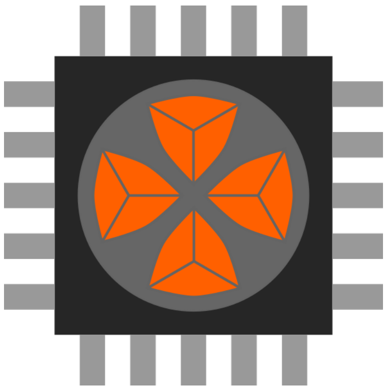
Hochschule
Augsburg University of
Applied Sciences

The *ParaNut* Processor

Architecture Description and Reference Manual



Gundolf Kiefer, Alexander Bahle, Christian H. Meyer,
Felix Wagner

Hochschule Augsburg – University of Applied Sciences

gundolf.kiefer@hs-augsburg.de

With contributions by:
Michael Schäferling, Anna Pfützner, Patrick Zacharias

Version: v1.0-0-gd3eb3c6*

November 22, 2021

Document History

Version	Date	Description
0.2.0	2015-02-19	Initial public release
0.2.1	2015-12-16	Add local CPU identification register, LL/SC instructions
0.3.0	2018-12-01	Change to RISC-V ISA
0.3.1	2020-11-16	Minor improvements
0.4.0	2020-02-08	Add User and Supervisor modes
0.4.1	2020-11-16	Minor improvements
0.4.2	2021-05-26	Minor improvements
1.0.0	2021-11-22	Major rework of the manual, avoiding duplication of general RISC-V information; Switch to Git-based versioning

Contents

1. Introduction

The goal of the *ParaNut* project is to develop an open, scalable and practically applicable multi-core processor architecture for embedded systems. Scalability is given by supporting parallelism at thread and data level based on multiple processing cores while keeping the design of the individual core itself as simple as possible.

ParaNut introduces a unique concept for SIMD (single instruction, multiple data) vectorization. Whereas SIMD extensions for workstation processors or embedded systems frequently contain specialized instructions leading to an inherently bad compiler support, SIMD code for the *ParaNut* can be programmed in a high-level language according to a paradigm very similar to thread programming.

The instruction set is kept compatible to the RISC-V specification. Hence, the RISC-V GCC tool chain and libraries/operation systems (newlib, Linux in the future with some necessary extensions) can be used with the *ParaNut* .

To date, the *ParaNut* project is still work in progress, and new contributors from industry and academia are welcome. An informal project overview including the implementation status and very promising benchmark results can be found in [1].

2. The *ParaNut* Architecture

2.1. Instruction Set Architecture

The *ParaNut* instruction set architecture is compatible with the RISC-V specification. The RISC-V architecture is an open source load and store RISC architecture designed with the purpose to support a wide spectrum of different chips from small microcontrollers to server CPUs. [2]. Scalability is achieved by defining a minimalistic basic instruction set (RV32I) together with optional extensions including a floating-point unit (FPU) or a memory management unit (MMU). Furthermore, the basic architecture offers configuration options such as different register file sizes or optional arithmetic instructions.

ParaNut processors implement all mandatory instructions according to the RV32I specification. Features unique to *ParaNut* require some additional *ParaNut* -specific instructions. These will be encapsulated in a small support library, so that they are still usable without compiler modifications. For software development, the GCC tool chain from the RISC-V project can be used without any modifications. A cycle-accurate SystemC model can be used as an instructions set simulator. To date, an operating environment based on the "newlib" C library allows to compile and run software both in the simulator and on real hardware.

2.2. Structural Organisation

The general structure of *ParaNut* is depicted in Figure 2.1. The core contains one *Central Processing Unit (CePU)* and a number of *Co-Processing Units (CoPU)*. The CePU is a full-featured CPU, whereas the CoPUs are CPUs with a more or less reduced functionality and complexity. Depending on the mode of execution (see below), the CoPUs may either be inactive (sequential code), execute a part of a vector operation, or execute a thread. In the sequel, the term CPU refers to any of a CePU or a CoPU.

All the CPUs are connected to a central *Memory Unit (MemU)*. The MemU contains the cache(s) and means to support synchronisation primitives. It provides a single bus interface to the main system bus, and independent read and write ports for each CPU. It is optimized to support parallel accesses by different CPUs. In particular, multiple read accesses to the same address can be served in parallel and run no slower than a single access, and accesses to neighboring addresses can mostly be served in parallel. These two properties are particularly important for the SIMD-like mode.

Each CPU contains an ALU, a register file and some control logic which together form the *Execution Unit (ExU)*. The *Instruction Fetch Unit (IFU)* is responsible for fetching instructions from the memory subsystem and contains a small buffer for prefetching instructions. The *Load-Store Unit (LSU)* is responsible for performing the data memory accesses of load and store operations. It contains a small store buffer and implements write combining and store forwarding mechanisms as well as mechanisms to support atomic op-

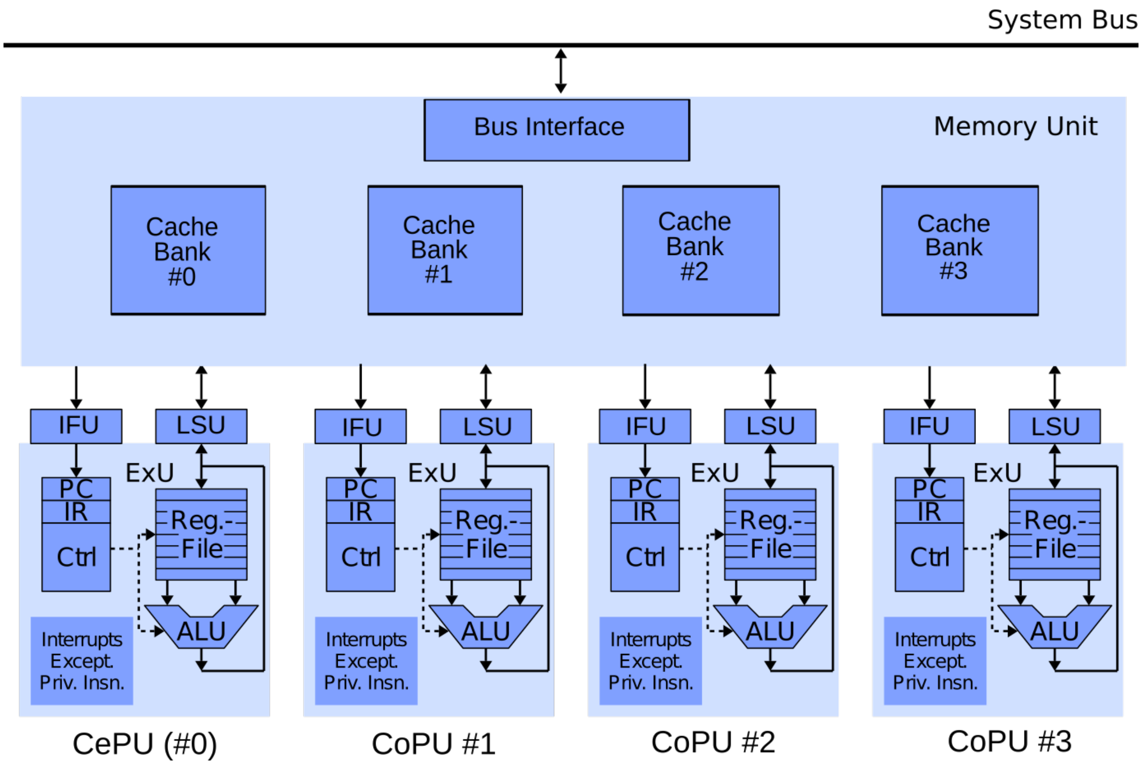


Figure 2.1.: A *ParaNut* instance with 4 cores

erations.

The Execution Unit is designed and optimized for a best-case throughput of one instruction in two clock cycles (CPI≈2, CPI = "clocks per instruction"). This is slower than modern pipeline designs targeting a best-case CPI value of 1. However, it allows to better optimize the execution unit for area, since no pipeline registers or extra components for the detection and resolution of pipeline conflicts are required. Furthermore, in a multi-core system, the performance is likely to be limited by bus and memory contention effects anyway, so that an *average* CPI value of 1 is expected to be hardly achievable in practice. In the *ParaNut* design, several measures help to maintain an average-case throughput very close to the best-case value of CPI≈2, even for multi-core implementations.

The design of the memory interface and cache organization is very critical for the scalability of many-core systems. In a *ParaNut* system, the Memory Unit (MemU) contains the cache, the system bus interface, and a multitude of read and write ports for the processor cores. Each core is connected to the MemU by two independent read ports for instructions and data and one write port for data. The cache memory logically operates as a shared cache for all cores and is organized in independent banks with switchable paths from each bank to each read and write port. Tag data is replicated to allow arbitrary concurrent lookups. Parallel cache data accesses by different ports can be performed concurrently if their addresses a) map to different banks or b) map to the same memory word in the same bank. Furthermore, by using dual-ported Block-RAM cells, each bank can be equipped with two ports, so that up to two conflicting accesses (i.e. same bank, different addresses) are possible in parallel. Hence, even for many cores, the likelihood of contention can be arbitrarily reduced by increasing the number of banks, which is configurable at synthesis time.

The cache can be configured to be 1/2/4-way set associative with configurable replacement strategies (e.g. pseudo-random or least-recently used). The Memory Unit implements mechanisms for uncached memory accesses (e.g. for I/O ports) and support for atomic operations. All transactions to and from the system bus are handled by a bus interface unit, which presently supports the Wishbone bus standard, but can easily be replaced to support other busses such as AXI.

2.3. Execution Modes and Capabilities

A CPU in the *ParaNut* architecture can run in 4 different modes:

Mode 0 (Halted): The CPU is inactive.

Mode 1 (Linked): The CPU does not fetch instructions, but executes the instruction stream fetched by the CPU.

Mode 2 (Unlinked): The CPU fetches and executes its own instructions. Exceptions trigger an exception of the controlling CePU and put this CPU into Mode 0. The CePU can later put this CPU into Mode 2 again, and the code execution continues as if the exception has been handled by this CPU.

Mode 3 (Autonomous): The CPU executes its own instructions. Exceptions and interrupts can be handled by this CPU.

Typically, the CePU always runs in Mode 3. The mode of the CoPUs is controlled by the CePU. Depending on the application, the CoPUs can be customized that they only support a subset of the 4 modes. For example, if only SIMD vectorization and no multi-threading is required, all the logic required for modes 2 and 3 can be stripped off. Now, the CoPU does not require much more area than a vector slice of a normal SIMD unit would. In general, a CoPU is customized for a *capability level* of m, meaning that all modes $\leq m$ are supported.

- A Capability-1-CoPU only contains very little logic besides the ALU and the register file. Hence, a *ParaNut* with only Capability-1-CoPUs does not require much more area than a normal SIMD processor.
- A Capability-2-CoPU additionally contains an instruction fetch unit and eventually one more read port to the Memory Unit (MemU) for it.
- A Capability-3-CoPU is basically a full-featured CePU. It contains logic to handle interrupts and exceptions and has its own set of special registers. This is not needed for multi-threading, but for multi-processing, where each CoPU is managed by the operating system as an individual CPU.

A CPU with Capability $\geq$2 in Mode 0 will reset its IFU. Upon changing to Mode 2 or higher the CPU starts executing at the reset vector address. This enables control of Mode 2 CoPUs through software. Figure 2.2 illustrates the active/required hardware for the 4 modes. The following sections briefly illustrate how SIMD vectorization or multi-threading can be performed. Further informal explanations and examples can be found in [1].

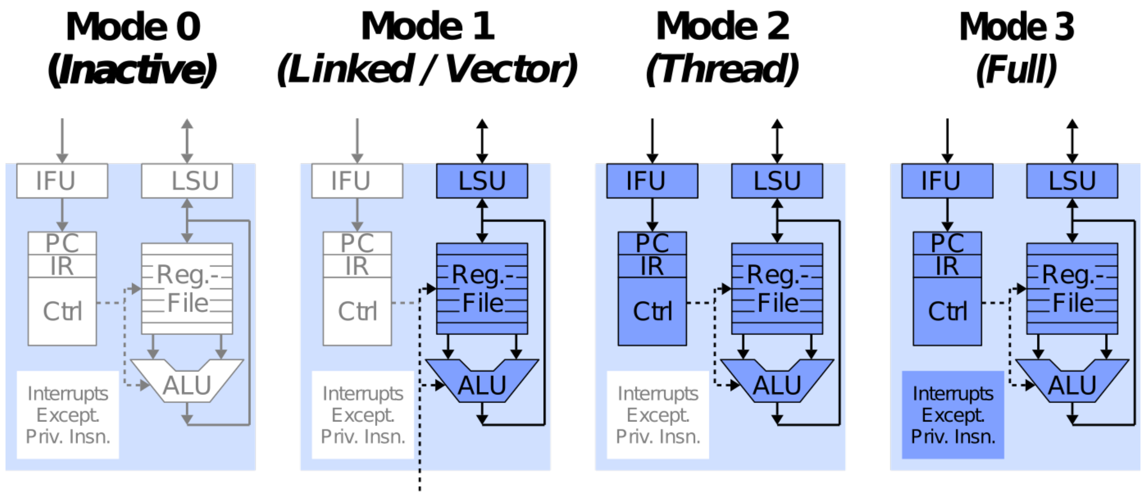


Figure 2.2.: *ParaNut* modes and required logic

2.4. SIMD Vectorization

In Mode 1, the CoPU performs exactly the same instructions as the CePU. This is the SIMD mode. All registers of the CePU can be regarded as a slice of a big vector register. Since all CPUs perform the same operation at a time, the memory bandwidth required for instruction fetching is reduced considerably and equivalent to the bandwith of a single-core processor.

From a software perspective, the code on a CoPU executes almost normally, just like multi-threaded code. There is only a single, well-defined exception: Conditional branches and jump instructions with variable target addresses are executed based on target address determined by the CePU. In the C language, such critical instructions can be generated out of “if” statements, “case” statements and loop constructs. As long as the conditions always evaluate equally on all CPUs, SIMD code can be easily written using a standard compiler and a thread-like programming model. Figure 2.3 shows an example of a vectorized loop. The macros 'pn_begin_linked' and 'pn_end_linked' open and close a parallel code section, respectively. Since the body of the “for” loop does not contain any conditional branches and the loop end condition “n $<$ 100” always evaluates equally on all CPUs, this code is executable on an SIMD-based processor variant.

2.5. Multi-Threading

To perform simultaneous multi-threading, the CoPUs are put into Mode 2. In this mode, all exceptions and interrupts are handled by the CePU. This is somewhat a limitation compared to Mode 3, in which the CPUs operate more autonomously. However, Mode 2 is sufficient for all typical applications, in which multi-threading is used as an acceleration measure.

```
int a[100], b[100], s[100];

void add_arrays_sequential () {
  for (n = 0; n < 100; n += 1)
    s[n] = a[n] + b[n];
}

void add_arrays_parallel () {
  int n, cpu_no;

  // Activate 3 (=4-1) CoPUs in the "Linked" state and
  pn_begin_linked (4);

  //  get the number of this CPU...
  cpu_no = pn_get_cpu_no();

  // performs 4 additions in  parallel
  for (n = 0; n < 100; n += 4)
    s[n + cpu_no] = a[n + cpu_no] + b[n + cpu_no];

  // End linked mode, deactivate the CoPUs...
  pn_end_linked ();
}
```

Figure 2.3.: Example of a vectorized loop

3. Instruction Set Reference

This chapter contains the instruction set reference for the *ParaNut* achitecture.

3.1. Privilege Levels

The *ParaNut* supports several combinations of privilege levels as specified in the RISC-V manual [3], which can be set in the global configuration setting `CFG_PRIV_LEVELS`. The currently supported combinations are listed in Table 3.1 and can be configured by setting the desired number of levels.

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 3.1.: Supported combinations of privilege modes. [3]

Note that the N-Extension, enabling User-mode exception and interrupt handling is currently not supported. Furthermore, no Memory Management Unit MMU is implemented at this time, thus no virtual address translation is possible in any mode.

3.2. Instructions

The *ParaNut* implements the RV32I base instruction set. It may be configured to additionally include the `M` and `A` extensions. For a full list of the corresponding instructions please refer to the RISC-V Instruction Set Manual Volume I [2]. This chapter contains additional implementation specific information on some instructions.

3.2.1. Conditional Branches

Currently no branch prediction is featured, branches as well as jumps stall the instruction fetch until the condition and/or address is evaluated.

3.2.2. Load and Store Instructions

A *ParaNut* raises the appropriate address misaligned exception on misaligned loads and stores. The trap is taken according to specification and the failing address is saved in *mtval* for further handling. Misaligned stores do not cause any changes in memory. Misaligned loads do not change the value of *rd*.

3.2.3. Memory Ordering Instructions

The *ParaNut* processor operates inorder and the write buffer of the Load Store Units is emptied inorder so the FENCE instruction is currently implemented as a LSU flush and the IFU buffer is also cleared.

For synchronization between a *ParaNut* processor and other hardware in the system the special cache control instructions described in Section 3.2.4 can be used.

3.2.4. Control and Status Register Instructions

SYSTEM instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other potentially privileged instructions.

3.2.5. Trap-Return Instructions

Information about these instructions can be found in the RISC-V Privileged Architecture Instruction Set Manual [3]

> ParaNut *does not implement the N-Extension, meaning URET is not supported. SRET is only available if S-mode is enabled.*

3.2.6. *ParaNut* Instructions

The *ParaNut* architecture uses the *custom-0* (0x0B) major opcode for its custom instructions as suggested in the RISC-V ISA manual [2].

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	HALT	0	CUSTOM-0
offset[11:0]	base	CINV	0	CUSTOM-0
offset[11:0]	base	CWB	0	CUSTOM-0
offset[11:0]	base	CFLUSH	0	CUSTOM-0
0	0	CINVA	0	CUSTOM-0
0	0	CWBA	0	CUSTOM-0
0	0	CFLUSHA	0	CUSTOM-0

The HALT instruction halts the current CPU by switching to Mode 0. If executed on the CePU it also halts all other CPUs in the system. Note that halting a mode 2 capable CPU will cause the reset of its program counter to the reset address.

The CINV, CWB and CFLUSH instructions control the MemU cache. All of these operate on the effective address obtained by adding register *rs1* to the sign extended 12-bit offset. CINV just invalidates the cache line containing the effective address, while CWB triggers a write back of the cache line to main memory. CFLUSH is the combination of CWB and CINV. Similarly the CINVA, CWBA and CFLUSHA serve the same function but execute it on the whole cache.

The CINV(A), CWB(A) and CFLUSH(A) instructions are also buffered in the LSU write buffer and are non blocking. They can take an arbitrary amount of time to complete. If you need the instruction to complete before continuing the execution follow it with a "fence" instruction to ensure the cache operation is fully executed.

3.3. Control and Status Registers (CSR)

This section describes the Control and Status Registers (CSRs), which are either standard machine or supervisor CSRs, or specific to the *ParaNut* architecture. The addresses used are defined in the RISC-V Privileged Architecture Instruction Set Manual [3]. All registers are 32 bits wide. Registers mentioned in Tables 3.4, 3.8, and 3.9 are readable only by the *CePU*.

The descriptions, tables and figures in Sections 3.3.1, 3.3.2 and 3.3.3 are derived from the RISC-V privileged ISA [3]. Clarifications or deviations from the specification are added as comments.

3.3.1. Terminology and Conventions for CSR Field Specifications

Tables 3.2 and 3.3 list abbreviations frequently used in this chapter. A more detailed description of the abbreviations may be found in Chapter 2.3 of the RISC-V Privileged Architecture Instruction Set Manual [3]. Tables 3.4, 3.8, and 3.9 contain information about the available CSRs and their access restrictions.

Abbreviation	Description
WIRI	Reserved Writes Ignored, Reads Ignore Values
WPRI	Reserved Writes Preserve Values, Reads Ignore Values
WLRL	Write/Read Only Legal Values
WARL	Write Any Values, Reads Legal Values

Table 3.2.: Write mode abbreviations

Privilege	Description
MRW	Machine Mode Readable/Writeable
MRO	Machine Mode Read-Only
URW	User Mode Readable/Writeable
URO	User Mode Read-Only
SRW	Supervisor Mode Readable/Writeable
SRO	Supervisor Mode Read-Only

Table 3.3.: Privilege abbriviations

3.3.2. Machine-Level Control and Status Registers

Table 3.4 lists all Control and Status Registers (CSR) implemented by the *ParaNut* architecture. Unless mentioned otherwise, they are implemented according to the RISC-V specification [3]. The following subsections describe the implementation-specific details as they are implemented on a *ParaNut* . Note, that all registers listed in this section are solely available on the CePU. Trying to access them from a CoPU raises an Illegal Instruction exception.

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRO	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.
Machine Counter/Timers			
0xB00	MRW	mcycle	Machine cycle counter.
0xB02	MRW	minstret	Machine instructions-retired counter.
0xB03	MRW	mhpmcounter3	Machine performance-monitoring counter.
0xB04	MRW	mhpmcounter4	Machine performance-monitoring counter.
		⋮	
0xB1F	MRW	mhpmcounter31	Machine performance-monitoring counter.
0xB80	MRW	mcycleh	Upper 32 bits of mcycle, RV32I only.
0xB82	MRW	minstreth	Upper 32 bits of minstret, RV32I only.
0xB83	MRW	mhpmcounter3h	Upper 32 bits of mhpmcounter3, RV32I only.
0xB84	MRW	mhpmcounter4h	Upper 32 bits of mhpmcounter4, RV32I only.
		⋮	
0xB9F	MRW	mhpmcounter31h	Upper 32 bits of mhpmcounter31, RV32I only.
Machine Counter Setup			
0x323	MRW	mhpmevent3	Machine performance-monitoring event selector.
0x324	MRW	mhpmevent4	Machine performance-monitoring event selector.
		⋮	
0x33F	MRW	mhpmevent31	Machine performance-monitoring event selector.
Machine Timer Registers			
0xF01	MRW	mtime	Machine timer register.
0xF02	MRW	mtimeh	Upper 32 bits of mtime
0xF03	MRW	mtimecmp	Machine timer compare register.
0xF04	MRW	mtimecmph	Upper 32 bits of mtimecmp

Table 3.4.: Currently defined standard RISC-V CSRs

3.3.2.1. Machine Vendor ID Register (`mvendorid`)

Returns a fixed value of 0 indicating a non-commercial implementation as defined in [3].

MXLEN-1   0
0 (Fixed)
MXLEN

Figure 3.1.: Vendor ID register (`mvendorid`).

3.3.2.2. Machine Architecture ID Register (`marchid`)

Returns a fixed value of 0, since the Architecture ID is not yet requested from the RISC-V Foundation.

MXLEN-1   0
0 (Fixed)
MXLEN

Figure 3.2.: Machine Architecture ID register (`marchid`).

3.3.2.3. Machine Implementation ID Register (`mimpid`)

This register provides detailed Information about the ParaNut hardware revision as shown in Figure 3.3. The ParaNut versioning scheme follows the very common Major, Minor, Revision scheme. Additionally bit 0 represents a dirty flag, indicating if the hardware has been modified.

31 24	23 16	15 1	0
Major	Minor	Revision	Dirty
8	8	15	1

Figure 3.3.: Machine Implementation ID register (`mimpid`).

3.3.2.4. Hart ID Register (`mhartid`)

The `mhartid` CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code. The RISC-V specification defines a hart as a single hardware thread. In the current ParaNut implementation, multiple hardware threads on a single core are not supported. Therefore, the Hart ID Register is equivalent to `pncoreid`. `mhartid` can only be accessed by the CePU, which means it always returns zero.

MXLEN-1   0
Hart ID
MXLEN

Figure 3.4.: Hart ID register (`mhartid`).

3.3.2.5. Machine Status Register (`mstatus`)

Implements the flags listed in Figure 3.5, which represent only a subset of `mstatus` in [3]. *WPRI* indicates that the bits are not yet implemented and should be preserved on writes for forward compatibility reasons, as indicated in Table 3.2

MXLEN-1 … 9	8	7	6	5	4	3	2	1	0
WPRI	SPP	MPIE	*WPRI*	SPIE	*WPRI*	MIE	*WPRI*	SIE	*WPRI*
23	1	1	1	1	1	1	1	1	1

Figure 3.5.: Machine-mode status register (`mstatus`) for RV32.

3.3.2.6. Machine ISA Register (`misa`)

The `misa` CSR is a *WARL read-only* register reporting the ISA supported by the hart. As the *ParaNut* is highly configurable, the Extensions filed may or may not report some extensions. Table 3.5 shows the possibilities of configuration. MXL is fixed to 1 to indicate 32-bit support.

MXLEN-1 MXLEN-2	MXLEN-3 26	25 … 0
MXL[1:0] (*WARL*)	*WIRI*	Extensions[25:0] (*WARL*)
2	MXLEN-28	26

Figure 3.6.: Machine ISA register (`misa`).

Bit	Character	Fixed/Configuration	Description
0	A	`CFG_EXU_A_EXTENSION=1`	Atomic extension
8	I	Fixed to `1`	RV32I/64I/128I base ISA
12	M	`CFG_EXU_M_EXTENSION=1`	Integer Multiply/Divide extension
18	S	`CFG_PRIV_LEVELS=3`	Supervisor mode implemented
20	U	`CFG_PRIV_LEVELS≥2`	User mode implemented
23	X	Fixed to `1`	ParaNut extensions present

Table 3.5.: Encoding of Extensions field in `misa`.

3.3.2.7. Machine Interrupt Registers (`mip` and `mie`)

These registers are read-write registers, but currently without any functionality. In later revisions they might be reworked.

3.3.2.8. Machine Trap Vector Base Address Register (`mtvec`)

Currently, the lowest two bits are fixed to zero, which indicates that all traps set the program counter to BASE+4.

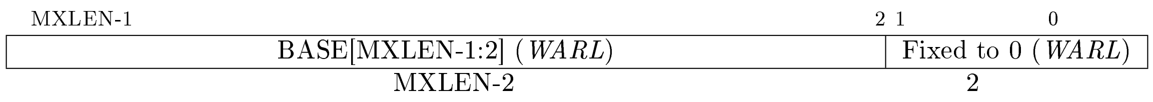


Figure 3.7.: Supervisor trap vector base address register (`stvec`).

3.3.2.9. Machine Trap Delegation Registers (`medeleg` and `mideleg`)

These registers are only available if the configuration parameter `CFG_PRIV_LEVELS` is set to 3, meaning supervisor mode is enabled.

3.3.2.10. Machine Cause Register (`mcause`)

After a trap occured, `mcause` contains one of the flags listed in Table 3.6. Note that environment calls may only occur if the corresponding mode is configured.

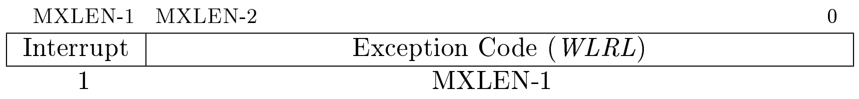


Figure 3.8.: Machine Cause register `mcause`.

3.3.2.11. Hardware Performance Monitor

The hardware performance monitor counters can be configured in the *ParaNut* at compile or synthesis time through the configuration file. They can be fully disabled for minimal space requirements. Reads will then return a fixed value of zero.

When the performance counters are enabled, `mcycle/h` has a width of 64 bit, but the width of all the other performance counters can be configured to be between 33 and 64 bit. Also the amount of performance registers can be changed from 8 to 32. A minimum of 8 is required because the first 6 are reserved for the events specified in Table 3.7. These registers will also be set to zero on reset and won't read an arbitrary value. Since the events for the counters are implementation specific the `mhpmevent3-mphmevent31` registers have a fixed value of zero.

3.3.2.12. Machine Timer Registers (`mtime` and `mtimecmp`)

The *ParaNut* currently doesn't implement any timers, hence `mtime/h` and `mtimecmp/h` read fixed values of zero and are implemented as *WARL* on writes. In a future implementation, they will be implemented according to [3].

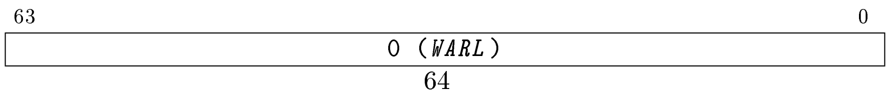


Figure 3.9.: Machine time register (memory-mapped control register).

Interrupt	Exception Code	Description
1	0	*Not implemented*
1	1	*Not implemented*
1	2	*Not implemented*
1	3	*Not implemented*
1	4	*Not implemented*
1	5	*Not implemented*
1	6	*Not implemented*
1	7	*Not implemented*
1	8	*Not implemented*
1	9	*Not implemented*
1	10	*Not implemented*
1	11	*Not implemented*
1	≥12	*Reserved*
0	0	Instruction address misaligned
0	1	*Not implemented*
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	*Not implemented*
0	6	Store/AMO address misaligned
0	7	*Not implemented*
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Not implemented*
0	11	Environment call from M-mode
0	12	*Not implemented*
0	13	*Not implemented*
0	14	*Not implemented*
0	15	*Not implemented*
0	16	ParaNut CoPU exception
0	≥17	*Reserved*

Table 3.6.: Machine cause register (`mcause`) values after trap.

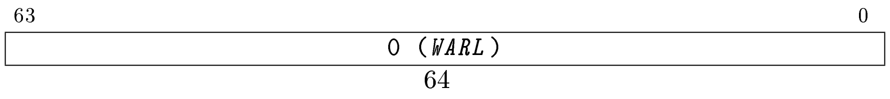


Figure 3.10.: Machine time compare register (memory-mapped control register).

3.3.3. Supervisor Control and Status Registers

This chapter describes the RISC-V supervisor-level Control and Status Registers listed in 3.8, which were originally specified in RISC-V Volume II [3]. Note that these registers are only available when the *ParaNut* was configured to implement supervisor mode.

In the following subsections, all registers and their flags are listed and explained if the

Register	Description/Event
`mhpmcounter3/h`	Number of ALU operations since reset. (ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND)
`mhpmcounter4/h`	Number of LOAD operations since reset. (LB, LH, LW, LBU, LHU)
`mhpmcounter5/h`	Number of STORE operations since reset. (SB, SH, SW)
`mhpmcounter6/h`	Number of JUMP/BRANCH operations since reset. (JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BLGEU)
`mhpmcounter7/h`	Number of SYSTEM/SPECIAL operations since reset. (FENCE, ECALL, EBREAK, MRET, CSRRW, CSRRS, CSRRC, CSRRWI, CSRRSI, CSRRCI)

Table 3.7.: Fixed events of the first four counters.

Number	Privilege	Name	Description
Supervisor Trap Setup			
`0x100`	SRW	`sstatus`	Supervisor status register.
`0x104`	SRW	`sie`	Supervisor interrupt-enable register.
`0x105`	SRW	`stvec`	Supervisor trap handler base address.
Supervisor Trap Handling			
`0x140`	SRW	`sscratch`	Scratch register for supervisor trap handlers.
`0x141`	SRW	`sepc`	Supervisor exception program counter.
`0x142`	SRW	`scause`	Supervisor trap cause.
`0x143`	SRW	`stval`	Supervisor bad address or instruction.

Table 3.8.: Currently allocated supervisor RISC-V CSRs

ParaNut 's behaviour differs from the RISC-V specification. All registers may only be accessed on the CePU. Trying to access them from a CoPU raises an Illegal Instruction exception.

3.3.3.1. Supervisor Status Register (`sstatus`)

The flags listed in Figure 3.11 represent a subset of `mstatus` and are implementes as defined in [3]. *WPRI* indicates that the bits are not yet implemented and should be preserved on writes for forward compatibility reasons.

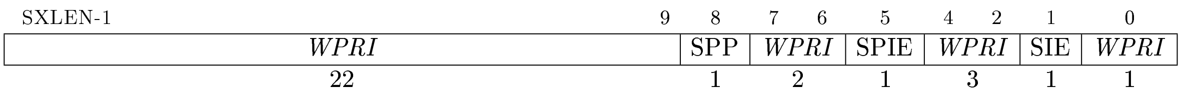


Figure 3.11.: Supervisor-mode status register (`sstatus`) for RV32.

3.3.3.2. Supervisor Cause Register (`scause`)

The `scause` register behaves analogous to `mcause` and may contain values listed in Table 3.6.

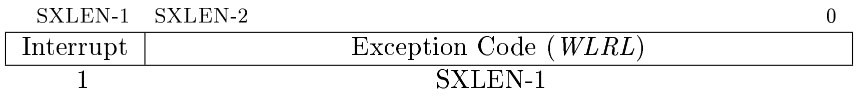


Figure 3.12.: Supervisor Cause register `scause`.

3.3.4. *ParaNut* -Specific Control and Status Registers

Table 3.9 shows the *ParaNut* -specific registers, which are used to query the hardware configuration and to read the status of the CPU array. All registers are only available on a CePU, except for `pncoreid`, which can also be read by CoPUs. All of these registers are available in any configuration of the *ParaNut* , regardless of which privilege modes are implemented.

Number	Privilege	Name	Description
ParaNut Machine R/W (Non-Standard R/W)			
`0x7C0`	MRW	`pncache`	ParaNut Cache Control register.
ParaNut User R/W (Non-Standard R/W)			
`0x8C0`	URW	`pngrpsel`	ParaNut CPU group select.
`0x8C1`	URW	`pnce`	ParaNut CPU enable register.
`0x8C2`	URW	`pnlm`	ParaNut CPU linked mode register.
`0x8C3`	URW	`pnxsel`	ParaNut CoPU exception select register.
ParaNut Machine RO (Non-Standard RO)			
`0xFC0`	MRO	`pnm2cp`	ParaNut CPU capabilities register
`0xFC1`	MRO	`pnx`	ParaNut CoPU exception pending.
`0xFC2`	MRO	`pncause`	ParaNut CoPU trap cause ID.
`0xFC3`	MRO	`pnepc`	ParaNut CoPU exception program counter.
`0xFC4`	MRO	`pncacheinfo`	ParaNut cache information.
`0xFC5`	MRO	`pncachesets`	ParaNut number of cache sets.
`0xFC6`	MRO	`pnclockinfo`	ParaNut clock speed information.
`0xFC7`	MRO	`pnmemsize`	ParaNut memory size.
ParaNut User R (Non-Standard R)			
`0xCD0`	URO	`pncpus`	ParaNut number of CPUs.
`0xCD4`	URO	`pncoreid`	ParaNut core ID. Can be accessed by CoPUs

Table 3.9.: Currently allocated *ParaNut* -specific CSRs

3.3.4.1. ParaNut CPU group select (`pngrpsel`)

The `pngrpsel` register is an MXLEN-bit read-write register formatted as shown in Figure 3.13. It only takes legal values (illegal values are ignored) and selects the group of 32

CPUs on which the *ParaNut* CSRs that work on one bit per CPU (`pnce, pnlm, pnxsel, pnm2cp, pnx`) function. On *ParaNut* systems with fewer than 32 CPUs this register will only read and hold a value of zero. On systems with more than 32 CPUs `pngrpsel` should be checked/set before reading or writing these CSRs.

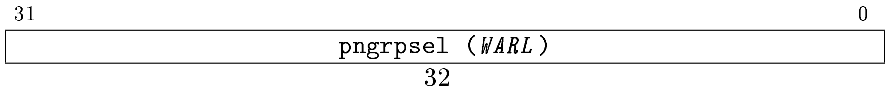


Figure 3.13.: ParaNut CPU group select (`pngrpsel`).

3.3.4.2. Supervisor Trap Vector Base Address Register (`stvec`)

Currently, the lowest two bits are fixed to zero, which indicates that all traps set the program counter to BASE+4.

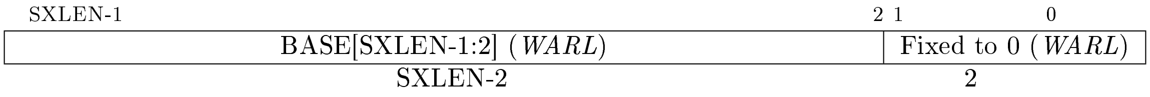


Figure 3.14.: Supervisor trap vector base address register (`stvec`).

3.3.4.3. ParaNut CPU enable register (`pnce`)

The `pnce` register is an MXLEN-bit read-write register formatted as shown in Figure 3.15. It only takes legal values (*WARL*). Each bit corresponds to one CPU, bit 0 represents the CePU. By writing into this register, the CePU can activate or deactivate CoPUs. By reading the register, the CePU can determine whether the CoPU is actually (in)active (enabled/halted). Both activation and deactivation may take some time until the CoPU reaches a stable state. On deactivation by the CePU the CoPU is guaranteed to finish it's current instruction.
After deactivation the CPU will be in Mode 0. For CPUs with capability ≥ 2 this means their IFU is reset and upon activation they will start execution at the reset vector address. In systems with more than 32 CPUs the `pngrpsel` register must be used to control CoPUs with core ID > 31.

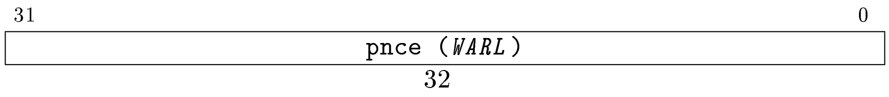


Figure 3.15.: ParaNut CPU enable register (`pnce`).

3.3.4.4. ParaNut CPU linked mode register (`pnlm`)

The `pnlm` register is an MXLEN-bit read-write register formatted as shown in Figure 3.16. It only takes legal values (*WARL*). Each bit corresponds to one CPU and bit 0 represents the CePU. If the bit is set for CoPU, the CoPU is in linked state (Mode 1). If the bit is

unset, it is in unlinked state (Mode 2 or 3). By writing into this register, the CePU can switch the mode of the CoPUs. Mode switching is allowed only if the CoPU is inactive and not presently activated. If a bit is changed in the PNLM register and the respective PNCE bit is 1, undefined behavior may result.
In systems with more than 32 CPUs the `pngrpsel` register must be used to control CoPUs with core ID $>$ 32.

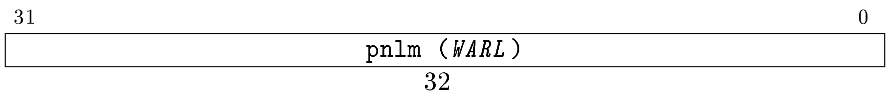


Figure 3.16.: ParaNut CPU linked mode register (`pnlm`).

3.3.4.5. ParaNut CoPU exception select register (`pnxsel`)

The `pnxsel` register is an MXLEN-bit read-write register formatted as shown in Figure 3.16. It only takes legal values (*WARL*). Each bit corresponds to one CPU and bit 0 represents the CePU. By writing into this register, the CePU can select which CoPUs exception information can be read from the `pnepc` and `pncause` CSRs. Only one bit should be set at any time to avoid unwanted behavior.
In systems with more than 32 CPUs the `pngrpsel` register must be used to control CoPUs with core ID $>$ 31.

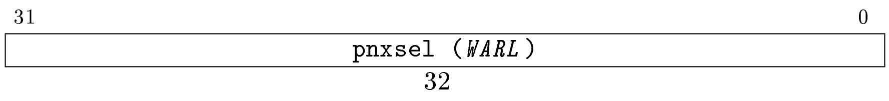


Figure 3.17.: ParaNut CoPU exception select register (`pnxsel`).

3.3.4.6. ParaNut Cache control register (`pncache`)

The `pncache` register is an MXLEN-bit read-write register formatted as shown in Figure 3.18. It only takes legal values (*WARL*).

The DEN field enables (1) or disables (0) the use of the cache for data access.

The IEN field enables (1) or disables (0) the use of the cache for data access.

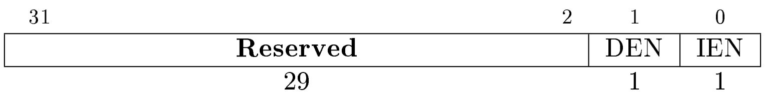


Figure 3.18.: ParaNut Cache control register.

Writing to these registers does not trigger any flush or write-back operation. Hence, when disabling the cache, it must be flushed or written back by software using the CFLUSH(A) or CWB(A) instructions listed in Section 3.2.6 if the cache may contain modified data.

3.3.4.7. ParaNut number of CPUs (`pncpus`)

The `pncpus` register is an MXLEN-bit read-only register formatted as shown in Figure 3.19. It holds the number of CPUs (including the CePU).

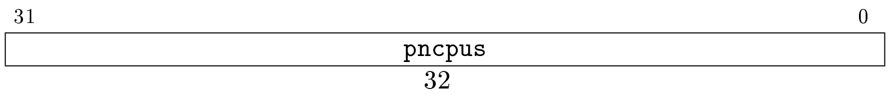


Figure 3.19.: ParaNut number of CPUs (`pncpus`).

3.3.4.8. ParaNut CPU capabilities register (`pnm2cp`)

The `pnm2cp` register is an MXLEN-bit read-only register formatted as shown in Figure 3.20. Each bit corresponds to one CPU. If the bit is set, the respective CPU supports Mode 2 (thread mode) or higher. If unset, the respective CPU supports only Mode 0 (halt) and Mode 1 (linked). Bit 0 represents the CePU and must be set in every implementation.
In systems with more than 32 CPUs the `pngrpsel` register must be used to read the capabilities of CoPUs with core ID $>$ 31.

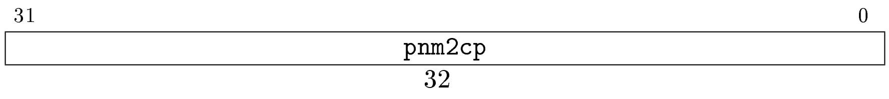


Figure 3.20.: ParaNut CPU capabilities register (`pnm2cp`.

3.3.4.9. ParaNut CoPU exception pending (`pnx`)

The `pnx` register is an MXLEN-bit read-only register formatted as shown in Figure 3.21. Each bit corresponds to one CPU. It is written by hardware on trap entry. If a bit is set, the represented CoPU encountered an exception and awaits handling.
In systems with more than 32 CPUs the `pngrpsel` register must be used to read the pending state of CoPUs with core ID $>$ 31.

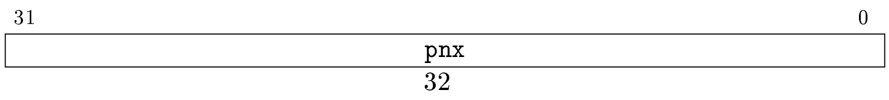


Figure 3.21.: ParaNut CoPU exception pending (`pnx`).

3.3.4.10. ParaNut CoPU trap cause ID (`pncause`)

The `pncause` register is an MXLEN-bit read-only register formatted as shown in Figure 3.22. It holds the cause of exception of the CoPU selected by `pnxsel` and `pngrpsel`. The CSR only holds legal values as defined in `mcause`.

31 … 0
`pncause`
32

Figure 3.22.: ParaNut CoPU trap cause ID (`pncause`).

3.3.4.11. ParaNut CoPU exception program counter (`pnepc`)

The `pnepc` register is an MXLEN-bit read-only register formatted as shown in Figure 3.23. It holds the exception program counter of the CoPU selected by `pnxsel` and `pngrpsel`. The CSR only holds legal values as defined in `mepc`.

31 … 0
`pnepc`
32

Figure 3.23.: ParaNut CoPU exception program counter (`pnepc`).

3.3.4.12. ParaNut cache information register (`pncacheinfo`)

The `pncacheinfo` register is an MXLEN-bit read-only register formatted as shown in Figure 3.24. It holds information about the cache properties.

31 … 8	7 … 3	2 … 1	0
Cache Banks	Arbiter Method	WAYS	REPM
24	5	2	1

Figure 3.24.: ParaNut cache information register (`pncacheinfo`).

The REPM field indicates the cache replacement method. A Least Recently Used (LRU) replacement strategy is used if it is set, else random replacement is in action.

The WAYS field shows the associativity of the cache. Valid values are 0, 1 and 2 corresponding to 1, 2 and 4 way associativity.

The Arbiter Method field encodes the used method during arbitration of cache and bus accesses. It is a **signed** number. On positive values a round-robin arbitration that switches every 2^{value} clocks is used. On negative values a pseudo-random arbitration based on Linear Feedback Shift Registers (LSFR) is used.

The Cache Banks field holds the number of cache banks.

The overall size of the available cache can be calculated as:
$pncachesets * Cache\ Banks * 4\ Bytes.$

3.3.4.13. ParaNut number of cache sets register (`pncachesets`)

The `pncachesets` register is an MXLEN-bit read-only register formatted as shown in Figure 3.25. It holds the number of cache sets.

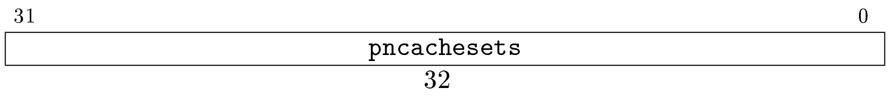


Figure 3.25.: ParaNut number of cache sets register (`pncachesets`).

The overall size of the available cache can be calculated as:
$pncachesets * Cache\ Banks * 4\ Bytes.$

3.3.4.14. ParaNut clock speed information register (`pnclockinfo`)

The `pnclockinfo` register is an MXLEN-bit read-only register formatted as shown in Figure 3.26. It holds the clock speed in Hz set at compile or synthesis time.

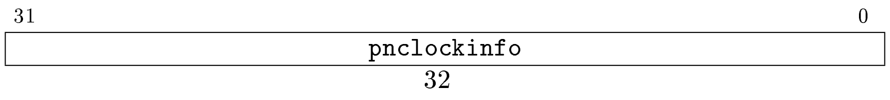


Figure 3.26.: ParaNut clock speed information register (`pnclockinfo`.

3.3.4.15. ParaNut memory size register (`pnmemsize`)

The `pnmemsize` register is an MXLEN-bit read-only register formatted as shown in Figure 3.27. It holds the memory size set at compile or synthesis time.

3.3.4.16. ParaNut core ID register (`pncoreid`)

The `pncoreid` register is an MXLEN-bit read-only register formatted as shown in figure 3.28. It is the only register accesible from CoPUs. This is required to initiate LinkedMode.

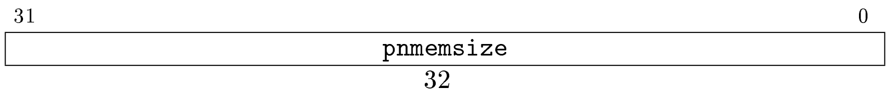


Figure 3.27.: ParaNut memory size register (`pnmemsize`).

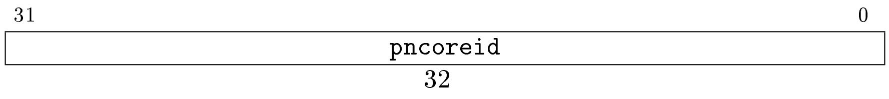


Figure 3.28.: ParaNut core ID register (`pncoreid`).

3.4. Exceptions

Table 3.6 lists the exceptions supported by the *ParaNut* architecture. At the moment, only those classified as `implemented` can occur in the CePU. In a CoPU of mode 2 the same exceptions may arise, excluding the `ParaNut CoPU exception`, which is used to signal to the CePU that an exception occured in one of the CoPUs.

If an exception occurs in the CePU, the following steps are performed:

1. Trap information is saved to the following registers:
   - The address of the current instruction (PC) in `mepc`
   - The appropriate cause in `mcause`
   - The current value of the pnx input port in `pnx`
   - Interrupts are disabled by writing the value of MIE to MPIE and setting MIE to zero in `mstatus`
2. The CePU triggers and waits for all CoPUs (enabled/linked or not) to change into Mode 0 (halt) after they finish their current instruction.
3. Execution is continued at the address saved in the `mtvec` register.
4. *Execution of the exception handler*
5. The exception handler finishes by using the MRET instruction which continues execution at the address saved in `mepc` and takes all CoPUs back to their previous exception state.

The change in execution mode in step 2 is not visible to the programmer through the `pnce` *or* `pnlm` *CSRs. However writing to these registers will influence/change the execution mode of the CoPUs after executing the MRET instruction in the CePU.*

We decided on this approach to simplify the hardware and remove the need for shadow registers which save the state of the `pnce` *or* `pnlm` *CSRs on exception entry.*

If an exception occurs inside a Mode 2 CoPU, the following steps are performed:

1. The CoPU halts itself and signals an exception to the CePU.
2. The CePU finishes it's current instruction and starts the exception handling procedure as described above with the special CoPU exception cause (see Table 3.6).
3. The CePU triggers and waits for all CoPUs (enabled/linked or not) to change into their exception state after they finish their current instruction.
4. Execution is continued at the address saved in the `mtvec` register.
5. *Execution of the exception handler*
   - By reading `pnx` the exception handler can determine on which CoPU(s) an exception occurred and after setting the `pnxsel` CSR the cause and PC of the selected CoPU can be read from the `pncause` and `pnepc`.
   - The CoPU must be enabled through `pnce` to indicate that the exception was handled and that the execution can continue for the next instruction. (Note: otherwise the CoPU will still indicate an exception to the CePU, which in turn will reenter the exception handling procedure again)
6. The exception handler finishes by using the MRET instruction which continues execution at the address saved in `mepc` and takes all CoPUs out of their exception state.

If an exception occurs inside a Mode 1 CoPU, the following steps are performed:

1. If any of the CoPUs is in linked mode (Mode 1), all Mode-1-CoPUs and the CePU must be designed such that they either all complete their current instruction or all of them perform a roll back. If this is not ensured, the interrupted code is not restartable.
2. The CoPU halts itself and signals an exception to the CePU.
3. The CePU starts the exception handling procedure as described above with the special CoPU exception cause (see Table 3.6).
4. The CePU triggers and waits for all CoPUs (enabled/linked or not) to change into their exception state after they finish their current instruction.
5. Execution is continued at the address saved in the `mtvec` register.
6. *Execution of the exception handler*
   - By reading `pnx` the exception handler can determine on which CoPU(s) an exception occurred and after setting the `pnxsel` CSR the cause and PC of the selected CoPU can be read from the `pncause` and `pnepc`.
   - The CoPU must be enabled through `pnce` to indicate that the exception was handled. (Note: otherwise the CoPU will still indicate an exception to the CePU, which in turn will reenter the exception handling procedure again)

7. The exception handler finishes by using the MRET instruction which continues execution at the address saved in `mepc` and takes all CoPUs out of their exception state.

Bibliography

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

[2] :Andrew Waterman, Krste Asanović, RISC-V Foundation: "The RISC-V Instruction Set Manual Volume I: User-Level ISA", Document Version 2.2, 2017, www.riscv.org

[3] Andrew Waterman, Krste Asanović, RISC-V Foundation: "The RISC-V Instruction Set Manual Volume II: Privileged Architecture", Document Version 1.10, 2017, www.riscv.org

[4] John. L. Hennessy, David A. Patterson: "Computer Architecture: A Quantitative Approach", 5th edition, Elsevier, 2012

A. Appendix

A.1. Building software for the *ParaNut* processor

Prerequisites:

- The RISC-V GCC toolchain.
- Built SystemC simulation (`paranut_tb`).

The *ParaNut* repository contains tested software in the `sw` folder. A good starting point for developing your own software would be the `hello_newlib` example. It contains following files:

```c
#include <stdio.h>
#include <unistd.h>

int main () {
  int n;

  for (n = 1; n <= 10; n++)
    printf ("%2i. Hello World!\n", n);
  return 0;
}
```

Listing A.1: hello_newlib.c, simple application using the newlib

```makefile
# Root of ParaNut repository or local project
PARANUT ?= ../..

# Flash target options
PN_FIRMWARE_ELF ?=
PN_SYSTEM_HDF ?=
PN_SYSTEM_BIT ?=

# Configuration options
CROSS_COMPILE ?= riscv64-unknown-elf

CC      := $(CROSS_COMPILE)-gcc
GXX     := $(CROSS_COMPILE)-g++
OBJDUMP := $(CROSS_COMPILE)-objdump
OBJCOPY := $(CROSS_COMPILE)-objcopy
GDB     := $(CROSS_COMPILE)-gdb
```

```
AR      := $(CROSS_COMPILE)-ar
SIZE    := $(CROSS_COMPILE)-size

ELF = hello_newlib
SOURCES = $(wildcard *.c)
OBJECTS = $(patsubst %.c,%.o,$(SOURCES))
HEADERS = $(wildcard *.h)

PN_SYSTEMS_DIR = $(PARANUT)/systems
RISCV_COMMON_DIR = $(PARANUT)/sw/riscv_common

CFG_MARCH ?= rv32i

CFLAGS = -O2 -march=$(CFG_MARCH) -mabi=ilp32 -I$(RISCV_COMMON_DIR)
LDFLAGS = $(CFLAGS) -static -nostartfiles -lc $(RISCV_COMMON_DIR)/startup.S $(
    RISCV_COMMON_DIR)/syscalls.c -T $(RISCV_COMMON_DIR)/paranut.ld

# Software Targets
all: $(ELF) dump

$(ELF): $(OBJECTS)
$(CC) -o $@ $^ $(LDFLAGS)

%.o: %.c $(HEADERS)
$(CC) -c $(CFLAGS) $<


# ParaNut Targets
.PHONY: sim
sim: $(ELF)
+$(MAKE) -C $(PARANUT)/hw/sim pn-sim
$(PARANUT)/hw/sim/pn-sim -t0 $<

# Generic Flash targets (set PN_* accordingly)
.PHONY: flash flash-bit
flash: bin
pn-flash -c -p $(ELF).bin $(PN_SYSTEM_HDF) $(PN_FIRMWARE_ELF)

flash-bit: bin
pn-flash -c -b $(PN_SYSTEM_BIT) -p $(ELF).bin $(PN_SYSTEM_HDF) $(
    PN_FIRMWARE_ELF)


# Special System Flash targets for testing inside the source repository
.PHONY: flash-%
flash-%: bin
if [ ! -d $(PN_SYSTEMS_DIR) ]; then echo; echo "INFO: The flash targets are
    only for testing inside the source repository!"; echo; exit 1; fi
pn-flash -c -p $(ELF).bin $(PN_SYSTEMS_DIR)/$*/hardware/build/system.hdf $(
    PN_SYSTEMS_DIR)/$*/hardware/firmware/firmware.elf
```

```

flash-%-bit: bin
if [ ! -d $(PN_SYSTEMS_DIR) ]; then echo "INFO: The flash targets are only for
    testing inside the source repository!"; exit 1; fi
pn-flash -c -b $(PN_SYSTEMS_DIR)/$*/hardware/build/system.bit -p $(ELF) \
$(PN_SYSTEMS_DIR)/$*/hardware/build/system.hdf $(PN_SYSTEMS_DIR)/$*/hardware/
    firmware/firmware.elf


# Misc Targets
.PHONY: dump
dump: $(ELF).dump
$(ELF).dump: $(ELF)
$(OBJDUMP) -S -D $< > $@

.PHONY: bin
bin: $(ELF).bin
$(ELF).bin: $(ELF)
$(OBJCOPY) -S -O binary $< $@

.PHONY: clean
clean:
rm -f *.o *.o.s *.c.s $(ELF) $(ELF).bin $(ELF).dump
```

Listing A.2: Makefile, for building software with the newlib

The Makefile requires the correct path to the top-level paranut folder `PN_PARANUT` set correctly to include the following *ParaNut* specific files:

- `startup.s`: *ParaNut* startup file containing the reset routine.
- `syscalls.c`: Implementation of the system calls required by the newlib (libgloss).
- `encoding.h`: Defines and other helpers.
- `paranut.ld`: Linker script for the *ParaNut* memory model.

By default the parameter `CFG_MARCH` is set to rv32i (only RV32I instructions). These can be changed according to the configuration made in the global config file.

To build the `hello_newlib` application follow these steps (provided you are currently in the top level directory of the paranut repository):

```
$ cd sw/hello_newlib
```

```
$ make
```

Example for a build with different configuration:

```
$ make CFG_MARCH=rv32im
```

A.1.1. Run the application in the SystemC simulation

To run the application in the SystemC simulation run the `paranut_tb` with the built ELF file as parameter:

```
$ $PARANUT_HOME/hw/sim/pn-sim hello_newlib
```

Or use the `sim` target of the Makefile:

```
$ make sim
```

To get a *GTK-Wave* compatible trace file run the SystemC simulation with the `-t` parameter and a number bigger than 0:

```
$ $PARANUT_HOME/hw/sim/pn-sim -t1 hello_newlib
```

- `-t0`: No trace file will be generated.
- `-t1`: Top level bus and paranut signals.
- `-t2`: First level of internal module signals (EXU, MEMU, IFU, LSU, ...).
- `-t3`: Second level of internal modules (MExtension, ReadPorts, WritePorts, ...)

A.2. Using GDB with the SystemC simulation

Prerequisites:

- The RISC-V compatible OpenOCD (See https://github.com/riscv/riscv-tools) for build instructions.
- The RISC-V GCC toolchain.
- Built SystemC simulation (`paranut_tb`).
- Built RISC-V application (with debug symbols and without optimization) (A.1).

The *ParaNut* SystemC simulation is compatible with the RISC-V External Debug Support Version 0.13. Thus it can be debugged using the GNU Debugger (GDB) of the RISC-V toolchain. Since the *ParaNut* simulation acts like real hardware we use OpenOCD to communicate with GDB.

Run the SystemC simulation with the ELF file you want to debug and the `-d` parameter to tell it to wait for a OpenOCD connection:

e.g. local repository:

```
$ $PARANUT_HOME/hw/sim/pn-sim -d hello_newlib
```

e.g. installed:

```
$ $PARANUT_HOME/bin/pn-sim -d hello_newlib
```

In a new shell start OpenOCD and use the `tools/etc/openocd-sim.cfg` configuration file:

> *Currently you have to use the OpenOCD built with the RISC-V tools. If you have not added the `$RISCV/bin` folder to your PATH or have a different version installed start OpenOCD with the full path name to avoid errors. E.g. `/opt/riscv/bin/openocd`*

```
$ openocd -f $PARANUT_TOOLS/etc/openocd-sim.cfg
```

In yet another shell start the RISC-V GDB debug session:

```
$ riscv64-unknown-elf-gdb hello_newlib
```

Lastly connect to OpenOCD as remote target:

```
(gdb) target remote localhost:3333
```

Now you are able to use all standard GDB commands to debug the application:

```
(gdb) break main
```

```
(gdb) continue
```

```
(gdb) next
```

```
(gdb) print n
```

```
(gdb) help
```

To reset the processor and start from the reset vector use following command:

```
(gdb) monitor reset halt
```

This will automatically reload the ELF file into the simulated memory.

A.3. Using and debugging the hardware

Prerequisites:

- The RISC-V compatible OpenOCD (See https://github.com/riscv/riscv-tools) for build instructions.
- The RISC-V GCC toolchain.

- Supported FPGA board (e.g. Digilent Zybo, Digilent Zybo Z7-20)
- A JTAG debugger (e.g. Amontec JTAGkey)
- Built RISC-V application (with debug symbols and without optimization) (A.1).

The *ParaNut* reference system located in the `systems` directory can be debugged using a standard JTAG debugger. The *ParaNut* processor in this system is compatible with the RISC-V External Debug Support Version 0.13. Thus it can be debugged using the GNU Debugger (GDB) of the RISC-V toolchain.

Build the reference design for the hardware you are using:

```
$ cd systems/refdesign
```

e.g. Digilent Zybo:

```
$ make build BOARD=zybo
```

e.g. Digilent Zybo Z7

```
$ make build BOARD=zybo_z7020
```

This will also build a firmware for the ARM core on these boards and a copy of the `hello_newlib` software in to the `software` directory.
Connect the board to your PC and program the firmware, bitfile and RISC-V software to the board by executing following command (see the Makefile to see the full command using the pn-flash tool):

```
$ make -C systems/refdesign run
```

A console will stay running and showing the standard output of the *ParaNut* processor. After a few seconds to invalidate the cache the "Hello World" messages should be visible.

Connect the JTAG debugger outputs to the JD Pmod pin header on the boards as shown in Table A.1. The table displays how the pins coming from the Amontec JTAGkey should be connected, so JD10 is TDI of the *ParaNut* JTAG TAP and JD7 is its TDO.

VCC	**GND**	**JD4**	**JD3**	**JD2**	**JD1**
N.C.	N.C.	N.C.	N.C.	N.C.	N.C.
VCC	**GND**	**JD10**	**JD9**	**JD8**	**JD7**
VREF	GND	TDO	TCK	TMS	TDI

Table A.1.: JD Pmod Port JTAG pin connections for the Amontec JTAGkey

In a new shell start OpenOCD and use the `tools/etc/openocd-board.cfg` configuration file if you use the Amontec JTAGkey (modify the configuration if you use a different JTAG debugger):

Currently you have to use the OpenOCD built with the RISC-V tools. If you have not added the `$RISCV/bin` *folder to your PATH or have a different version installed start OpenOCD with the full path name to avoid errors. E.g.* `/opt/riscv/bin/openocd`

```
$ openocd -f $PARANUT_TOOLS/etc/openocd-board.cfg
```

In yet another shell start the RISC-V GDB debug session:

```
$ riscv64-unknown-elf-gdb software/hello_newlib
```

Lastly connect to OpenOCD as remote target:

```
(gdb) target remote localhost:3333
```

Now you are able to use all standard GDB commands to debug the application:

```
(gdb) break main
```

```
(gdb) continue
```

```
(gdb) next
```

```
(gdb) print n
```

```
(gdb) help
```

Load the elf again through GDB:

```
(gdb) load
```

To reset the processor and start from the reset vector use following command:

```
(gdb) monitor reset halt
```

A.4. Integrating your own hardware modules

Due to the permissive license of the *ParaNut* project, anyone is allowed to add modules. In general, there are two ways to do so:
- Integrating an AXI compatible module to the SoC
- Extending the *ParaNut* architecture/hardware itself.

A.4.1. AXI compatible modules

A.4.2. Extending the *ParaNut* architecture/hardware

For simplification, all steps are explained with the CSR module as an example.

1. Develop your module and adapt the other modules according to your needs.

2. Integrate your SystemC module into the simulator (if developing in VHDL, you can skip to step 4

   - In the easiest case, you can instantiate a submodule in the parent module (similar to the `MMExtension` submodule inside the ExU - see `mextension.h/cpp and exu.h/cpp`). However, this inhibits the High Level Synthesis in case the submodule and the parent module include the same header file
   - For any other case, create a signal for each port of your new module in `paranut.h`. Afterwards, instantiate the module in `paranut.cpp` and bind all ports to their corresponding signal as well as all newly created ports in the other modules.

3. High Level Synthesis (HLS)

   - Copy a HLS script in `sysc`, e.g. `exu.tcl`, name it similar to your source file (`csr.tcl`) and change the following parameters:
     - `open_project` (the Vivado HLS project name and resulting folder; usually the modules name prefixed by `hls-`: `hls-csr`)
     - `set_top` (the module name, i.e. the SystemC class: `MCsr`)
     - `add_files` (all source files: `csr.cpp`)
   - The script will automatically be executed when creating the IP core. You may also run the script manually by executing make `copy-yourmodulesname` (`make copy-csr`)
   - The resulting files of HLS are files are copied to the directory `hw/rtl/vhdl/` and are usually prefixed with the the modules name (`MCsr*.vhd`)

4. Now copy two new files in `hw/rtl/vhdl/` similar to `mcsr.vhd` for the module wrapper and `csr.vhd` for the port declaration; adapt them to your module. This step hides all ports behind a more convenient port declaration, usually named similar to the module and prefixed with `i` for its input ports or an `o` for output ports respectively (`csri`, `csro`).

5. In the file `hw/bin/paranut.tcl`, add all newly created files to the corresponding section. Hint: add the results of the HLS (MCsr*.vhd) or any other VHDL files and the two files from step 4 (`mcsr.vhd`, `csr.vhd`).

6. Connect all ports in `hw/rtl/vhdl/paranut.vhd` to each other.