



**Hochschule**  
**Augsburg** University of  
Applied Sciences

Faculty of Computer Science

MASTER PROJECT REPORT  
**A Memory Management Unit**  
**for the *ParaNut***

Christian H. Meyer (2063665)

Supervisor: Prof. Dr.-Ing. Gundolf Kiefer

July 3, 2022



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0/>.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Purpose of This Work . . . . .	1
1.3	Structure of This Work . . . . .	1
<b>2</b>	<b>Basics</b>	<b>3</b>
2.1	The <i>ParaNut</i> Architecture . . . . .	3
2.1.1	Concept . . . . .	3
2.1.2	Structure . . . . .	3
2.2	(Virtual) Memory Management . . . . .	4
2.2.1	Requirements . . . . .	4
2.2.2	Paging . . . . .	6
2.3	RISC-V . . . . .	7
2.3.1	An Open and Free Instruction Set Architecture . . . . .	7
2.3.2	Privilege Modes . . . . .	7
2.3.3	Control and Status Registers . . . . .	9
2.3.4	"SYSTEM" Instructions . . . . .	15
2.3.5	Virtual Address Translation . . . . .	16
<b>3</b>	<b>Implementing RISC-V's Privilege Modes</b>	<b>19</b>
<b>4</b>	<b>Virtual Memory Model for the <i>ParaNut</i></b>	<b>21</b>
<b>5</b>	<b>Implementing a Page Table Walker</b>	<b>26</b>
<b>6</b>	<b>Implementing a Translation Lookaside Buffer</b>	<b>28</b>
<b>7</b>	<b>Validation</b>	<b>30</b>
7.1	TLB Test Bench . . . . .	30
7.2	MMU Test program . . . . .	30
7.3	Booting Linux . . . . .	32
<b>8</b>	<b>Conclusion</b>	<b>34</b>
8.1	Summary . . . . .	34
8.2	Future work . . . . .	34
	<b>References</b>	<b>36</b>

## Abbreviations

<b>ABI</b>	application binary interface
<b>AEE</b>	application execution environment
<b>BRAM</b>	Block RAM
<b>BusIf</b>	bus interface
<b>CePU</b>	central processing unit
<b>CoPU</b>	coprocessing unit
<b>CSR</b>	control and status register
<b>DLP</b>	data-level parallelism
<b>ExU</b>	execution unit
<b>FPGA</b>	field-programmable gate array
<b>HDD</b>	hard drive disc
<b>HLS</b>	High-Level-Synthesis
<b>IFU</b>	instruction fetch unit
<b>IR</b>	instruction register
<b>ISA</b>	instruction set architecture
<b>IO</b>	Input/Output
<b>LRU</b>	Least Recently Used
<b>LSU</b>	load-store unit
<b>M-mode</b>	machine-mode
<b>MemU</b>	memory unit
<b>MMU</b>	memory management unit
<b>OS</b>	operating system
<b>PC</b>	program counter
<b>PLRU</b>	Pseudo Least Recently Used
<b>PPN</b>	physical page number
<b>PTE</b>	page table entry
<b>PTW</b>	page table walker
<b>RP</b>	read port
<b>RWXU</b>	Read, Write, Execute, User
<b>RAM</b>	random access memory
<b>S-mode</b>	supervisor-mode
<b>SEE</b>	supervisor execution environment
<b>SBI</b>	supervisor binary interface
<b>TLB</b>	translation lookaside buffer
<b>TLP</b>	thread-level parallelism
<b>U-mode</b>	user-mode
<b>VPN</b>	physical page number
<b>WP</b>	write port

# 1 Introduction

## 1.1 Motivation

Since the RISC-V ISA was introduced, a lot of cores using its architecture were introduced[1]. The *ParaNut* is a RISC-V compatible and highly scalable soft-core for field-programmable gate arrays (FPGAs), which is developed at the Augsburg University of Applied Sciences and is publicly available with a permissive open source license [2]. Similar to many other RISC-V cores, the *ParaNut* was only capable of running bare-metal and real-time operating systems like FreeRTOS<sup>1</sup> code prior to this work. This required a huge effort from software developers, e.g. for driving Input/Output (IO) devices or managing memory. Apart from the required implementation time, writing a lot of code is error-prone and might introduce security flaws.

To simplify work, software developers can use modern operating systems, which often come with a full set of drivers for different peripherals and security features. However, many operating systems require dedicated hardware to support virtual memory management like the Linux kernel.

## 1.2 Purpose of This Work

To serve as a hardware basis for the Linux kernel, in the *ParaNut*, this work shows how a memory management unit (MMU) was implemented into the *ParaNut* and how the architecture was prepared to support virtual memory. It introduces RISC-V's privilege modes, which enables separate software layers, and the new hardware modules page table walker (PTW) and translation lookaside buffer (TLB). Focus was set to enable synthesis configurability for all new features, especially that the MMU can be disabled completely to save chip area.

## 1.3 Structure of This Work

Section “Basics” introduces the *ParaNut* architecture, gives an overview of virtual memory management with special focus on paging and also discloses relevant parts of the RISC-V privileged architecture. This will lay a theoretical foundation for the subsequent chapters, which give more detail about the implementations done in this work. In section 3 “Implementing RISC-V's Privilege Modes”, the details of implementing privilege levels in the *ParaNut* are summarized. Afterwards, the

---

<sup>1</sup>Information about this project is going to be published soon

virtual memory model of the *ParaNut* is shown, i.e. how the new components are integrated into the *ParaNut* architecture. Next, two new modules, namely the PTW and the TLB, are introduced in the subsequent chapters “5” and “6”. Finally, the process and tools of design validation are presented in section “7” together with details on how the Linux kernel was run for the very first time on hardware.

## 2 Basics

### 2.1 The *ParaNut* Architecture

#### 2.1.1 Concept

The *ParaNut* is a new generation of soft-cores for FPGAs with focus on little area requirements in favor of few clocks per instruction. While formerly developed with the OpenRISC instruction set architecture (ISA), it was later moved to RISC-V. Furthermore, it is highly customizable and open source, allowing any individual to adapt it further. Though some performance-critical components are implemented with VHDL, its key components are implemented in the SystemC Synthesizable Subset. This allows to simulate with good speed and cycle-accuracy, while still being able to synthesize it to hardware.

In order to achieve parallelism while keeping area requirements small, it implements a special concept of data-level parallelism (DLP) and thread-level parallelism (TLP), which abstracts vector and thread programming to simple standard ISA compatible control and status register (CSR) instructions (more on CSRs and CSR modification instructions in section 2.3.3 and 2.3.4). More details on the *ParaNut* and its parallelism concept can be found in [2] and [3].

#### 2.1.2 Structure

A block diagram of the *ParaNut* with 4 cores configured is shown in figure 1. At the bottom of the picture are four full-featured cores, each containing an ALU, a Register File, Control Logic, CSRs as well as exception and interrupt (in the RISC-V ecosystem together known as *traps*) handling logic<sup>1</sup>. The combination of all these components is called an execution unit (ExU). Each is connected to the memory unit (MemU) two times: First via the IFU, which fetches instructions and stores some prefetched instructions in a small size-configurable buffer; second via a load-store unit (LSU), which handles data reads and writes and contains a little size-configurable write buffer as well.

The LSU and IFU are connected to the MemU via read ports (RPs) and write ports (WPs). The MemU itself interacts with the system bus (AXI or WishBone) via a single bus interface (BusIf) and contains a configurable number of cache banks

---

<sup>1</sup>The picture also shows an instruction register (IR) and a program counter (PC), which are actually routed from the instruction fetch unit (IFU) and do not require additional registers. Therefore, they can be seen as virtual registers.



Due to system's ability to swap data from and to main memory, e.g. to an HDD, putting it back at the exact same place as before would be inefficient and limiting. Therefore, it should be possible to relocate it anywhere in the memory. Again, the programmer is unaware of where this will be.

## Protection

The operating system (OS) and processes need protection against unwanted interference from other processes, whether accidental or intentional. Therefore, it is important that all memory accesses are checked during run time for their validity.

## Sharing

While in some cases it must be avoided that processes are able to access memory of other processes, in other cases it is desired that two processes may share the same memory region. Example: two processes execute the same program, so both may use the same copy of the program rather than having two identical copies of it.

## Physical organization

Memory can be divided into two types: main memory (random access memory (RAM)) and secondary memory, e.g. a hard drive disc (HDD). While RAM is fast and volatile, the costs per capacity are comparably high. In contrast, secondary memory is much cheaper, but slow and not volatile. Managing these two levels is a major system concern. [4]

## Logical organization

Programs are organized in modules, for example the main program and several libraries. However, this does not reflect how main and secondary memory in a computer system is organized. Therefore, it would be good to segment memory into several areas of variable size with the capability of specifying access permissions like read-only or execute-only to it.

### 2.2.2 Paging

According to [4], a widely used approach of managing memory in modern operating systems is *paging*. Processes are divided into *pages*, which are small, fixed-size pieces of equal size. Furthermore, the main memory is also divided into pieces of equal size called (page) frames and the pages are assigned to it. In addition, each virtual address is mapped to a physical address. The mapping of each process is maintained in an individual page table by the OS. Each virtual address consists of a page number and an offset within that page. Each time a virtual address is accessed in memory, the system needs to translate the virtual address first into a physical address by the help of the MMU.

The MMU also checks if the requested access is permitted. A system can additionally swap pages to secondary memory and relocate them afterwards. If mapping a physical address to a virtual address or any access checks fail, a page fault is raised.

The exact procedure of paging are platform dependent, but can be stylized this way: A virtual address is split into two parts: The most significant part of a virtual address forms a page number, while the least significant part marks an offset in the memory. When a virtual address is going to be translated, a page table entry (PTE) is first being looked up in the page table. This is done by adding the page number to the page table pointer, which is stored in a dedicated register. In particular, this means that the page number represents an offset in the page table. This results in a PTE containing access bits (Read, Write, Execute, etc.) and a frame number. The latter is prefixed to the offset of the virtual address to form a physical address.

Since having only a single page table would require a huge page table and therefore a lot of memory, engineers came up with multi-level paging. In case of a two-level paging system, the root page table contains pointers to a second level page table. Not every entry in a page table must be flagged valid. Instead, when they are flagged invalid, no second level page table has to be allocated, allowing to reduce memory costs.

In a two-level paging system, a virtual address consists of three parts: a memory offset and two page numbers or page table offset. First, the most significant part, which forms a page number, is combined with the root page table pointer in order to look up a PTE in the root page table. The result of the first page table lookup gives the address to a secondary page table, which is again combined with the next page number. This second lookup finally reveals the frame number, forming the physical address together with the memory offset of the virtual address.

However, looking up page tables has a big disadvantage: It requires memory accesses before the actually addressed data can be retrieved, because the page tables need to be walked. Therefore, many systems speed up address translation by using a TLB, which is a small cache for PTEs. On every translation, the page number is first looked up in the TLB, and in case of a TLB hit, the physical address can already be assembled. In contrast, in case of a TLB miss, the frame number needs to be retrieved from a page table stored in main memory.

## 2.3 RISC-V

### 2.3.1 An Open and Free Instruction Set Architecture

The *ParaNut* uses the RISC-V instruction set architecture (ISA) RV32IMA, part of the RISC-V ISA family. As stated in [5], this family was originally designed for computer architecture research and education. Over the years, RISC-V became an open and free standard and is not only used in different academical, but in several industry implementations, e.g. SweRV (EH1/EH2/EL2) by Western Digital[6][7][8] or several cores by SiFive[9], to just name a few from [1]. Primarily, RISC-V defines two base integer variants, RV32I and RV64I for 32-bit and 64-bit address space variants respectively. It further defines RV32I's subset variant RV32E for small microcontrollers and sketches a future RV128I variant for an 128-bit address space.[5]

The RISC-V families were designed to avoid specific implementation as much as possible, e.g. no microarchitecture style (in-order, decoupled, out-of-order, etc.) was defined, but each of them is allowed. It further defines several ISA extensions like the "M" extension for integer multiplication and division, "A" for atomic instructions or "F" for floating point operations. Besides these "standard" extensions, it also supports specialized and custom extensions. [5, p. 1f, 4f]

### 2.3.2 Privilege Modes

As specified in [3, p. 2ff], a RISC-V processor is always running in a privilege level encoded as a mode, all possibilities being displayed in table 1. Privilege levels protect different components of the software stack.

Two possibilities for software stacks are shown in fig 2. The white boxes represent an execution environment and the black boxes an abstract interface. On the left-hand side is an illustration of a simple system with a single application running on a application execution environment (AEE). The application uses an application binary interface (ABI) to interact with the AEE. The ABI hides details of AEE from

application, allowing greater flexibility on implementing the AEE. On the right-hand side, a configuration with a conventional OS is shown. It supports running multiple applications, each communicating with the OS over an ABI. Similarly, the OS communicates with a supervisor execution environment (SEE) (e.g. a simple bootloader or a BIOS-style IO system) via a supervisor binary interface (SBI).

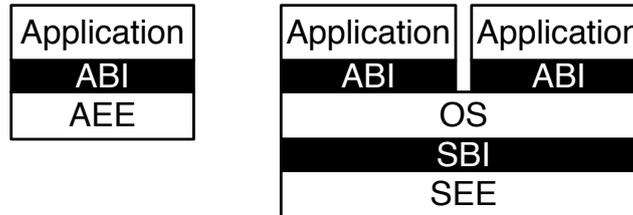


Figure 2: Different implementation stacks supporting various forms of privileged execution. [10, p. 2] (Removed Hypervisor execution context of original image)

This means, combined with the knowledge from section 2.2, that the different applications must be protected from interfering with each other or the OS. In practice, this can be achieved by providing memory protection with a paging system as described in section 2.2 or an optional physical memory protection unit, though the latter not being available for the *ParaNut* yet and also not being part of this project. Furthermore, some CSRs may not be accessible from each application context, as well as some instructions.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	<i>Reserved</i>	
3	11	Machine	M

Table 1: RISC-V privilege levels. [10, p. 3]

The highest level, machine-mode (M-mode), has full and low-level access on the machine, meaning all available peripherals, memory, CSRs or instructions can be accessed/executed. Therefore, it is mandatory on every RISC-V implementation. With user-mode (U-mode), a privilege mode with restricted machine access was added, providing an additional protection layer and enabling secure embedded systems. It may only access/execute instructions, CSR, memory or peripherals which are intended for it. To run Unix-like operating systems, supervisor-mode (S-mode) was introduced with dedicated CSRs and instructions for address translation and protection schemes. It is further able to access U-mode’s peripherals, memory, CSRs or instructions, but not M-mode’s.

Applications usually run in U-mode mode until some trap (exception or interrupt) occurs and makes the system jump into a trap handler by changing the current PC to `xtvec` and the current and storing it in `xeepc` ( $x$  represents the target mode in which the trap is handled). Traps are either horizontal (the trap is delegated to a handler in the same mode) or vertical (the trap is handled in a higher mode). See table 2 for an overview of all possible privilege mode combinations.

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

Table 2: Supported combinations of privilege modes.[10, p. 3]

### 2.3.3 Control and Status Registers

RISC-V implementations contain control and status registers (CSRs), which are read/modified/written atomically by specialized CSR instructions. They are destined for a specific privilege level, which can be identified by the first letter of its name, e.g. `mcause` belongs to M-mode, while `scause` belongs to S-mode. However, a CSR may also be accessed by higher privilege levels.

CSRs are identified by 12-bit wide addresses (`csr[11:0]`), leading to a maximum of 4,096 registers. The encoding of the addresses uses two conventions:

- The 2 most significant bits (`csr[11:10]`) encode the read-only (00, 01, or 10) or read/write(11) capabilities.
- The next 2 bits (`csr[9:8]`) encode the lowest privilege mode which is allowed to access the CSR.

Any access to a CSR which contradicts these two conventions results in an illegal instruction exception.

Though a basic set of CSRs was already implemented prior to this project, several standard CSRs from [10] and one non-standard register were required to be added in order to implement a MMU. In this chapter, all CSRs which were newly implemented or modified are listed and explained. Table 3 gives an overview of these registers. In the register's content listings below, fields which are either marked as reserved by [10] or not (yet) being implemented by the *ParaNut* are marked here as *Reserved*. Where not mentioned differently, *Reserved* fields are fixed to 0. If any field is written with bold letters, it means that the CSR existed before, but was extended with the

boldly written fields during this project. If none is written bold, the whole CSR was implemented from scratch. For each register, a brief summary of [10] and [3]<sup>1</sup> is given.

CSR Name	Change
<i>Standard Machine CSRs</i>	
<b>mstatus</b>	New fields introduced
<b>medeleg</b>	Completely new
<b>mideleg</b>	Completely new
<b>mcause</b>	New values introduced
<i>Standard Supervisor CSRs</i>	
<b>sstatus</b>	Completely new
<b>scause</b>	Completely new
<b>sscratch</b>	Completely new
<b>sepc</b>	Completely new
<b>stvec</b>	Completely new
<b>stval</b>	Completely new
<b>satp</b>	Completely new
<i>ParaNut-specific CSRs</i>	
<b>pnece</b>	Completely new

Table 3: Overview of the newly implemented or altered CSRs

### Machine Status Register (**mstatus**)

The **mstatus** register is used to keep track and control the current operating state of a core.

31	23	22	21	20	19	18	17	13	12	11
<i>Reserved</i>	<b>TSR</b>	<i>Reserved</i>	<b>TVM</b>	<i>Reserved</i>	<b>SUM</b>	<i>Reserved</i>	<b>MPP[1:0]</b>			
9	1	1	1	1	1	5	2			
10	9	8	7	6	5	4	3	2	1	0
<i>Reserved</i>	<b>SPP</b>	MPIE	<i>Reserved</i>	<b>SPIE</b>	<i>Reserved</i>	MIE	<i>Reserved</i>	<b>SIE</b>	<i>Reserved</i>	
2	1	1	1	1	1	1	1	1	1	1

Figure 3: Machine-mode status register (**mstatus**) of the *ParaNut*.

If **TSR** (Trap SRET) is set to 1, executing an SRET traps if not running in S-mode. If set to 0, executing an SRET in S-mode is permitted.

When the current mode is S-mode, modifying **satp** or executing a SFENCE.VMA instruction raises an illegal instruction exception if **TVM** (Trap Virtual Memory) is set to 1. If set to 0, virtual memory operations are permitted in S-mode.

<sup>1</sup>Will be updated soon after this paper's release.

Accessing a page marked to be accessible by U-mode traps if SUM (permit Supervisor User Memory access) is set to 0 and currently running in S-mode. If set to 1, S-mode is permitted to access user pages.

When a trap occurs in mode  $x$  and results in the new mode being  $y$ ,  $x$  is written into  $yPP$  and the current mode is set to  $y$ . Further,  $yPIE$  is set to the current value of  $yIE$  and  $yIE$  is set to 0.

If  $xIE$  is 0, external interrupts are disabled in mode  $x$ , or enabled if  $xIE$  is 1, respectively.

When a  $xRET$  is executed, the privilege mode is changed to the current value of  $xPP$ , and  $xPP$  is set to 0. Similarly,  $xIE$  is set to  $xPIE$ .

### Machine Trap Delegation Registers (`medeleg` and `mideleg`)

Usually, a trap is always handled in M-mode. However, a RISC-V system can be configured to delegate trap handling to S-mode. To do so, the bit at the position mentioned in table 4 must be set to 0 in `medeleg` for exceptions or `mideleg` for interrupts. Note that traps are only delegated when they occur in modes lower than M-mode. In M-mode, `medeleg` and `mideleg` are simply ignored.

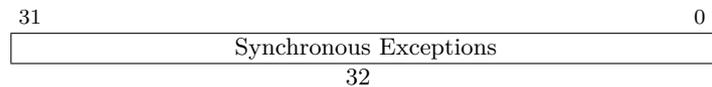


Figure 4: Machine Exception Delegation Register `medeleg`.

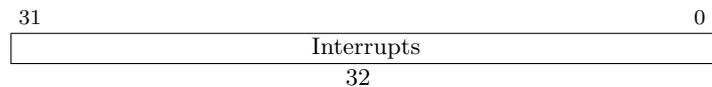


Figure 5: Machine Interrupt Delegation Register `mideleg`.

### Machine Cause Register (`mcause`)

When a trap is taken into machine mode, `mcause` is written with a value from table 4. Although this register was previously available, the boldly marked values from table 4 were newly implemented in this work.

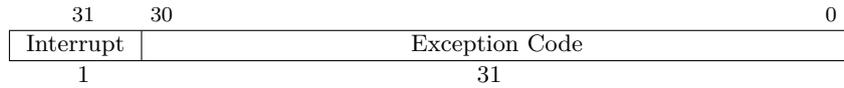


Figure 6: Machine Cause register `mcause`.

Interrupt	Exception Code	Description
1	0–7	<i>Reserved</i>
1	8	<b>User external interrupt</b>
1	9	<b>Supervisor external interrupt</b>
1	10	<i>Reserved</i>
1	11	Machine external interrupt
1	≥11	<i>Reserved</i>
0	0	Instruction address misaligned
0	1	<i>Reserved</i>
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	<i>Reserved</i>
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	<b>Environment call from U-mode</b>
0	9	<b>Environment call from S-mode</b>
0	10	Environment call from M-mode
0	11	<i>Reserved</i>
0	12	<b>Instruction page fault</b>
0	13	<b>Load page fault</b>
0	14	<i>Reserved</i>
0	15	<b>Store/AMO page fault</b>
0	≥16	<i>Reserved</i>

Table 4: Trap causes on the *ParaNut*. [10, p. 37, adapted for the *ParaNut*]

### Supervisor Status Register (`sstatus`)

The CSR `sstatus` represents a subset of `mstatus`, containing only the fields shown in fig 7. For details on the field’s meaning, refer to `mstatus`.

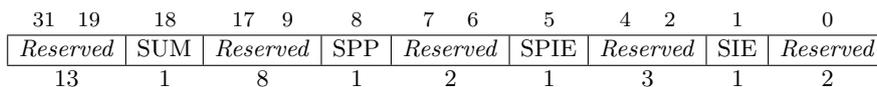


Figure 7: Supervisor-mode status register (`sstatus`) of the *ParaNut*.

### Supervisor Cause Register (`scause`)

When an trap occurs and the interrupt is delegated to S-mode, the cause of the trap is written into `scause` instead of `mcause`. Similarly to `mcause`, the values from table 4 may be written into `scause`, except of *machine external interrupt* and *Environment call from M-mode*.

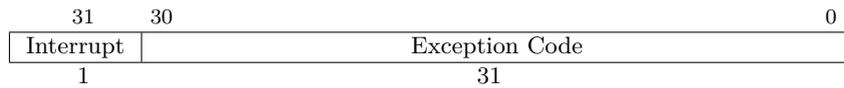


Figure 8: Supervisor Cause register `scause`.

### Supervisor Scratch Register (`sscratch`)

Usually, `sscratch` holds a pointer to the core-local supervisor context. On a trap, its content is swapped with a user register and acts as a initial working register.

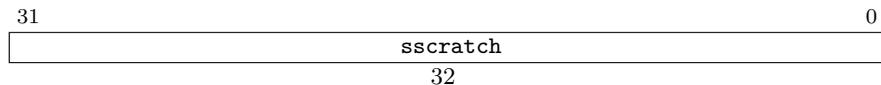


Figure 9: Supervisor Scratch Register.

### Supervisor Exception Program Counter (`sepc`)

When a trap is taken into S-mode, the current program counter is written into `sepc`. Furthermore, when a `SRET` is executed, the current value of `sepc` becomes the new PC.

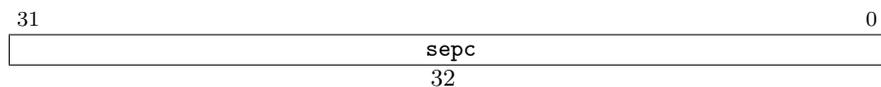


Figure 10: Supervisor exception program counter register.

### Supervisor Trap Vector Base Address Register (`stvec`)

When a trap is delegated to S-mode, the current program counter is set to the current value of `stvec`. The least significant two bits are always fixed to 0 to assure proper alignment to a 4 byte boundary.

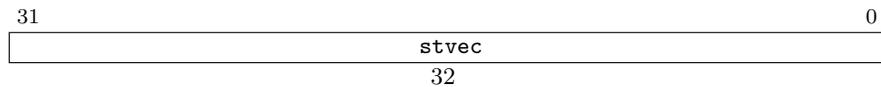


Figure 11: Supervisor trap vector base address register (**stvec**).

### Supervisor Trap Value (**stval**) Register

When a trap occurs, **stval** contains trap-specific information, helping the software to handle the trap. In particular, **stval** is written with the faulting virtual address on a hardware breakpoint, a faulty instruction-fetch, load or store address-misaligned, or a page-fault. On illegal instructions, **stval** is written with the faulting instruction.

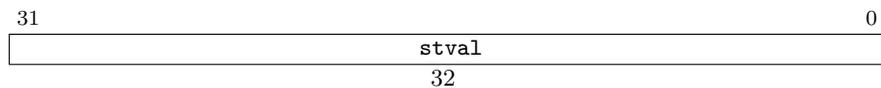


Figure 12: Supervisor Trap Value register.

### Supervisor Address Translation and Protection (**satp**) Register

A key CSR for the MMU is **satp**. It contains a pointer to the current root page table in the 20 bits wide field **PPN**<sup>1</sup> as well as an address translation enable bit in **MODE** field. When **MODE** is set to 0, virtual address translation is disabled, while setting it to 1 enables the translation scheme described in section 4.

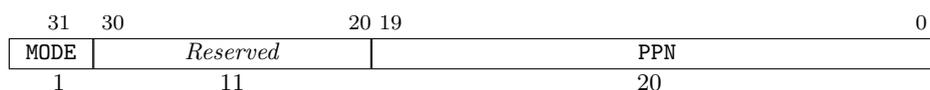


Figure 13: RV32 Supervisor address translation and protection register **satp**.

### **ParaNut Core Enable (pnce) and ParaNut Exception Core Enable (pnce) Register**

To enable coprocessing unit (CoPU)  $x$ , the central processing unit (CePU) sets the bit at position  $x$  in **pnce**. Similar, setting bit at position  $x + 1$  to 0 disables CoPU  $x$ . Bit 0 represents the CePU and must always be set.

<sup>1</sup>Although the field **PPN** is 22 bits wide in [10], which enables system bus widths of 34 bits, the *ParaNut* reserves only 20 bits for **PPN** due to its limited bus width of 32 bits.

When an exception occurs, the current `pnce` is written to `pnece` and `pnce` is set to 1, disabling all CoPUs.

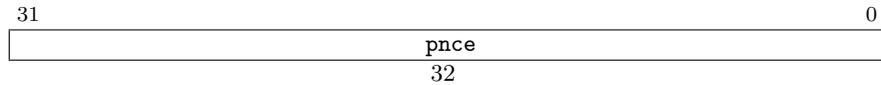


Figure 14: *ParaNut* Core Enable (`pnce`).

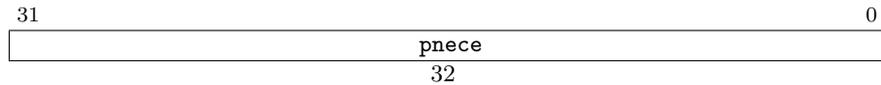


Figure 15: *ParaNut* Exception Core Enable (`pnece`).

### 2.3.4 "SYSTEM" Instructions

Implementing privilege modes together with a MMU and still complying with [10] requires adding some new instructions and updating some already existing. Table 5 lists these instructions, all of them being assigned to the instruction type *SYSTEM* by RISC-V.

The MRET instruction was already implemented prior to this work. However, since new privilege levels were introduced in this project, it was necessary to check the current privilege mode to be M-mode and raise an illegal instruction exception otherwise.

The SRET instruction has the exact same encoding as MRET with a single exception: the bit range [29:28] in SRET does not equal the encoding of M-mode but rather S-mode. Therefore, once the execution of an *xRET* was detected, checking that bit range [29:28] is at *x* or higher is required. Furthermore, SRET may only be executed in S-mode if the TSR bit in `mstatus` is not set (see `mstatus` in section 2.3.3); therefore, when executing in S-mode checking TSR is also required.

CSR modification instructions were already implemented on the *ParaNut*. For completeness, they are also listed in table 5, but refer to [5] for more detailed explanations on them. Since there were only M-mode CSRs available at the beginning of this work, privilege mode checks need to be implemented to make sure that they are not accessed from lower privileged levels. Furthermore, the *ParaNut* had a significant limitation which differed from RISC-V specification: Rather than raising an illegal instruction exception when accessing non-existent CSRs, the instruction completed successfully and returned a value of 0. This is also adjusted in this work.

Already implemented was the ECALL instruction, which causes a planned trap by software. At beginning of this work, only M-mode was available on the *ParaNut*,

ECALL made `mcause` always contain an *Environment Call from M-mode exception*. Therefore, when executing an ECALL, it is necessary to check for the current privilege mode first and set the exception cause to the appropriate mode in order to comply with the causes listed in table 4.

A completely new instruction is SFENCE.VMA. In its simplest form, it just flushes any caches related to address translation, i.e. the TLB of the *ParaNut*. In more sophisticated RISC-V implementations, `rs2` and `rs1` may be used to flush only particular entries meeting defined conditions. Though this is omitted in the *ParaNut* for simplicity.

Trap-Return Instructions												
31	25	24	20	19	15	14	12	11	7	6	0	
0001000	00010		00000		000		00000		1110011			SRET
0011000	00010		00000		000		00000		1110011			MRET
Supervisor Memory-Management Instructions												
31	25	24	20	19	15	14	12	11	7	6	0	
0001001	rs2		rs1		000		00000		1110011			SFENCE.VMA
Environment Call												
31	20	19	15	14	12	11	7	6	0			
000000000000			00000		000		00000		1110011			ECALL
CSR Instructions												
31	20	19	15	14	12	11	7	6	0			
csr	rs1		001		rd		1110011			CSRRW		
csr	rs1		010		rd		1110011			CSRRS		
csr	rs1		011		rd		1110011			CSRRC		
csr	uimm		101		rd		1110011			CSRRWI		
csr	uimm		110		rd		1110011			CSRRSI		
csr	uimm		111		rd		1110011			CSRRCI		

Table 5: New or altered RISC-V SYSTEM instructions of the *ParaNut*. Sources: [10, p. 76][5, p. 130] adapted to match the *ParaNut*

### 2.3.5 Virtual Address Translation

RISC-V defines several translation schemes, on the one hand to support different address space types (32-bit, 64-bit), on the other hand to provide a trade-off between the size of address space and minimizing address-translation cost. All schemes enable support for modern Unix-based operating systems by translate a virtual into physical addresses. To do so, the schemes involve traversing a page table made up by a radix-tree, which is explained in [11] as a tree with two types of node: leaf nodes, which store the values in correspondance to their keys, and inner nodes, mapping partial

keys to other nodes. On tree traversal, a portion of the key is used as an index into an array, which determines the next child node. For page tables, RISC-V calls leaf nodes *leaf (page table) entries*, and all the other nodes *pointer to (page table) entries*.

For 32-bit implementations like the *ParaNut*, only a single translation scheme exists, which is named *Sv32*. It divides the virtual address space into 4KiB pages and 4MiB superpages/megapages and is explained in section 4. Address translation for S-mode and U-mode can be enabled in the CSR `satp`. For M-mode, paging is always disabled.

A virtual address is structured as in figure 16, and is translated into a physical address, illustrated in fig. 17, by traversing a two-level page table, similar as explained in section 2.2. When translating, only the upper 20-bit forming a physical page number (VPN) are translated into 20-bit physical page number (PPN)<sup>1</sup>, the page offset remains untranslated.

A page table consists of  $2^{10}$  page table entries (PTEs), each 4 byte long, i.e. a page table is exactly 4KiB big and fits exactly into a page. A PTE (`pte[31:0]`) consists of the fields illustrated in fig. 18: Range `pte[31:10]` comprises two PPNs, which are used to form either a physical address or point to another page table. The PPN of the root page table is stored in `satp`. `pte[9:8]` is named RSW, which means "*reserved for software*" and should be ignored by implementations. `pte[7:0]` consists of the fields listed below.

- V: valid; if 0, all other bits are don't-cares and are freely usable by software
- R,W,X: read,write,execute permissions. When all three equal 0, this PTE is a pointer to the next level page table. Otherwise, it is a leaf PTE. It is obligatory that writable pages are also marked readable, any combination differing from this is reserved for future use by [10]. Table 6 lists all possible combinations and their meaning. If any memory access fails due to missing permission, a page fault exception is raised corresponding to its access type.
- U: page is accessible by U-mode or not. In case SUM in `mstatus/sstatus` is set, pages with the U bit set may also be accessed by S-mode.
- G: global mapping, i.e. the mapping exists in all address spaces. For non-leaf PTEs it means that all mappings in subsequent levels are global. The G bit is ignored by the *ParaNut* in order to keep the hardware simple, though it might reduce performance a bit.
- A (accessed) and D (dirty). RISC-V allows two schemes, of which currently this one is implemented in the *ParaNut*: Accesses when A is clear or writes

---

<sup>1</sup>Though Sv32 also supports translating 20-bit VPNs into 22-bit PPNs, this is not implemented on the *ParaNut*, as it does not support 34-bit physical addresses.

when D is clear raise a page fault according to the current access type. For the other scheme, refer to [10, p. 69]

The fields A, D, and U are currently only relevant for leaf PTE; for non-leaf PTEs, they are reserved for future use. As the fields R,W,X,U control the access types to their corresponding addresses, they are later on called access control bits. Additionally, the fields A,D are simply called status bits.

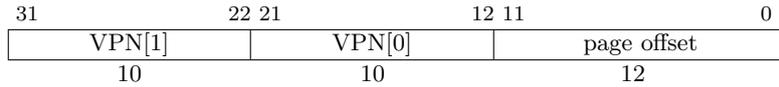


Figure 16: Virtual address[10, p. 68].

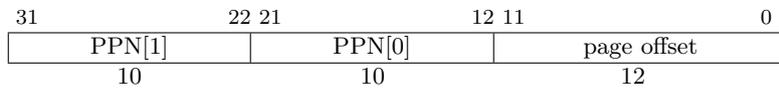


Figure 17: Physical address[10, p. 68]

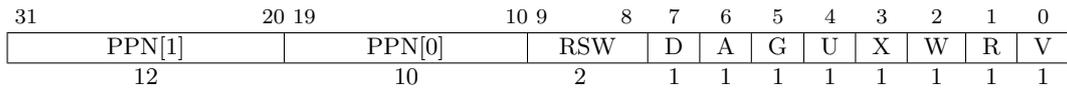


Figure 18: page table entry[10, p. 68].

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	<i>Reserved for future use.</i>
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	<i>Reserved for future use.</i>
1	1	1	Read-write-execute page.

Table 6: Encoding of the PTE's fields R,W,X fields. Source: [10, p. 68]

### 3 Implementing RISC-V's Privilege Modes

RISC-V's privilege modes are highly correlated to the control and status registers (CSRs) and are therefore strongly coupled to the CSR module.<sup>1</sup> An overview on how the privilege modes were implemented is given in this section.

As stated in section 2.3.2, there exist several privilege modes on RISC-V. On the *ParaNut*, the number of supported modes is configurable in the global *ParaNut* configuration file to the following options: 1 (only M-mode, only available mode prior to this work), 2 (M-mode and U-mode) or 3 modes (M-mode, S-mode, U-mode), as shown in section 2.3.2. The mode in which the privilege mode is currently running is stored in a register of two bits width<sup>2</sup> in the CSR module. In contrast to the CSRs, the privilege mode is a hidden register and not visible to software. The only way to change to a lower mode in software is by setting the MPP or SPP fields in `mstatus` and execute a MRET or SRET respectively. It is also possible to run into an exception or on interrupts, which will result in the same or a higher privilege mode. As the name CoPU intends, they are only meant to be used as coprocessors and will mostly be used in U-mode. However, the CoPU's registers can only be accessed by themselves. Thus, to enable saving the register's content on context switches into pages owned by S-mode's, they must be able to execute in S-mode as well. Therefore, the decision was made that they are always running in the same mode as the CePU.

By configuring more modes, more CSRs are added to the CePU. When S-mode is configured, the registers listed in fig. 3 in section 2.3.3 are added to the CSR module, adding the functionalities stated there. Whether a CSR is accessible in a specific mode is determined by comparing the bit field [9:8] of the CSR address, as shown in section 2.3.3, to the current value stored in the privilege mode register. If the address' bit field [9:8] is lower than the current mode, an illegal instruction exception is raised.

The same behavior applies to system return instructions (MRET and SRET): If field [29:28] in the instruction encoding is lower than the current privilege mode, an illegal instruction exception is raised. The URET instruction also raises an exception, because the *ParaNut* does not support user mode traps. When running in S-mode, an SRET also raises an illegal instruction exception if TSR is set

---

<sup>1</sup>CSR functionalities were moved from the ExU to a separate CSR module in order to reduce size and complexity of the ExU. However, they are still highly correlated to each other and many signals are routed between each other.

<sup>2</sup>[10] allows to use only a single bit if only two modes are implemented, but for simplicity the choice to do so is let to the synthesis tool.

When an exception occurs in S-mode and U-mode, the *ParaNut* checks on trap entry if the exception/interrupt is set in `mideleg/medeleg`, and if so, sets the new privilege mode to S-mode and fills S-mode's trap register, otherwise changes to M-mode and M-mode's trap registers are filled. If M-mode was the privilege mode when the trap occurred, the delegation registers are not checked, leading to the trap being handled in M-mode.

In addition, trap handling was also required to be adapted: Previously, all cores of the *ParaNut* finished their current instruction on traps and changed into a dedicated exception state. This state halted all CoPUs until the exception state was left, i.e. when executing a MRET/SRET/DRET instruction. However, when running software implementing context switches, this didn't allow to run the CoPUs in trap handlers, preventing the CoPUs from being able to save their current context and switch to another one.

In the new implementation, the exception state was omitted and CoPUs are simply halted instead. Furthermore, the state of the CSR `pnce` prior to the trap is stored in the register `pnce` (*ParaNut* exception core enable). When executing a MRET/SRET/DRET instruction, `pnce` is set to `pnce`, similar to the behavior of the privilege mode register and MPP described above.

## 4 Virtual Memory Model for the *ParaNut*

When designing the virtual memory model of the *ParaNut*, strong focus was set on supporting the *Linux kernel*. Hence, the *ParaNut* implements the *Sv32* model proposed in [10], since this is the memory model which was introduced in the official *Linux kernel* sources. While still remaining compatibility, the Sv32 model was slightly adjusted. Figure 19 illustrates the address translation process of virtual address  $va$  to physical address  $pa$ . To do so, the two virtual page numbers  $vpn[1 : 0]$  are translated to the physical page numbers  $ppn[1 : 0]$  by combining them with the physical page number  $ppn[1 : 0]$  and the fields read ( $r$ ), write ( $w$ ), execute ( $x$ ), user ( $u$ ) and valid ( $v$ ) in the page table entry  $pte$  as follows:

1. Get the address of the first  $pte$  by combining the root page table pointer stored in  $ppn$  in the CSR  $satp$  with the first virtual page number  $vpn$ :  $(satp.ppn \ll 12) + (va.vpn[1] \ll 2)$ .
2. If the  $pte$ 's fields  $pte.r = 0$  and  $pte.w = 1$ , or  $pte.v = 0$ , a page fault corresponding to the access type (read, write, execute) is raised.
3. If the  $pte$  is valid, check if  $pte.r = 1$  or  $pte.x = 1$  to determine if it is a megapage. If so, go to step 6. Otherwise, it is a pointer to another page table.
4. Get the secondary page table entry at the address  $(pte.ppn \ll 12) + (va.vpn[0] \ll 2)$ .
5. Check again if  $pte.r = 0$  and  $pte.w = 1$ , or  $pte.v = 0$  and raise a page fault corresponding to the access type (read, write, execute) if so. Otherwise, go on with the next step.
6. A leaf  $pte$  was found, which gives the mapping of  $va$  to  $pa$  depending on the following cases:
  - If superpage: Check for proper superpage alignment by comparing  $pte.ppn[0] = 0$ . Raise a page fault corresponding to the access type (read, write, execute) if it fails. Otherwise,  $pa = pte.ppn[1] \ll 22 + va.offset$
  - If page:  $pa = pte.ppn \ll 12 + va.offset$

Though Sv32 would allow any RISC-V implementation to translate a 32 bit wide virtual address into a 34 bit wide physical address, the *ParaNut* simply ignores the two most significant bits in the PTEs. Hence, 32 bit wide virtual addresses are always translated into 32 bit wide physical addresses, because the *ParaNut* uses an address bus of 32 bit length.

In order to perform address translation, the *ParaNut* introduces a MMU, which represents a collection of two modules. As visible in the light purple box in fig 20, it consists of the page table walker (PTW), described in section 5, and a translation lookaside buffer (TLB), explained in section 6.

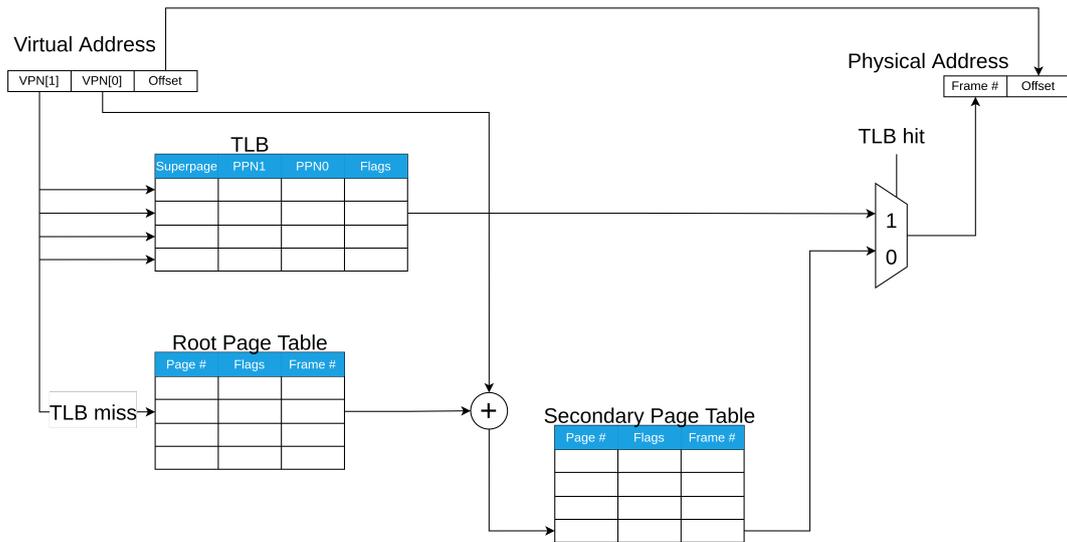


Figure 19: Illustration of the *ParaNut*'s address translation process for regular pages. For superpages, PTE lookup in secondary page table is omitted.

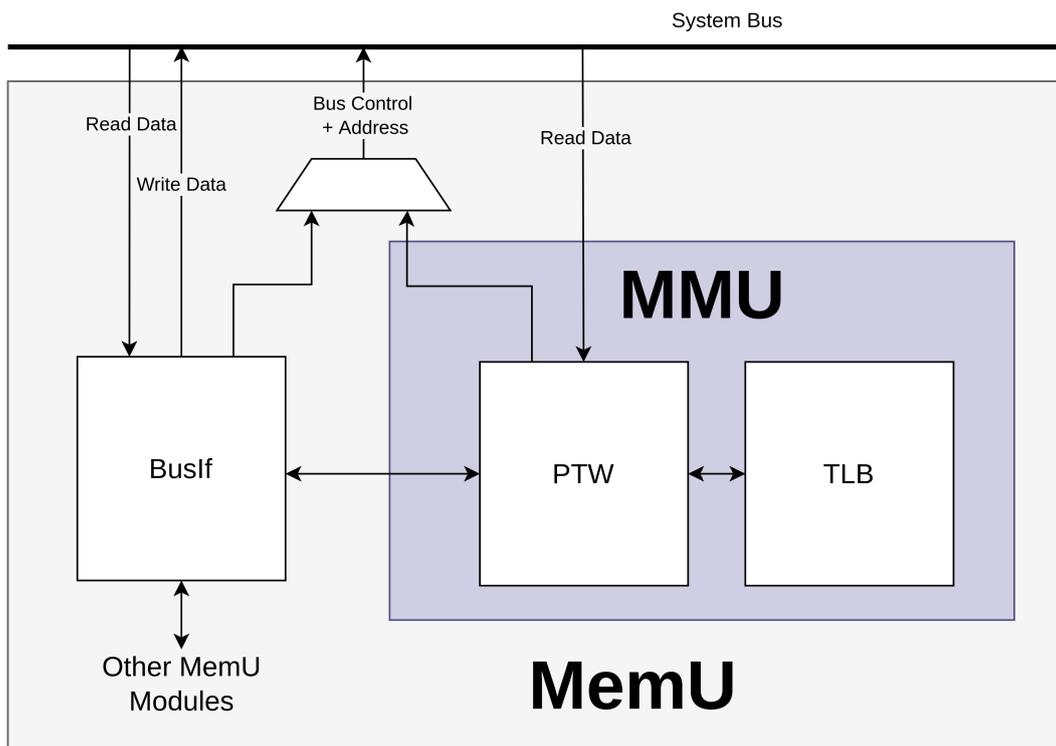


Figure 20: The MMU and their components

When paging is enabled by setting **MODE** in the **satp** CSR, the **BusIf** invokes the **PTW** to translate the address as soon as possible, and waits for it to complete the translation. Since step 1 and 4 in the listing above each require a memory access, the **PTW** implements a simple WishBone bus interface, only capable of reading

accesses. Simple multiplexers forward the bus signals of the PTW and the BusIf to the system bus. This is sufficient because the BusIf waits for the PTW to return a translated address before doing an access if paging is enabled. If paging is disabled or the translation completed, there is no need for the PTW to do any memory access. Hence, the hardware design guarantees that there will be no concurrent accesses by these two bus masters.

As already mentioned in section 4, reading one or two memory addresses before doing the actual memory request, is very expensive. Therefore, to speed up address translation, the *ParaNut* integrates a translation lookaside buffer (TLB) serving as a cache for PTEs inside the MMU. Before the PTW requests the first PTE from the BusIf, it sends a request to the TLB and waits for a hit or a miss. See section 6 for more details on the TLB.

Fig. 21 illustrates the flow of address translation: First, the TLB checks if it already stores the PTE. In case of a hit, all information required for superpage or page physical address generation are handed over to the PTW. On a TLB miss, the PTW reads the PTE from main memory and checks the flags as well as proper alignment. At this step, there exist three possibilities on how to move on:

1. Invalid flag combinations or invalid superpage alignment: A page fault is indicated to the BusIf. Any pending memory access is omitted.
2. On detected superpages: The superpage physical address is generated
3. If a pointer to the next level is found: A second memory access is performed, followed by another flag check. Again, if this check fails, the memory access is skipped, otherwise the page physical address is generated.

Once the PTW or the TLB succeeds with a hit, the requested virtual address translated to physical address is ready to be read or written from system bus by the BusIf.

To maintain good processing speed, the caches of the *ParaNut* are virtually addressed, which means that all cache tags are considered as virtual addresses as soon as the paging was enabled. When paging is disabled, all virtual addresses = physical addresses. This requires the tag RAM to store the access control flags Read, Write, Execute, User (RWXU) in the cache tag. When paging is disabled, RWXU bits are always filled with 1, which means that no page faults will occur. This performance improvement is incidental with the danger of inconsistencies on virtual address aliasing: When different virtual addresses reference the same physical address, they might be stored several times at different locations in cache. When one of them is altered, the other one is not.

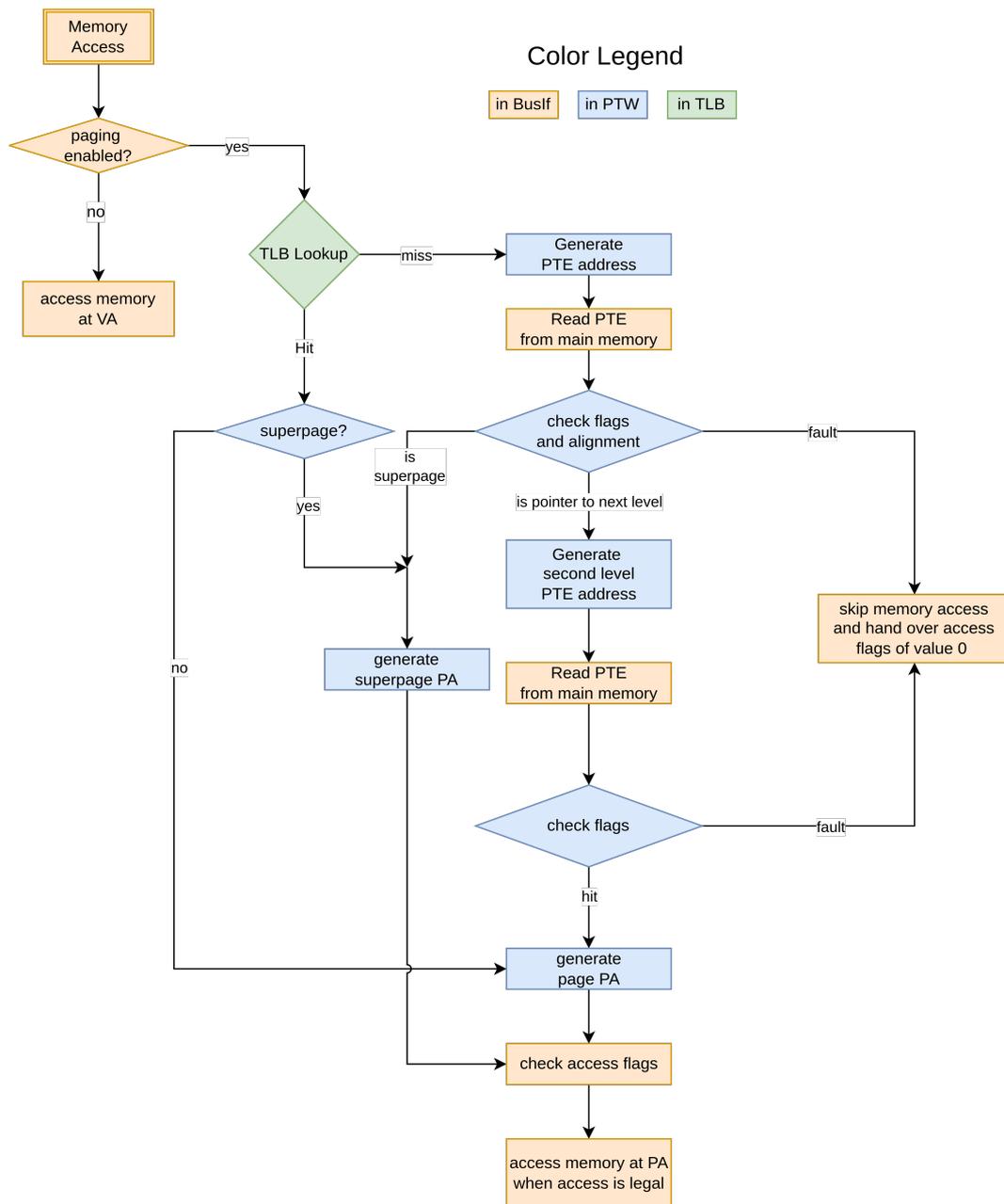


Figure 21: Abstract illustration of the address translation process.

The access control flags are always routed from the cache tag RAM or the BusIf via the read/write port to the LSU and the IFU as shown in fig. 22, and raise a page fault corresponding to their access type when set to 0 in the ExU.

On reads, the U bit is routed together with the data to the ExU, which then chooses to raise a page fault if set or not. However, on writes the data is forwarded the other way from the ExU as a sender to either the BusIf or the cache as a receiver. As a result, a signal is routed together with the data to the receiver, indicating if a page fault should be raised, depending on the status of the U bit. The receiver

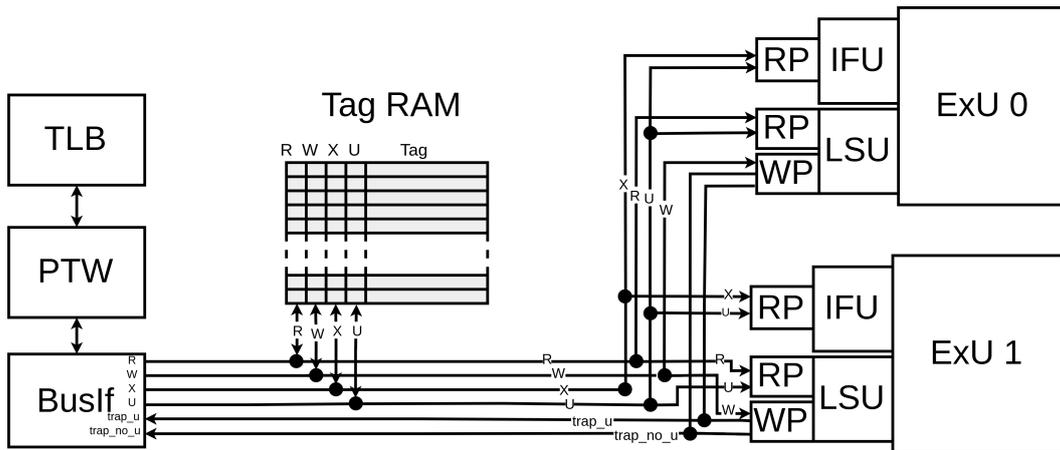


Figure 22: Overview of the *ParaNut*'s memory model

then decides (after translating the address in case of the BusIf) if the write will be performed or not and indicates a page fault by setting the W bit to 0 if applicable.

Since the IFU and the LSU have got a little instruction buffer and a little write buffer respectively, slight changes to them were required: The IFU pre-fetches instructions and stores them in an instruction buffer, therefore the U and X bit are also stored next to the encoded instruction data and handed over together to the ExU.

Furthermore, to guarantee that a store page fault is raised no later than the instruction which triggers the write, the LSU's write buffer accepts only one single item when paging is enabled. The write is not acknowledged to the ExU prior to the write being completed, in order to be able to guarantee access control to the ExU.

As described in 2.3.3, paging is only enabled when the MODE field in the `satp` CSR is 1. Furthermore, paging is always disabled in M-mode. Changes from U-mode or S-mode into M-mode and vice versa while `MODE = 1` therefore disable and enable paging respectively. Since this changes the addressing mode (virtual or physical) in the caches and buffers, they are flushed by hardware to avoid data loss or misbehavior<sup>1</sup>.

<sup>1</sup>This was also found while using the debugging module, as the debugging ROM code is always executed in M-mode.

## 5 Implementing a Page Table Walker

As soon as paging is enabled, each address which the ExU processes is considered a virtual address inside the MemU. Therefore, the BusIf can *not* simply write and fetch addresses from the system bus as before. Instead, the virtual addresses are translated to physical addresses inside the MMU by *walking the page tables*. This process is performed by a newly implemented and high-level-synthesizable SystemC module called page table walker (PTW).

As already explained in section 2.2, the PTW is coupled and invoked by the BusIf. All read and write ports route a paging signal to the BusIf to indicate if the requested address is to be considered a virtual address, i.e. if it must be converted from virtual address space to physical, or can already be considered to be physical. In case it is physical, the BusIf directly accesses the system bus, similar to before any project related changes were done. In contrast, if it is to be considered virtual, for direct writes and reads on the system bus, the BusIf starts the PTW and waits for it to acknowledge and return a physical address as well as the access control bits (R,W,X,U) and the status bits (A,D). In case of the access type not being permitted, the access to the resulting physical address is omitted and an access control bit of 0 is returned to the port. Otherwise, the data is read or written from the data bus and the corresponding access control bits are returned to the port.

If S-mode is disabled in the *ParaNut* hardware configuration, it is impossible to change the CSR `satp`, effectively setting all paging signals to 0. Thus, the PTW is never invoked, which allows the synthesis tool to remove it from the system and save chip area.

If an address is to be translated, the page tables need to be walked by the PTW. The process of translation, namely *Sv32*, was already explained in section 2.2, and is performed by a Moore machine. Since all PTEs are stored in main memory, the PTW implements a simple Wishbone bus interface, only capable of ‘Classic Cycle’ reading operations. That means that there is no indication to the slave about what is to be done in the next cycle by the bus master, as described in [12, p. 75]. The bus input emitted by the PTW is multiplexed as visualized in fig. 20 as soon as the PTW indicates a bus cycle.

To make sure that *ParaNut*’s BusIf is capable of working both with physical and virtual address spaces, it was slightly adjusted: Before project start, it consisted of a Moore-type machine, processing system bus (reads and writes) and cache data management (loading, invalidating and writing back) requests of the read and write ports. The requested bus address was stored in a single register, which was filled in the moment of the request. On direct reads and writes, this address was simply

read or written; on cache operations, the address was read and modified to read data from subsequent addresses. Furthermore, on cache writebacks the address was exchanged with the address stored in the cache tag.

In the new implementation, two registers are used: one for the virtual address, the other for physical addresses. Actual bus cycles are always performed with the address stored in the physical address register. However, the virtual address register is still required to indicate the ports which address was written.

When a port requests an operation from the bus interface, both registers are first filled with the same value. In the next states, the bus interface requests address translation from the PTW and stores the result in the physical address register. Furthermore, there are dedicated registers in the BusIf for the access control bits, one for each of R,W,X,U. Once the PTW successfully completes address translation, it returns the access control bits. The R and X bits are then ANDed with the A bit and the W bit is ANDed with both the A and D bit, in order to match A's and D's access scheme described in section 2.3.5.

For virtually addressed cache operations, when the cache is filled, the virtual address is first converted to a physical address, and the access control bits are stored together with the virtual address in the cache tag. Afterwards, the bank is filled with a bus burst operation on the incrementing physical address. Similar, when the cache is written back, the virtual address is read from the cache tag, translated to a physical address, and used for the incrementing cache bank write back. The access control bits are simply ignored, as access control is already guaranteed by the ports and the cache tag.

The process of address translation is performed as follows: When the BusIf leaves the idle state due to a write/read port request, it saves the routed paging signal in a register and changes the state as usual. In the following states, it decides as soon as possible if the address is to be translated and requests address translation from the PTW if required. Before the BusIf performs any request to the system bus, it waits for the PTW to successfully complete with a physical address and access control bits. A page fault is indicated by setting all access control bits to 0, otherwise the PTE's access control bits are forwarded. On read operations, the memory access is omitted when both R and X are 0. On write operations, W must be set or U must match the current active privilege mode. Read ports are then acknowledged with data of value zero and all the access control bits set to 0, write ports simply with the W bit set to 0. If a cache fill operation is to be performed, the cache is filled, invalidated or written back regardless of the access control bits' state. The ports accessing the cache make sure that that all data accesses are valid by reading the access control bits stored in the tag RAM.

## 6 Implementing a Translation Lookaside Buffer

Loading a leaf PTEs from main memory is very time-critical: A superpages requires at least one memory request, regular pages even two. Many modern processors with a MMU therefore include a cache for PTEs called translation lookaside buffer (TLB), and so does the *ParaNut*.

It is a high-level-synthesizable SystemC implementation of a fully-associative cache for leaf PTEs with a latency of 1 clock cycle, i.e. the result is available after a single clock cycle. Similar to the PTW, the TLB is never invoked if no supervisor mode is configured, thus the synthesis tool may remove the TLB from the hardware and save chip area.

Furthermore, there exist two parameters the TLB: First, it is possible to disable it completely to save chip area though S-mode is enabled. Second, the number of entries is configurable to a number to the power of two, though the size is currently limited to 8 entries.<sup>1</sup> Each entry is capable of storing either a page descriptor or a superpage descriptor and contains:

1. a valid bit,
2. a tag,
3. a superpage bit,
4. status bits: A, D,
5. access control bits: R, W, X, U,
6. and a PPN.

For simplicity, above items 1-3 are later simply called *TLB Tag* and items 4-6 *TLB Data*.

The superpage bit is required to determine if only the most significant half of the tag address is to be compared, or also the lower. This is required because there is no way to differentiate physical addresses of pages with the lower ten bits set to zero by pure chance from superpages, where the lower 10 bits are always set to 0 (see section 2.3.5 and 2.2 for more details on pages and superpages).

To determine where a new entry is to be stored, a tree-based Pseudo Least Recently Used (PLRU) strategy, described by Abel in [13, p. 82f], was implemented, which approximates a Least Recently Used (LRU) strategy but with lower hardware requirements. Each node is represented by a single bit which indicates the direction of the search. The total number of bits required is  $x - 1$ , where  $x$  is the number

---

<sup>1</sup>When Vivado is used to high-level-synthesize the SystemC module with more than 8 TLB entries, Vivado crashes the developers machine by filling the RAM completely.

of entries. Fig. 23 illustrates a PLRU tree for a cache with eight entries. As it is visible, the tree currently points to *Entry<sub>3</sub>*. The binary tree is not updated on reads as long as it is not completely full in order to make sure that every entry is filled. As soon as each entry is valid, both reads and writes update the tree by setting each bit on the path to point away from the accessed entry. If a write were to be performed and the binary tree were like in fig. 23, *Entry<sub>3</sub>* would be written and bits 0, 1 and 4 would be flipped, which makes the tree point to *Entry<sub>6</sub>* afterwards.

When a new entry is to be written into the TLB, it does not check if the new address is already available or not. This is not required, as the PTW does not fill the TLB with new entries after TLB hits, but only after TLB misses.

In the TLB implementation, the PLRU register is written a single clock cycle later than the resulting data is available, as this decreases the longest path significantly and allows higher system clock rates. To do so, the TLB stores the ID of the accessed entry in a dedicated register, as well as another register used to indicate that the PLRU is to be updated.

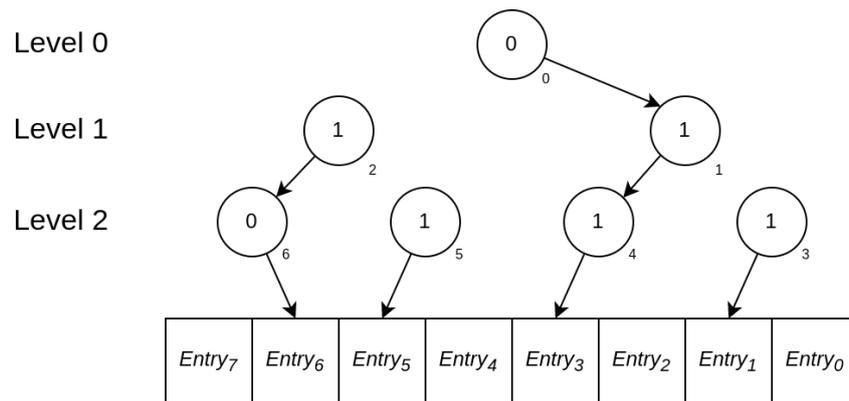


Figure 23: Illustration of a tree-based PLRU for a cache with eight entries.

Originally, it was intended to store the *TLB Data* in a Block RAM (BRAM) and only the *TLB Tag* in (flip-flop) registers. However, High-Level-Synthesis (HLS) did not allow the BRAM to be accessible immediately. Instead, it added two more clocks delay, resulting in a access latency of 3 cycles, which is very slow compared to other modern implementations of TLBs, e.g. the Inter Core i7 6700 with an access latency of 1 as stated in [14, p. 134]. Therefore, the *TLB Data* is also stored in (flip-flop) registers like the *TLB Tag*.

## 7 Validation

While the project was progressing, many changes were made in different components of the *ParaNut*. Therefore, it was necessary to ensure that all prior features of the *ParaNut* still work and no functionalities are broken. Furthermore, all new features need to be validated. To do so, some tools were developed, and also some other projects were used to do so: A test bench was developed for the TLB, as well as a little MMU / privilege mode testing program. Additionally, the official Linux kernel was compiled and ran in the *ParaNut* simulator and on hardware, and successfully printed approx. 30 lines of the boot output.

### 7.1 TLB Test Bench

To validate the TLB, which was implemented in SystemC, the test bench was also developed in SystemC. At the very beginning, the test bench creates a list of test cases with *pseudo-random* virtual address, physical address, page table bits and the superpage bit. Then the actual validation is done: The TLB is reset, and for each address, the following tests are performed:

At first, it is checked if requesting the virtual addresses leads to the expected miss. Afterwards, the TLB is filled with this virtual addresses, physical addresses, the page table bits and the superpage bit. Next, the virtual addresses is requested again, but this time a hit is expected and the resulted physical addresses, page table bits and the superpage bit are checked if they match the previously written ones. Then it is checked if all previously inserted addresses are still available, with respect to the maximum available entry count.

Once all test cases were written and checked, the TLB is flushed and the operation is repeated in order to validate the successful flush.

### 7.2 MMU Test program

In order to test the MMU in total, an in-system test bench in the form of a RISC-V program was developed for the *ParaNut*. Basically, it validates two features:

1. Privilege mode: The tool performs different instructions requiring different privilege levels, which should invoke exceptions under certain conditions. Furthermore, it tests interrupt delegation with M-mode's hardware timer `mtimer`.

2. Virtual memory: A set of page tables are created and filled to allow load and store operations on the same physical addresses from different virtual addresses.

To validate the privilege mode's functionality, there exist two arrays, one which was manually filled with the expected results in form of cause IDs, separate for each privilege mode; the other one is initialized at execution time with zeroes (as this exception code can never occur at execution time). When a trap occurs, the entry of the test case is filled with the trap ID.

As the *ParaNut* can be configured at compile/synthesis time, the results slightly differ depending on the selected configuration. First of all, the privilege modes which are not included by the configuration can be omitted. Second, some CSR (fields) do not exist and can therefore be omitted, too. Therefore, the test program determines the privilege levels which are currently configured into the *ParaNut* by writing and immediately reading the MPP of `mstatus`. Afterwards, the array with the expected results is adjusted.

The privilege mode tests which are executed are:

1. Reading non-existent CSR: should fail always
2. Writing non-existent CSR: should fail always
3. Reading M-Mode RO CSR: only allowed in M-mode
4. Writing M-Mode RO CSR: never allowed
5. Reading M-Mode RW CSR: only allowed in M-mode
6. Writing M-Mode RW CSR: only allowed in M-mode
7. Reading S-Mode RW CSR<sup>1</sup>: only allowed in S-mode and M-mode
8. Writing S-Mode RW CSR: only allowed in S-mode and M-mode
9. Reading U-Mode RO CSR: always allowed
10. Writing U-Mode RO CSR: never allowed
11. Reading U-Mode RW CSR: always allowed
12. Writing U-Mode RW CSR: always allowed
13. Executing a MRET: only allowed in M-mode
14. Executing a SRET with TSR=0: allowed in S-mode and M-mode
15. Executing a SRET with TSR=1: only allowed in S-mode
16. Executing an URET: never allowed because the N-Extension is not implemented
17. Receiving an interrupt from the `mtimer` with MIE=0 and SIE=0: interrupt occurs only in U-mode

---

<sup>1</sup>Note that reading and writing S-mode RO registers are omitted because the *ParaNut* does not implement any.

18. Receiving an interrupt from the `mtimer` with `MIE=0` and `SIE=1`: interrupt occurs in U-mode and S-mode
19. Receiving an interrupt from the `mtimer` with `MIE=1` and `SIE=0`: interrupt occurs in M-mode and U-mode
20. Receiving an interrupt from the `mtimer` with `MIE=1` and `SIE=1`: interrupt occurs always
21. Receive an interrupt and set it to be delegated to S-mode with `MIE=0` and `SIE=0`: interrupt only in U-mode and handled by S-mode
22. Receive an interrupt and set it to be delegated to S-mode with `MIE=0` and `SIE=1`: interrupt handled in S-mode, but in M-mode no interrupt
23. Receive an interrupt and set it to be delegated to S-mode with `MIE=1` and `SIE=0`: no interrupt in S-mode, in U-mode handled by S-mode
24. Receive an interrupt and set it to be delegated to S-mode with `MIE=1` and `SIE=1`: handled by M-mode if in M-mode, otherwise handled in S-mode
25. ECALL from current Mode: raises an ECALL from current mode exception

To test the MMU, three separate arrays are declared, which were aligned to a boundary of  $2^{12}$  with GCC variables attributes (see [15]) and initially filled with values of 0 to generate invalid PTEs. One array is meant as the root page table, the other two as second level page tables. Some entries are then filled to make the completely different virtual addresses `0x42344000` and `0x87354000` point to the exact same page frame with a size of 4 KB; Furthermore, three different superpage entries are generated in the root page table so that the stack, `.text` and `.data` segment's virtual addresses equals their physical address.

Then `satp.MODE` is set to 1 and `satp.PPN` is set to point to the root page table. Afterwards, the paging is activated by changing into U-mode. The paging example code then simply fills the page frame with data and compares if both addresses point to the same data when both virtual addresses do share the same page offset. The validation succeeds if both arrays do have the completely same content though their addresses are completely different, as their physical addresses are in fact similar.

### 7.3 Booting Linux

Writing testbenches and validation programs does only provide a simulated way to validate the reliability of the *ParaNut's* MMU. Misunderstanding and misinterpretation of the standard [10] led to errors which were introduced in the hardware as well as the software. Therefore, it was inevitable to use a sophisticated program which makes use of the *Sv32* address translation scheme introduced in [10] and which was developed by a third party: the Linux kernel.

This provided many benefits: It is a sophisticated program with sophisticated paging facilities and was already validated on other RISC-V processors. Besides that, it is a professionally used software without contrived test cases. In conjunction with [16], many bugs were found:

- Wrong combinations of physical page numbers and with virtual page numbers or offsets, thus resulting in invalid physical addresses.
- Exception flag: the *ParaNut* set a flag when it entered a trap handler and resetting it when leaving the trap handler with a trap return instruction. This was used to stall the CoPUs while the CePU handles the trap. However, The Linux provokes a page fault to change the virtual addresses, but does not leave it with a trap return instruction. This flag was removed and the CoPUs now halt completely on traps once their enable and mode status was stored in separate CSRs.
- The cache was not flushed when the privilege mode was changed from S-mode to M-mode and vice versa while paging was enabled. This led to an invalid mix of virtual and physical addresses. The CSR module was changed to detect this condition on trap or debug entry as well as on trap and debug return instructions and flush the cache if required.
- Some timing errors were investigated, including the PTW bus output address changing too early.
- When paging was enabled, the LSU's write buffer was disabled completely without flushing it.
- Similar, the access control bit R was not transmitted from the read port correctly in the LSU and IFU.

After all mentioned bugs were fixed, the *ParaNut* was able to print about 30 lines of Linux' boot output in the simulator, and a reference design was created for the Zybo Z7[17] to run it on FPGA hardware as well. More details about Linux on the *ParaNut* are available in [16].

## 8 Conclusion

### 8.1 Summary

The results of this project show, how a fully functional RISC-V compatible MMU was developed and integrated into the RISC-V compatible processor *ParaNut*. In this particular case, the MMU was implemented with two modules implemented in SystemC: The first being a PTW, which translates virtual addresses to physical addresses completely in hardware by accessing PTEs stored in page tables located inside the main memory, and second a TLB, which is used as a cache for PTEs and helps to reduce the number of memory accesses, hence speeding up the address translation. The latter is, similar to the *ParaNut*'s regular cache, configurable in its size and can be disabled completely to reduce the required chip area at synthesis.

Furthermore, privilege modes (machine, supervisor and user) were implemented which enable software to provide different layers of abstraction, thus allowing to embed further security in the software. At synthesis time, the number of available privilege modes is configurable to either one, two or three modes, where only the presence of supervisor mode makes the MMU available in hardware, hence allowing to save chip area if not required. If configured, virtual addressing is enabled by writing into a CSR, which sets the regular cache of the *ParaNut* to be virtually addressed to provide a reasonable performance.

All results of this work were tested and validated with simple and fully functional RISC-V bare-metal applications or test benches. Furthermore, the capabilities of the implemented design were proved when the Linux kernel was booted in the simulator and on FPGA hardware, having the kernel print about 30 lines of boot output.

### 8.2 Future work

The presence of the MMU enables a full set of new projects: With the ability to use virtual address spaces and translating them, a typical requirement of operating systems is provided. Therefore, focus can be set on various RISC-V compatible operating systems, especially on improving the Linux support.

Also, the current implementation of the MMU can be optimized for smaller area or timing requirements, e.g. the TLB may be optimized to use BRAM to store PTEs instead of flip-flop registers.

Furthermore, thanks to the *ParaNut* being publicly available with a permissive license, this implementation can serve as a reference implementation for other RISC-V cores.

## References

- [1] RISC-V International. (2022). RISC-V Exchange, RISC-V International, [Online]. Available: <https://riscv.org/exchange/> (visited on 06/23/2022).
- [2] A. Bahle, G. Kiefer, A. K. Pfütznner, and L. Vollbracht, “The ParaNut/RISC-V Processor - An Open, Parallel, and Highly Scalable Processor Architecture for FPGA-based Systems”, *Embedded World*, 2021.
- [3] G. Kiefer, A. Bahle, C. H. Meyer, and F. Wagner, “The ParaNut Processor: Architecture Description and Reference Manual”, vol. 1.0-0-gd3eb3c6\*, in collaboration with M. Schäferling, A. Pfütznner, and P. Zacharias, Nov. 22, 2021.
- [4] W. Stallings, *Operating Systems: Internals and Design Principles*, 7th ed. Boston: Prentice Hall, 2012, ISBN: 978-0-13-230998-1.
- [5] A. Waterman and K. Asanovic, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA”, RISC-V Foundation, Document Version 20191213, Dec. 2019.
- [6] Western Digital, *EH1 RISC-V SweRV Core 1.9*, CHIPS Alliance. [Online]. Available: <https://github.com/chipsalliance/Cores-SweRV> (visited on 07/01/2022).
- [7] Western Digital, *EH2 SweRV RISC-V Core 1.4*, CHIPS Alliance. [Online]. Available: <https://github.com/chipsalliance/Cores-SweRV-EH2> (visited on 07/01/2022).
- [8] Western Digital, *EL2 SweRV RISC-V Core 1.4*, CHIPS Alliance. [Online]. Available: <https://github.com/chipsalliance/Cores-SweRV-EL2> (visited on 07/01/2022).
- [9] SiFive, Inc. (2022). RISC-V Core IP, SiFive, [Online]. Available: <https://www.sifive.com/risc-v-core-ip> (visited on 06/01/2022).
- [10] A. Waterman and K. Asanovic, “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture”, RISC-V Foundation, Document Version 20190608-Priv-MSU-Ratified, Jun. 2019.
- [11] V. Leis, A. Kemper, and T. Neumann, “The adaptive radix tree: ARTful indexing for main-memory databases”, in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, Brisbane, QLD: IEEE, Apr. 2013, pp. 38–49, ISBN: 978-1-4673-4910-9 978-1-4673-4909-3 978-1-4673-4908-6.
- [12] R. Herveille, “WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores”, *OpenCores Organization*, vol. Revision: B.3, Sep. 7, 2002.

- [13] A. Abel, “Automatic Generation of Models of Microarchitectures”, PhD thesis, Saarland University, 2020. [Online]. Available: <https://publikationen.sulb.uni-saarland.de/bitstream/20.500.11880/29336/1/thesis.pdf>.
- [14] J. L. Hennessy and Patterson, David A., *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., Dec. 2017, ISBN: 978-0-12-383872-8.
- [15] Free Software Foundation, Inc. (2022). GCC, the GNU Compiler Collection, Common Variable Attributes, [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html> (visited on 05/22/2022).
- [16] N. Borgsmüller, *Linux for the ParaNut Processor*. Published with this work, 2022.
- [17] Digilent, Inc. (2022). Zybo Z7 - Digilent Reference, Zybo Z7, [Online]. Available: <https://digilent.com/reference/programmable-logic/zybo-z7/start> (visited on 06/22/2022).