dependent on registers or registered inputs. To achieve this the logic for reserving an address during a `lres (Load Reserved)` instruction had to be moved from the Writeport to the Readport to eliminate the necessity to connect the current input address of the LSU to the Writeport. Listing 6 shows the current interface to the Writeport only using registers or registered inputs.

```
struct SWbufEntry {
        sc_uint<30> adr;
        sc_uint<32> data;
        sc_uint<4> valid;
        sc_uint<2> special;

        // Necessary operators for using this structure as signals...
        bool operator== (const SWbufEntry &t) const {...}
        SWbufEntry operator= (const SWbufEntry &t) {...}

        // Displaying
        friend ostream& operator << ( ostream& os, const SWbufEntry &t ) {...}

        // Overload trace function
        friend void sc_trace( sc_trace_file* f, const SWbufEntry& t, const
        ↪   std::string& _s ) {...}
};
```

Listing 5: lsu.h, LSU synthesizable write buffer struct

```
// Generate MEMU write port signals ...
wp_adr = WbufToTWord(wbuf_var[0].adr);
wp_data = wbuf_var[0].data;
wp_bsel = wbuf_var[0].valid;
wp_writeback =  wbuf_var[0].special[0] & !wp_ack_reg;
wp_invalidate = wbuf_var[0].special[1] & !wp_ack_reg;
wp_lres_scond = lres_scond.read ();
wp_wr = wbuf_var[0].valid != 0 & !wp_ack_reg;
wp_direct = !dcache_enable | !AdrIsCached (WbufToTWord(wbuf_var[0].adr));
```

Listing 6: lsu.cpp, LSU to Writeport interface

The Readport interface could not be changed in a similar way because of the dependence on fast execution of data read instructions. The critical path being the `rd` input from the EXU, checking for a hit in the write buffer and depending on the hit starting a Readport read through the `rp_rd` output in the same cycle. The Readport in turn tries to read the Tagram as fast as possible and tries to acquire a tag read grant in the same cycle the `rp_rd` is set through the Arbiter triggering a tag read in the connected Tagram. This dependency could may be removed in future versions by starting a read in the Readport independently from the write buffer hit detection. Only in case of a full write buffer hit the tag read in the Readport would be wasted and a way to cancel a read access would be necessary inside the Readport to stop the current read transaction.

6 Evaluation

This chapter describes the tests and their results in order to evaluate the changes to the *ParaNut* done in this work. Starting with the next section 6.1 "Resource Usage" where the used FPGA resources before and now are compared. The achieved timing improvements are shown in section 6.2 "Timing Results". And lastly the section 6.3 "Benchmark Results" presents the industry standard benchmark and internal test software results in relation to the findings of the Embedded World 2020 paper.

The evaluation configuration is outlined in table 6.1. The clock speed used for the different benchmarks are defined in the respective section. These values were partly chosen to ensure comparability to the results presented in the Embedded World 2020 paper. The rest of the system architecture is the same as described in the paper and shown in figure 6.1. [Bahle u. a. 2020]

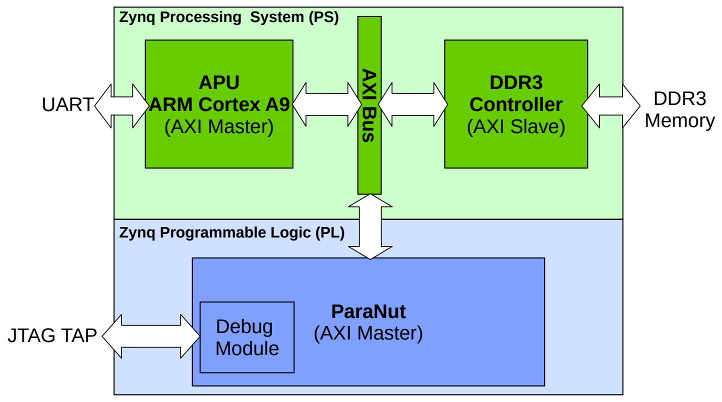

Figure 6.1: Block diagram of the *ParaNut* system

Parameter	Value
Clock Speed	20 ... 100MHz
CPU Cores	1 ... 8
M-Extension	✓
A-Extension (lr.w and sc.w)	✓
Performance Counter Enable	1
Cache size	32 kB
Cache sets	512
Cache line size	16 Bytes (4 Banks)
Cache associativity	4 ways
Cache replacement strategy	LRU
Instruction buffer size (IFU)	4 words
Write buffer size (LSU)	4 words
MEMU arbitration	7 (256 cycles)

Table 6.1: The *ParaNut* Benchmark configuration

6.1 Resource Usage

The resource usage of the *ParaNut* processor was positively influenced by the changes made in every respect. Table 6.2 shows the resource usage of the *ParaNut* for different configurations of cores for the Embedded World Conference 2020 paper. One capability 3 core and up to 7 cores with a capability of 2 are displayed. The *Freedom E310* processor based on a *RocketChip* was synthesized for the *Artix-7 35T Arty FGPA Evaluation Kit* to have a comparable processor and its results for a single core, called “tile“, are shown in the last row. [Bahle u. a. 2020] Table 6.3 displays the results for the same configuration after the optimizations. The amount of used slices is put in proportion with the slices used before and the percentage of saved resources is listed in the last column. On average the *ParaNut* processor is 21% smaller now and a single core *ParaNut* is using less slices than the *Freedom E310*.

Cores	Slice LUTs	Slice FFs	Slices	Increase
1	7,139	3,759	2,504	1.00
2	12,433	6,094	4,185	1.67
4	23,599	10,712	7,265	2.90
8	44,393	19,536	12,473	4.98
Freedom E310 (1 tile)	6,713	4,131	2,139	

Table 6.2: Zynq 7000 resource usage before
Source: [Bahle u. a. 2020]

Cores	Slice LUTs	Slice FFs	Slices	Increase	Ratio to 6.2
1	5,463	3,649	1,984	1.00	0.79
2	9,626	5,905	3,506	1.77	0.84
4	16,900	10,462	5,799	2.92	0.80
8	31,381	19,113	9,147	4.61	0.73

Table 6.3: Zynq 7000 resource usage after the optimization with saving compared to table 6.2

6.2 Timing Results

The timing results are collected by setting the input clock frequency in the FPGA settings to a value and checking the timing results after the implementation to see if the timing requirements could be met. For all configurations the default Vivado implementation strategy was chosen to produce fair results. The findings are displayed in table 6.4 for the unoptimized and optimized *ParaNut* . The rows show the values for a growing number of up to 8 cores. The *Ratio to single core* column displays the ratio compared to the single core configuration of the same optimization level. The *Ratio to Unopt.* on the other hand shows the optimization improvement compared to the same configuration of the unoptimized *ParaNut* .

Looking at the ratios one can see that the *ParaNut* before looses about 20% of maximum clock speed from 1 to 2 and from 2 to 4 cores compared to the now optimized one only loosing about 10% for each increase. At 8 cores the unoptimized *ParaNut* drops around 40% clock speed from 4 cores. The optimized *ParaNut* still drops around 44% clock speed from 4 to 8 cores indicating that there is still potential to improve the scalability even though some increased delay can be attributed to the higher resource usage of more cores, making it harder for the place and route step to meet the timing targets.

Comparing the two generations of *ParaNut* the biggest improvement can be seen for the 4 core configuration. The *ParaNut* it is now able to run at nearly two and a half times the speed before. All the other configurations also could at least double their clock frequency.

	Cores	**MHz**	**Ratio to single core**	**Ratio to before**
Before	1	50	1.00	
	2	40	0.80	
	4	33	0.66	
	8	20	0.40	
After	1	100	1.00	2.00
	2	90	0.90	2.25
	4	80	0.80	2.42
	8	45	0.45	2.25

Table 6.4: Zynq 7000 maximum clock frequency

6.3 Benchmark Results

To evaluate the impact on real world software different industry standard benchmarks and some test applications where executed on the *ParaNut* systems. Some of these only measure the single core performance, for example the Dhrystone benchmark, whilst others like the CoreMark benchmark can be used to evaluate multi core systems.

All software was compiled using the GCC Version 8.3.0 (SiFive GCC 8.3.0-2019.08.0). The optimization options -O2 or -O3 and the architecture -march=rv32im -mabi=ilp32 were used to match the underlying hardware.

Some optimization towards a higher base clock frequency did cost some performance on a per MHz basis. Looking at the Embedded World Conference 2020 paper results shown in table 6.5 the CoreMark benchmark was run at 20MHz and the number of threads increased from 1 up to 8. [Bahle u. a. 2020] The list also features results from the official EEMBC database for another softcore (MicroBlaze) and another RISC-V core (HiFive Unleashed). [EEMBC 2019] The table 6.6 below shows the CoreMark results after the optimization. The benchmark was also executed at 20MHz and with growing number of used threads. The results for 8 cores are not shown because of the problems with the new arbitration described in section 5.1.4 because of which the benchmark could not execute properly.

For a single core only about 2% where lost on a per MHz basis which can easily be salvaged by increasing the clock frequency to much higher values as shown in section 6.2. For multiple cores the hit may seem bigger (8% for 2 cores and 25% for 4 cores) but mostly result from the strict arbitration policies described in section 5.1.4.

Processor	Cores	CoreMark/MHz	Speedup
ParaNut	1	0.87	1.00
	2	1.73	2.00
	4	3.45	3.97
	8	6.59	7.59
MicroBlaze [EEMBC 2019]	1	1.90	
HiFive Unleashed [EEMBC 2019]	1	2.01	

Table 6.5: CoreMark benchmark results before
Source: [Bahle u. a. 2020]

Processor	Cores	CoreMark/MHz	Speedup	Ratio to 6.5
ParaNut	1	0.85	1.00	0.98
	2	1.35	1.58	0.92
	4	2.21	2.60	0.75
	8	-	-	-

Table 6.6: CoreMark benchmark results after the optimization with comparison to table 6.5

Another industry standard is the Dhrystone benchmark. It can only evaluate the single core performance of a processor and thus is run on the single core *ParaNut* configurations. Once with the implementation at the time of the Embedded World 2020 paper and once with the changes of this work at their maximum possible clock speed. The results for 1,000,000 iterations are shown in table 6.7. The 2% performance hit seen with the CoreMark benchmark running on a single core is not visible in the Dhrystone benchmark, neither looking at the performance per MHz nor at the overall score. This can have many reasons but is probably a result of the imprecise time measuring used in the benchmark that only uses full seconds.

	Dhrystones/MHz	**Ratio**	**Dhrystones**	**Ratio**
Before (50MHz)	909.09	1.00	45,454.5	1.00
After (100MHz)	909,09	1.00	90,909.1	2.00

Table 6.7: Dhrystone benchmark results

7 Conclusion

In the context of this work, the SystemC model and VHDL implementation of the *ParaNut* processor were optimized using standard timing closure techniques and redesign of core components. Some changes to further improve the scalability of the *ParaNut* processor by implementing a different internal interconnect were described. A set of industry standard benchmarks and special test software were used to validate the functionality and effects of the changes. The target clock frequency of 100MHz for a single core *ParaNut* is now possible on a current generation Xilinx FPGA. All while the resource usage could be reduced by 21% on average for different configurations with 1 to 8 CPU cores and still having the same functionality as implementations before. The performance of the single core *ParaNut* decreased by only 2% while the changes allow the processor to run at twice the speed compared to before. Although the systems with multiple cores do not yet reach 100MHz, the possible clock frequency has been at least doubled by the optimizations and further improvements in future works will benefit from the implemented changes.

Bibliography

Bahle u. a. 2020
Bahle, Alexander ; Kiefer, Gundolf ; PfÃ¼tzner, Anna K. ; Vollbracht, Lutz: *"The ParaNut/RISC-V Processor - An Open, Parallel, and Highly Scalable Processor Architecture for FPGA-based Systems", Proceedings of the* embedded world Conference*, Nuernberg 2020, Feb. 25-27.* 2020 [Seite 3, 9, 33, 34, 36]

Diaconis u. a. 1983
Diaconis, Persi ; Graham, R.L. ; Kantor, William M.: "The Mathematics of Perfect Shuffles". In: *Advances in Applied Mathematics* (1983), Nr. 4, S. 175–196 [Seite 8]

EEMBC 2019
EEMBC, Embedded Microprocessor Benchmark Consortium: *"CoreMark Benchmark Scores Database.", December 2019.* https://www.eembc.org/coremark/scores.php. Version: 2019 [Seite 36]

Kahng u. a. 2011
Kahng, Andrew B. ; Lienig, Jens ; Markov, Igor L. ; Hu, Jin: *"VLSI Physical Design: From Graph Partitioning to Timing Closure".* Springer, 2011 [Seite 5]

Kiefer u. a. 2015a
Kiefer, Gundolf ; Bahle, Alexander ; Meyer, Christian: *The ParaNut Processor - Architecture Description and Reference Manual*, 2015 [Seite 3, 4]

Kiefer u. a. 2015b
Kiefer, Gundolf ; Seider, Michael ; Schaeferling, Michael: "ParaNut – *An Open, Scalable, and Highly Parallel Processor Architecture for FPGA-based Systems", Proceedings of the* embedded world Conference *2015, Nuernberg, Feb. 24-26.* 2015 [Seite 3]

Lawrie 1975
Lawrie, Duncan H.: "Access and Alignment of Data in an Array Processor". In: *IEEE Transactions on Computers* C-24 (1975), Nr. 12, S. 1145–1155 [Seite 8]

Patterson u. Hennessy 2014
Patterson, David A. ; Hennessy, John L.: *"Computer Organization and Design - The Hardware/Software Interface".* Elsevier, 2014 [Seite 6]

Scudder u. Reynolds 1939

Scudder, F. J. ; Reynolds, J. N.: "Crossbar dial telephone switching system". In: *Electrical Engineering* 58 (1939), Nr. 5, S. 179–192 [Seite 7]

Simpson 2015

Simpson, Philip A.: *"FPGA Design - Best Practices for Team-based Reuse"*. Springer, Cham, 2015 [Seite 6]

Waterman u. a. 2019

Waterman, Andrew ; Asanović, Krste ; RISC-V Foundation: *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*, 2019 [Seite 2]

Whatcott u. Xilinx Inc. 2008

Whatcott, Rhett ; Xilinx Inc.: *"Timing Closure 6.1i", WP331 (v1.0.2)*. 2008 [Seite 6]

Xilinx 2016

Xilinx: *"Vivado Design Suite User Guide Synthesis (UG901)"*, 2016 [Seite 20]

Xilinx 2018

Xilinx: *"7 Series DSP48E1 Slice (UG479)"*, 2018 [Seite 18]

Xilinx 2019

Xilinx: *"7 Series FPGAs Memory Resources (UG473)"*, 2019 [Seite 12, 22]