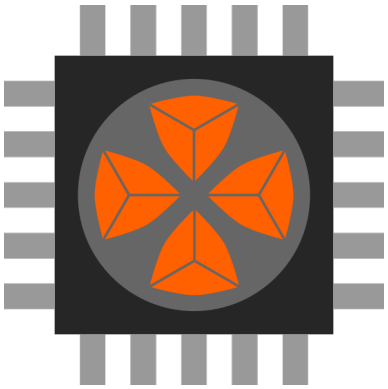
Hochschule
Augsburg University of
Applied Sciences

wird

The *ParaNut* Processor

Architecture Description and Reference Manual



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

Hochschule Augsburg – University of Applied Sciences

gundolf.kiefer@hs-augsburg.de

With contributions by:
Michael Schäferling, Anna Pfützner, Patrick Zacharias, Abdurrahman Celep,
Lukas Bauer

Version: v1.0-114-g23393c6b*

February 15, 2023

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
1.0.0	2022-07-04	Add experimental MMU and Linux support
1.0.47	2022-12-02	Add Rust language setup
1.0.0	2023-01-17	Add Config Creator user manual
1.0.114	2023-02-15	Rewoked CSR Section, added instructions to install GDB and OpenOCD and reworked debugging chapters in appendix

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-

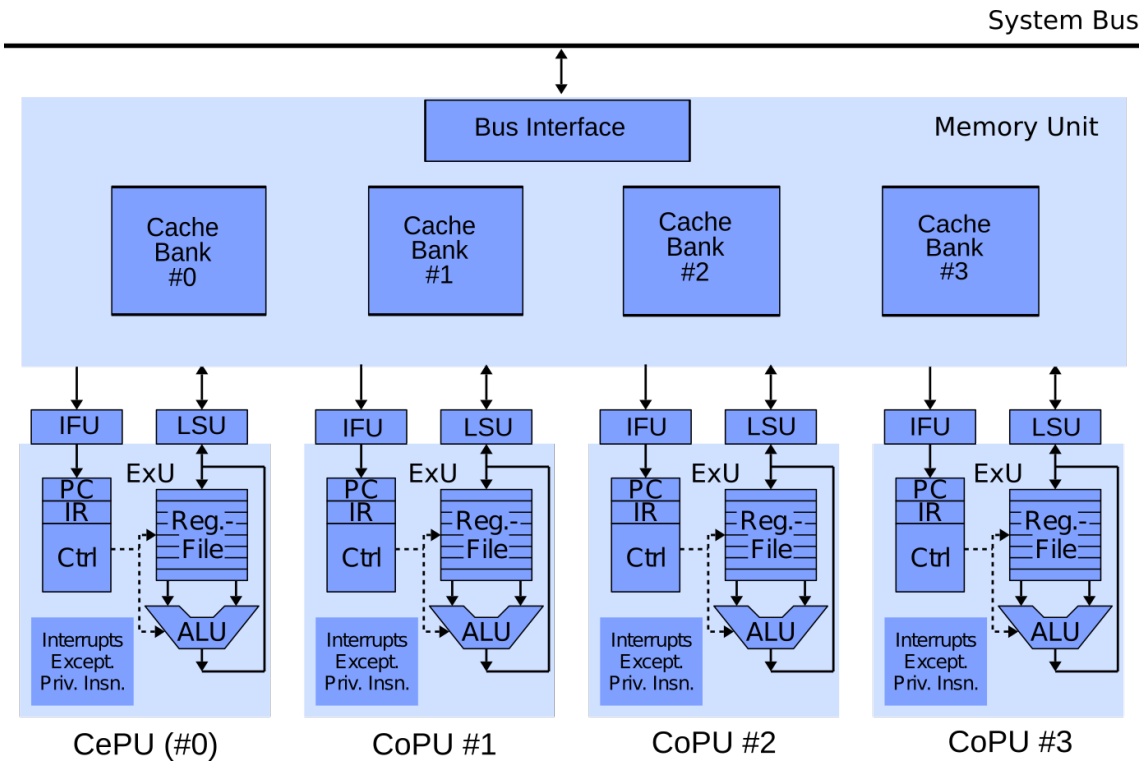


Figure 2.1.: A *ParaNut* instance with 4 cores

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].

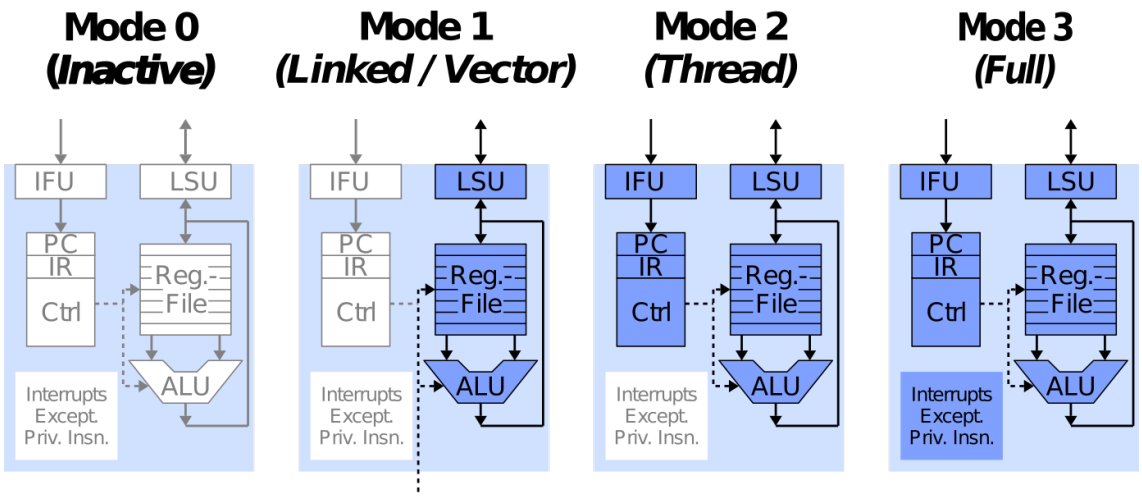


Figure 2.2.: *ParaNut* modes and required logic

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.

```
1  int a[100], b[100], s[100];
2
3  void add_arrays_sequential () {
4    for (n = 0; n < 100; n += 1)
5      s[n] = a[n] + b[n];
6  }
7
8  void add_arrays_parallel () {
9    int n, cpu_no;
10
11   // Activate 3 (=4-1) CoPUs in the "Linked" state and
12   pn_begin_linked (4);
13
14   //   get the number of this CPU...
15   cpu_no = pn_get_cpu_no();
16
17   // performs 4 additions in parallel
18   for (n = 0; n < 100; n += 4)
19     s[n + cpu_no] = a[n + cpu_no] + b[n + cpu_no];
20
21   // End linked mode, deactivate the CoPUs...
22   pn_end_linked ();
23 }
```

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 user-mode exception and interrupt handling is currently not supported. If supervisor mode is configured, a Memory Management Unit (MMU) is available. For more details on the MMU, see [5].

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 3.10 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 3.10 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. Because the address of the registers `mtime`, `mtimeh`, `mtimecmp` and `mtimecmph` are configurable and memory-mapped, a fixed address is not given.

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.
		⋮	
0xB08	MRW	mhpmcounter8	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.
		⋮	
0xB88	MRW	mhpmcounter8	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			
-	MRW	mtime	Machine timer register.
-	MRW	mtimeh	Upper 32 bits of mtime.
-	MRW	mtimecmp	Machine timer compare register.
-	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].

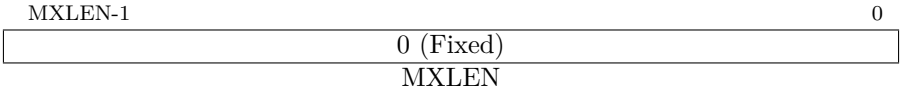


Figure 3.1.: Vendor ID register (`mvendorid`).

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.

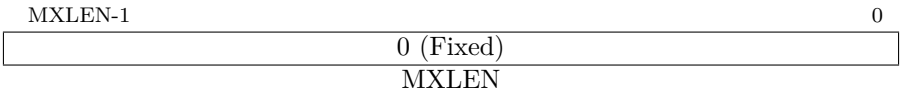


Figure 3.2.: Machine Architecture ID register (`marchid`).

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.

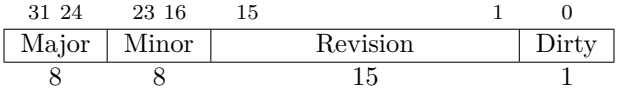


Figure 3.3.: Machine Implementation ID register (`mimpid`).

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.

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

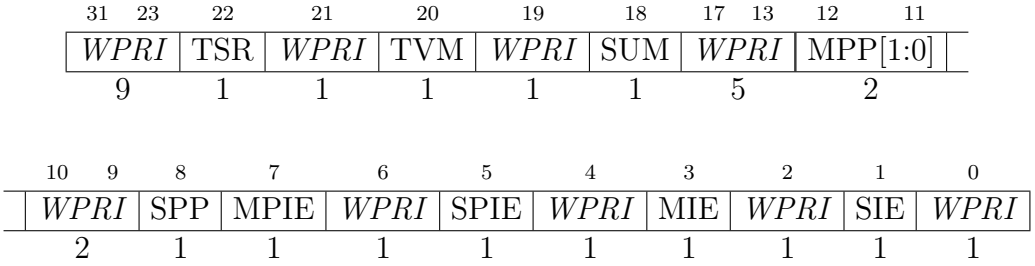


Figure 3.5.: Machine-mode status register (`mstatus`) of the *ParaNut* .

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.

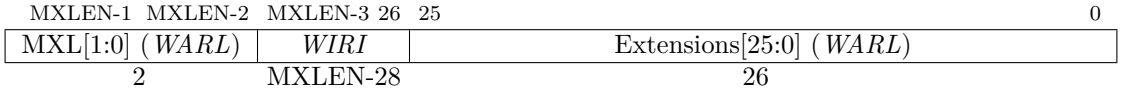


Figure 3.6.: Machine ISA register (`misa`).

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 only the `MTIP` bit of the `mip` register and the `MTIE` bit of the `mie` register are implemented according to [3].

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.

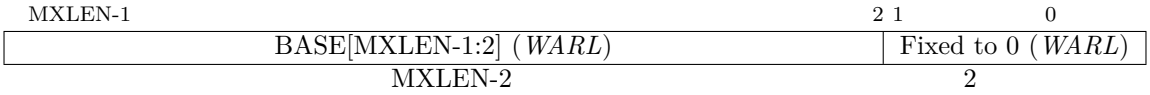


Figure 3.7.: Supervisor trap vector base address register (`stvec`).

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.

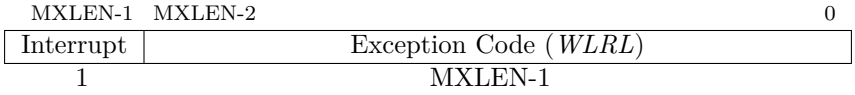


Figure 3.8.: Machine Cause register `mcause`.

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 64-bit `mtimeh` and `mtimecmp` registers are split up into the 32-bit memory mapped registers `mtime` / `mtimeh` and `mtimecmp` / `mtimecmph`. The upper 32-bit of `mtime` and `mtimecmp` are the `mtimeh` and `mtimecmph` registers.

The `mtime/h` registers provide a real-time counter with 64-bit precision running at a constant frequency. The timebase is defined in the `pntimebase` register described in

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	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Not implemented*
1	11	Machine external interrupt
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	Instruction page fault
0	13	Load page fault
0	14	*Not implemented*
0	15	Store/AMO page fault
0	16	ParaNut CoPU exception
0	≥17	*Reserved*

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

Section 3.3.5.17. A 64-bit timer compare register is provided by the `mtimecmp/h` registers. A timer interrupt occurs when the `mtime/h` register contains a value greater than or equal to the value in the `mtimecmp/h` register.

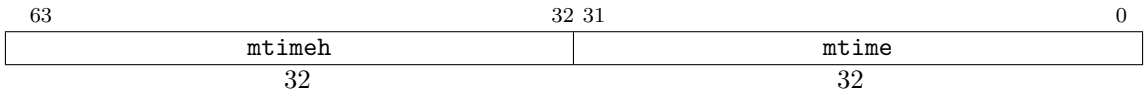


Figure 3.9.: Machine time register (memory-mapped control register).

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.

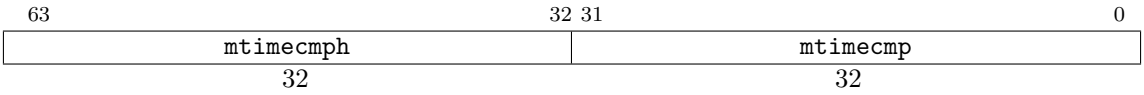


Figure 3.10.: Machine time compare register (memory-mapped control register).

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.

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.
Supervisor Protection and Translation			
`0x180`	SRW	`satp`	Supervisor address translation and protection.

Table 3.8.: Currently allocated supervisor RISC-V CSRs

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

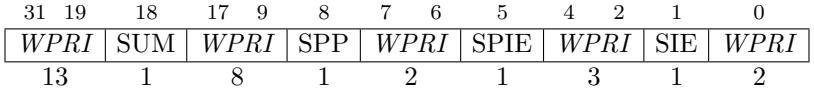


Figure 3.11.: Supervisor-mode status register (`sstatus`) of the *ParaNut* .

3.3.3.2. Supervisor Cause Register (`scause`)

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

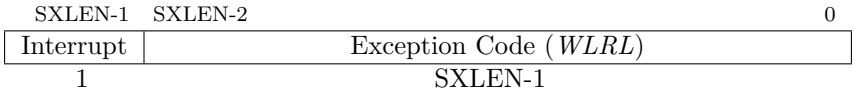


Figure 3.12.: Supervisor Cause register `scause`.

3.3.3.3. Supervisor Address Translation and Protection (`satp`) Register

The field `MODE` enables virtual addressing as explained in [5]. In contrast to [3], field `PPN` is only 20 bits wide.

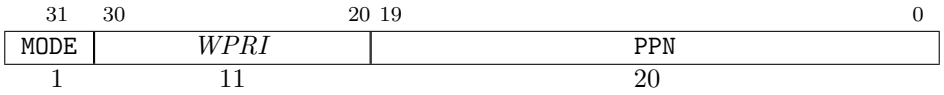


Figure 3.13.: RV32 Supervisor address translation and protection register `satp`.

3.3.4. Unprivileged/User Control and Status Registers

This chapter describes the RISC-V unprivileged(/user)-level Control and Status Registers listed in 3.9, which were originally specified in RISC-V Volume II [3]. Note that these registers are only available when the *ParaNut* was configured to implement user mode. The registers listed in Table 3.9 are the only registers of the user-mode with functunality. The rest of the user mode registers are listed in Chapter 3.3.6 and have no functunality but are implemented to prevent errors when using OpenOCD.
Note: In the SystemC Code the prefix "u" is used for CSR registers of user mode. This was done to prevent a collison between the C function "time" and the CSR "time" while

maintaing a uniform naming convention for all user mode CSRs. When trying to access the registers with assembler instructions the names from table 3.9 or the addresses can be used.

Number	Privilege	Name	Description
Unprivileged/User Counter/Timers RO			
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC80	URO	cycleh	Upper 32 bits of cycle, RV32 only.

Table 3.9.: Currently allocated unprivileged/user RISC-V CSRs

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

3.3.4.1. Cycle Registers (cycle/cycleh)

These registers are shadow registers of mcycle and mcycleh and are a read-only variant of the registers for use in user mode. For a description of the registers see Chapter 3.3.2.11.

3.3.5. *ParaNut* -Specific Control and Status Registers

Table 3.10 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.

3.3.5.1. ParaNut CPU group select (pngrpsel)

The pngrpsel register is an MXLEN-bit read-write register formatted as shown in Figure 3.14. 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.

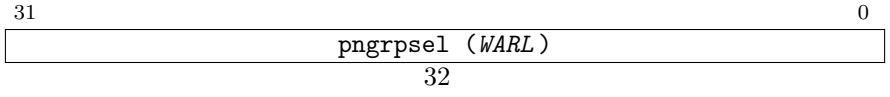


Figure 3.14.: ParaNut CPU group select (pngrpsel).

3.3.5.2. 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.

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)			
0x841	URW	pncause	ParaNut CoPU trap cause ID.
0x842	URW	pnepc	ParaNut CoPU exception program counter.
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.
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.
0xFC8	MRO	pnece	ParaNut exception chip enable.
0xFC9	MRO	pntimebase	ParaNut machine timer timbease.
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 3.10.: Currently allocated *ParaNut* -specific CSRs

Figure 3.15.: Supervisor trap vector base address register (stvec).

3.3.5.3. ParaNut CPU enable register (pnce)

The pnce 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, 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.

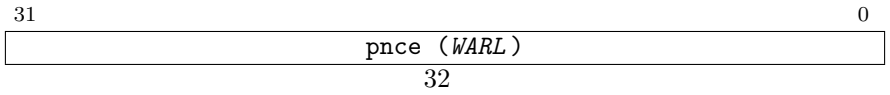


Figure 3.16.: ParaNut CPU enable register (`pnce`).

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

The `pnlm` register is an MXLEN-bit read-write register formatted as shown in Figure 3.17. 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.

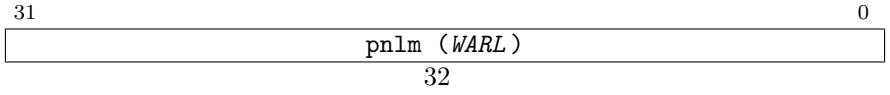


Figure 3.17.: ParaNut CPU linked mode register (`pnlm`).

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

The `pnxsel` register is an MXLEN-bit read-write register formatted as shown in Figure 3.17. 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.

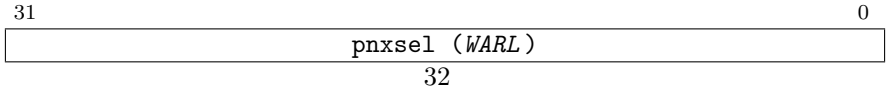


Figure 3.18.: ParaNut CoPU exception select register (`pnxsel`).

3.3.5.6. ParaNut Cache control register (`pncache`)

The `pncache` register is an MXLEN-bit read-write register formatted as shown in Figure 3.19. 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.

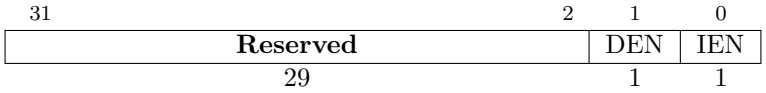


Figure 3.19.: ParaNut Cache control register.

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.

3.3.5.7. ParaNut number of CPUs (pncpus)

The pncpus register is an MXLEN-bit read-only register formatted as shown in Figure 3.20. It holds the number of CPUs (including the CePU).

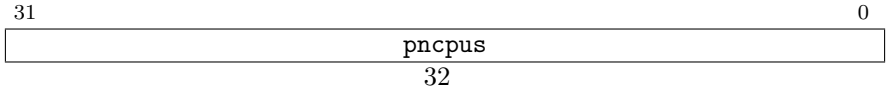


Figure 3.20.: ParaNut number of CPUs (pncpus).

3.3.5.8. ParaNut CPU capabilities register (pnm2cp)

The pnm2cp register is an MXLEN-bit read-only register formatted as shown in Figure 3.21. 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.

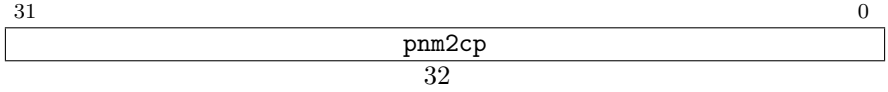


Figure 3.21.: ParaNut CPU capabilities register (pnm2cp.

3.3.5.9. ParaNut CoPU exception pending (pnx)

The pnx register is an MXLEN-bit read-only register formatted as shown in Figure 3.22. 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.

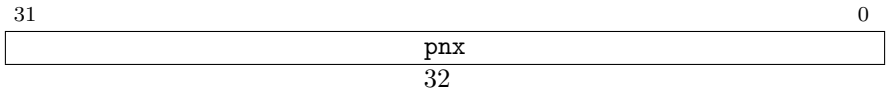


Figure 3.22.: ParaNut CoPU exception pending (`pnx`).

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

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

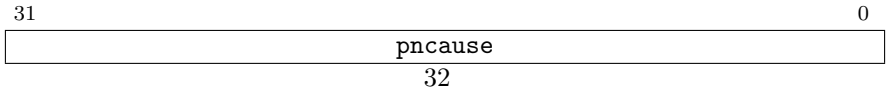


Figure 3.23.: ParaNut CoPU trap cause ID (`pncause`).

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

The `pnepc` register is an MXLEN-bit read-write register formatted as shown in Figure 3.24. It holds the exception program counter of the CoPU selected by `pnxsel` and `pngrpsel`. The CSR only holds legal values as defined in `mepc`.

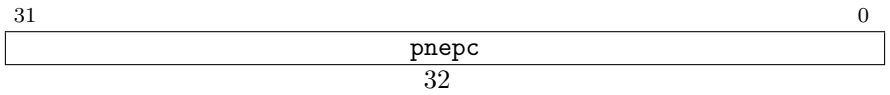


Figure 3.24.: ParaNut CoPU exception program counter (`pnepc`).

3.3.5.12. ParaNut cache information register (`pncacheinfo`)

The `pncacheinfo` register is an MXLEN-bit read-only register formatted as shown in Figure 3.25. It holds information about the cache properties.

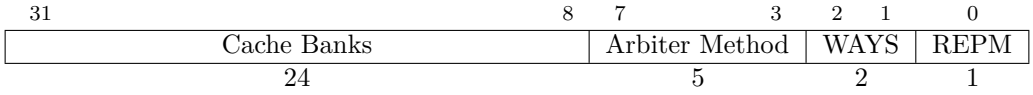


Figure 3.25.: ParaNut cache information register (`pncacheinfo`).

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:
$pncachesets * Cache\ Banks * 4\ Bytes.$

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

The `pncachesets` register is an MXLEN-bit read-only register formatted as shown in Figure 3.26. It holds the number of cache sets.

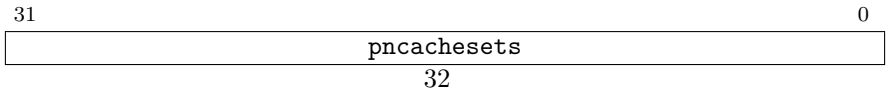


Figure 3.26.: ParaNut number of cache sets register (`pncachesets`).

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

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

The `pnclockinfo` register is an MXLEN-bit read-only register formatted as shown in Figure 3.27. It holds the clock speed in Hz set at compile or synthesis time.

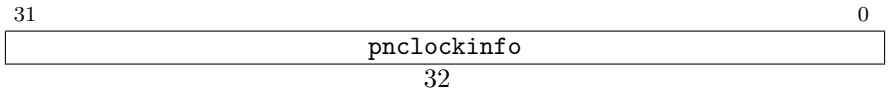


Figure 3.27.: ParaNut clock speed information register (`pnclockinfo`.

3.3.5.15. ParaNut memory size register (`pnmemsize`)

The `pnmemsize` register is an MXLEN-bit read-only register formatted as shown in Figure 3.28. It holds the memory size set at compile or synthesis time.

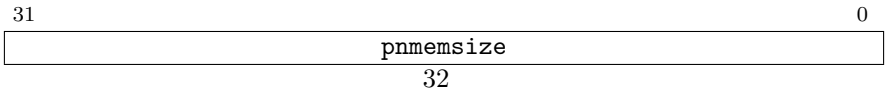


Figure 3.28.: ParaNut memory size register (`pnmemsize`).

3.3.5.16. ParaNut exception CPU enable (`pnece`)

The `pnece` register is an MXLEN-bit read-only register formatted as shown in Figure 3.29. It stores the value of `pnce` before execution of the exception or interrupt process as described in 3.4.

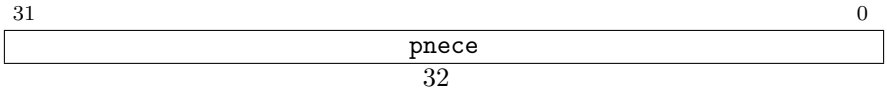


Figure 3.29.: ParaNut exception chip enable (`pnece`).

3.3.5.17. ParaNut machine timer timebase (`pntimebase`)

The `pntimebase` register is an MXLEN-bit read-only register formatted as shown in figure 3.30. It holds the machine timer timebase in µs set at compile or synthesis time.

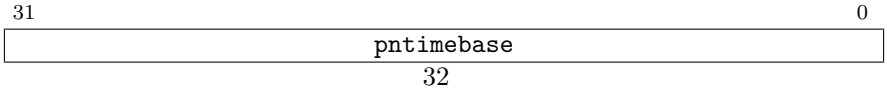


Figure 3.30.: ParaNut machine timer timebase (`pntimebase`).

3.3.5.18. ParaNut core ID register (`pncoreid`)

The `pncoreid` register is an MXLEN-bit read-only register formatted as shown in figure 3.31. It is the only register accesible from CoPUs. This is required to initiate LinkedMode.

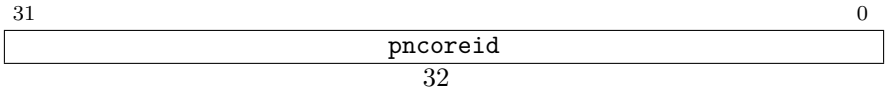


Figure 3.31.: ParaNut core ID register (`pncoreid`).

3.3.6. Control and Status registers without implementation

Table 3.11 shows the registers that where implemented to comply with the RISC-V specification, they don't have any functionality. This was done to prevent errors while using OpenOCD with the *ParaNut* . Reading from these registers will return the value 0x0 and writes to these registers will be ignorred.
Note: That the user mode CSR have the prefix "u" added in the SystemC Code. This was done to prevent a collison between the C function "time" and the CSR "time" while maintaing a uniform naming convention for all user mode CSRs.

Number	Privilege	Name	Description
			Supervisor RW
0x106	SRW	scounteren	Supervisor counter enable.
0x10A	SRW	senvcfg	Supervisor environment configuration register.
0x5A8	SRW	scontext	Supervisor-mode context register.
			Machine R/W
0x306	MRW	mcounteren	Machine counter enable.
0x310	MRW	mstatush	Additional machine status register, RV32 only
0x30A	MRW	menvcfg	Machine environment configuration register.
0x31A	MRW	menvcfgh	Additional machine env. conf. register, RV32 only.
0x320	MRW	mcounterinhibit	Machine counter-inhibit register.
0x323	MRW	mhpmevent3	Machine performance-monitoring event selector.
0x324	MRW	mhpmevent4	Machine performance-monitoring event selector.
		⋮	
0x33F	MRW	mhpmevent31	Machine performance-monitoring event selector.
0x7A0	MRW	tselect	Debug/Trace trigger register select.
0xB09	MRW	mhpmcounter9	Machine performance-monitoring counter.
		⋮	
0xB1F	MRW	mhpmcounter31	Machine performance-monitoring counter.
0xB89	MRW	mhpmcounter9h	Upper 32 bits of mhpmcounter9, RV32 only.
		⋮	
0xB1F	MRW	mhpmcounter31h	Upper 32 bits of mhpmcounter31, RV32 only.
			User RO
0xC01	URO	time	Timer for RDTIME instruction.
0xCD2	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xCD3	URO	hpmcounter3	Performance-monitoring counter.
0xCD4	URO	hpmcounter4	Performance-monitoring counter.
		⋮	
0xC1F	URO	hpmcounter31	Performance-monitoring counter.
0xC81	URO	timeh	Upper 32 bits of time , RV32 only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32 only.
0xC83	URO	hpmcounter3h	Upper 32 bits of uhpmcounter3, RV32 only.
0xC84	URO	hpmcounter4h	Upper 32 bits of uhpmcounter4, RV32 only.
		⋮	
0xC9F	URO	hpmcounter31h	Upper 32 bits of uhpmcounter31, RV32 only.
			Machine RO
0xF15	MRO	mconfigptr	Pointer to configuration data structure .

Table 3.11.: CSRs without functionality

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) for all cores each in their `mepc`
   - The enable ParaNut CPU enable register 3.16 in `pnece`
   - 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`.

> *The change in execution mode in step 2 is visible to the programmer through the* `pnce` *or* `pnlm` *CSRs. Writing to these registers will influence/change the execution mode of the CoPUs immediately.*
> *This allows for simultaneous saving/restoring of the current context and is required for task-switching. In this case the old value can be read from* `pnece` *and saved to* `pnce`*, which will re-enable the CoPUs. No additional shadow register is required for* `pnlm` *as it can be written/read while only the CePU is running.*
> *Please note that this means, that* `pnece` *will need to be saved to* `pnce` *as part of the exception handler to continue execution.*

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`.
6. The exception handler finishes by using the MRET instruction which continues execution at the address saved in `mepc`.

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`.
7. The exception handler finishes by using the MRET instruction which continues execution at the address saved in `mepc`.

If an interrupt occurs the following steps are performed (MIE == 1):

1. The CePU and CoPUs all complete their current instruction.
2. The CePU halts itself and waits for the CoPUs to halt as well (according to the MIE flag set in `mstatus`).
3. The CoPUs halt by respecting the interrupt enable flags as set in the CePU (prior to them being disabled).
4. Execution is continued at the address saved in the `mtvec` register.
5. *Execution of the trap handler*
6. The trap handler finishes by using the MRET instruction which continues execution at the address saved in `mepc`.

If an interrupt occurs the following steps are performed (MIE == 0):

1. The appropriate interrupt pending bit is set.
2. If an exception handler finishes by using the MRET instruction, MPIE is restored and MIE is used accordingly. This means if MIE and the pending bit is set (or the situation which sets the pending bit is unresolved), the trap handler will be entered once more. Otherwise execution at the address saved in `mepc` will continue.

4. Libparanut

5. Operating Environments

5.1. Bare Metal and newlib

5.2. FreeRTOS

5.3. Linux

Linux as one of the most common operating system kernels is already partially starting on the *ParaNut* . It uses *OpenSBI* as a bootloader. A pre-configured setup can be found at `sw/linux`.

Before compiling the project, make sure that the `config.mk` file of the *ParaNut* has the correct values set. The A and M extensions (`CFG_EXU_M_EXTENSION`, `CFG_EXU_A_EXTENSION`) have to be enabled and the number of privilege levels (`CFG_PRIV_LEVELS`) must be set to 3. The TLB (`CFG_MMU_TLB_ENABLE`) can be enabled for faster execution.

Running

```
$ cd sw/linux
```

```
$ make
```

clones the *Linux* Kernel and *OpenSBI* as well as downloads a compiler toolchain to the `external` folder. Note that either `PARANUT_TOOLS` or directly `PN_EXTERNAL` (to the `external` folder) has to be set. It then copies a prepared *Linux* Config inside and compiles a very simple version of the *Linux* Kernel. A device tree `paranut.dts` is generated from the template `paranut-template.dts` by filling in values from the config, including the memory address or the `mtimer` location and frequency. The device tree as well as the Kernel is combined with *OpenSBI* and compiled into a single ELF file.

To run it in the simulator, use

```
$ make sim
```

You will first see the *OpenSBI* boot messages appear before the *Linux* Kernel begins to start. Note that this process might take a very long time in the simulator.

For running in hardware, a separate refdesign is provided at `systems/linux`.

5.4. Rust

Author: Abdurrahman Celep

5.4.1. Why Rust with Paranut?

The Rust language is a modern and growing programming language, which is similar to C and C++. In contrast to other programming languages, memory safety is guaranteed. Rust provides security features that focus on preventing program errors that lead to memory access errors or buffer overflows. To accomplish this, it uses a borrow checker to validate references. Without garbage collection, Rust can guarantee memory safety and has optional reference counting. This makes Rust more robust against security vulnerabilities introduced in programming.

With the help of this new language, it is possible to write programs for the ParaNut project that can guarantee reliable code and secure memory access.

5.4.2. Working with Rust

To work with the programming language, it is important to look at Rust Cross-Compilation for RISC-V. The main aspects are: how `rustc` works and what configurations are necessary to target RISC-V when compiling Rust programs.

The support platforms, also called targets, are separated in three tiers. Each of the tiers has a different set of guarantees. The ParaNut uses the riscv32i-unknown-none-elf platform, which is currently in Tier 2. It means that target and toolchain are available and will build guaranteed.

More information about the platforms can be found on the website [7].

5.4.3. Build the Project

For the ParaNut processor, the target riscv32i-unknown-non-elf is needed. To use the platform in a Rust project, target support for RISC-V must be added with:

```
1 $ rustup target add riscv32i-unknown-none-elf
```

The target support is also needed in the nightly channel.

```
1 $ rustup +nightly target add riscv32i-unknown-none-elf
```

There are two ways to build Rust project. It is possible to work with and without a Cargo environment.

5.4.3.1. Working without Cargo Enviroment

Working without Cargo Environment is an option to build a small Rust project, which only creates an elf file and gives up all features of the Cargo Environment like build, run, doc, clean, etc.

Configuration

The first step is to create a Rustfile.

```
$ touch main.rs
```

More details about the main.rs file, can be found in Chapter 5.4.4.

To compile main.rs and get the elf file, it is important to give the rustc compiler flags:

```
$ rustc +nightly --target=riscv32i-unknown-none-elf\\
 --extern=$(PATH_TO_COMPILER_BUILTINS)
 -Clink-args=$(C_FLAGS) -C linker=riscv64-unknown-elf-gcc\\
 main.rs -o $(elf_data_name)
```

With the `-extern` flag an extern crate will be included in the Rust project. The crate `compier-builts` is needed to fix remaining link errors. Usually the path for the `compiler-builtins` is .cargo/registry/src/github.com*/compiler_builtins-*. Please note, that the nightly channel is needed for this crate.

5.4.3.2. Working with Cargo Enviroment

Cargo is a package manager for Rust, which makes development much easier. The package manager downloads the Rust package's dependencies, compiles them, creates distributable packages, and uploads them to creates.io, the Rust community's package registry.[8]

Configuration

First, a project must be created:

```
$ cargo new hello_rust
```

It will give the following directory structure:

```
rusty-risc
├── Cargo.toml
└── src
    └── main.rs
```

In Chapter 5.4.4 main.rs is presented in more details.

To use the RISC-V toolchain, the Cargo Environment has to be configured. For the configuration a direcory named **.cargo** containing a file named **config.toml** has to be created.

```
rusty-risc
├── .cargo
│   └── config.toml
├── Cargo.toml
└── src
    └── main.rs
```

The **config.toml** contains the configuration for the package manager. When a project is being built, cargo searches all parent directories and the current directory for configuration files.[9]

For the ParaNut project the **config.toml** file looks like:

```
[target.riscv32i-unknown-none-elf]
runner = $(RISCV_SIMULATOR)
rustflags = [
... // $(C_FLAGS)
]

linker = "riscv64-unknown-elf-gcc"

[build]
target = "riscv32i-unknown-none-elf"
```

To use the **riscv32i-unknown-none-elf** platform support, the default target has to be overwritten in the [**build**] table. This is followed by setting up the platform target in table [**target.riscv32i-unknown-none-elf**]. Inside the table the linker, runner and Rustflags can be defined.

The extern crate `compiler-builtins` have to be include in **Cargo.toml**.

```
[dependencies]
compiler-builtins = <VERSION_NUMBER>
```

Now the Rust project can be built with:

```
$ cargo +nightly build
```

After the build, the elf file will be shown in **target/riscv32i-unknown-none-elf/debug/<elf-file>** and it can be run with a RISC-V simulator. In case of the ParaNut project use **pn-sim**.

5.4.4. The main Program Example

This is what main.rs should look like

```
#![no_std]
#![no_main]

...

[no_mangle]
pub extern fn main(_nargs: i32, _args: *const *const u8) -> i32 {

  /* inside the main function*/

  // Exit with a return status of 0.
  0
}
```

It is important to note that the program will run on bare metal, so the standard Rust library and standard main function should not be used [10]. They can be disabled with **#![no_std]** and **#![no_main]**. The **no_mangle** means, that any function exported by Rust that is used outside of Rust must be instructed not be mangled by the compiler. This is because the Rust compiler mangles symbol names differently than native code linkers expect [11]. In the next step, a custom main function is defined, whose return value is zero.

To define the behavior of **panic!** in a **no_std** application, it is important to create a **panic_handler**. This ensures that when problems occur, the system is kept in a loop.[12]

```
use core::panic::PanicInfo;

...

#[panic_handler]
fn panic(_panic: &PanicInfo<'_>) -> ! {
  loop {}
}
```

5.4.4.1. Print out Hello ParaNut

To get a terminal output, a special crate is needed (**bindgen_stdio_crate**). For this purpose an internal crate was created, which can be found in the folder **/sw/intern_rust_crate**. Please note that this folder contains only crates that are not published on the crate.io website. (If you want create new internal crates, feel free to do it in this directory) Also please be sure that **bindgen** is also installed. (See more about bindgen in [13])

It should be mentioned that, for rust projects with and without cargo, the integration of crates is handled differently.

5.4.4.2. Working without Cargo Enviroment

In this example only an extra flag (–**extern**) is added to the command to tell the compiler that a crate is being used.

```
    $ rustc --target=riscv32i-unknown-none-elf\\
    --extern=stdio=libbindgen_stdio_sys.rlib\\
    -Clink-args=$(C_FLAGS) -C linker=riscv64-unknown-elf-gcc\\
    main.rs -o $(elf_data_name)
```

Please note, that the **rlib**-Data has to be generated by compiling the crate. (An example can be found on the Website [14])

The main.rs would look like this:

```
...
pub extern fn main(_nargs: i32, _args: *const *const u8) -> i32 {

  stdio::printf(b"Hello ParaNut!\n\0" as *const u8 as *const i8);

  ...
}
```

The **use** command to work with crates is not needed.

5.4.4.3. Working with Cargo Enviroment

To be able to work with the crate, the desired crates must be written in the Cargo.toml.

```
[dependencies]
bindgen-stdio-sys = {path = "../rust_intern_crates/bindgen-stdio-sys"}
```

Under the **dependencies** the crate **bindgen_stdio_crate** is defined, which is located in the path **/rust_intern_crates/bindgen-stdio-sys**. After that, the crate can be used in the **main.rs**.

```
use bindgen_stdio_sys as stdio;

...

pub extern fn main(_nargs: i32, _args: *const *const u8) -> i32 {
  stdio::printf(b"Hello ParaNut!\n\n\0" as *const u8 as *const i8);
  ...
}
```

5.4.4.4. Using the Simulator

The generated elf file can be run in the simulator **pn_sim**. Information on building and running the simulator can be found in Chapter A.1.1. Finally, a HelloParaNut should appear in the terminal.

```
/*without Cargo Enviroment*/

$ $PARANUT_HOME/hw/sim/pn-sim <elf file>

/*with Cargo Enviroment*/

$ $PARANUT_HOME/hw/sim/pn-sim /target/riscv32i-unknown-none-elf/debug/<elf file
   >
}
```

6. Tools

6.1. *ParaNut* : Config Creator - User Manual

The Config creator is a tool that allows the user to modify/create the configuration file of the ParaNut processor.

Additional features:

- Saving presets of the configuration
- Approximating resource usage

6.1.1. Starting page

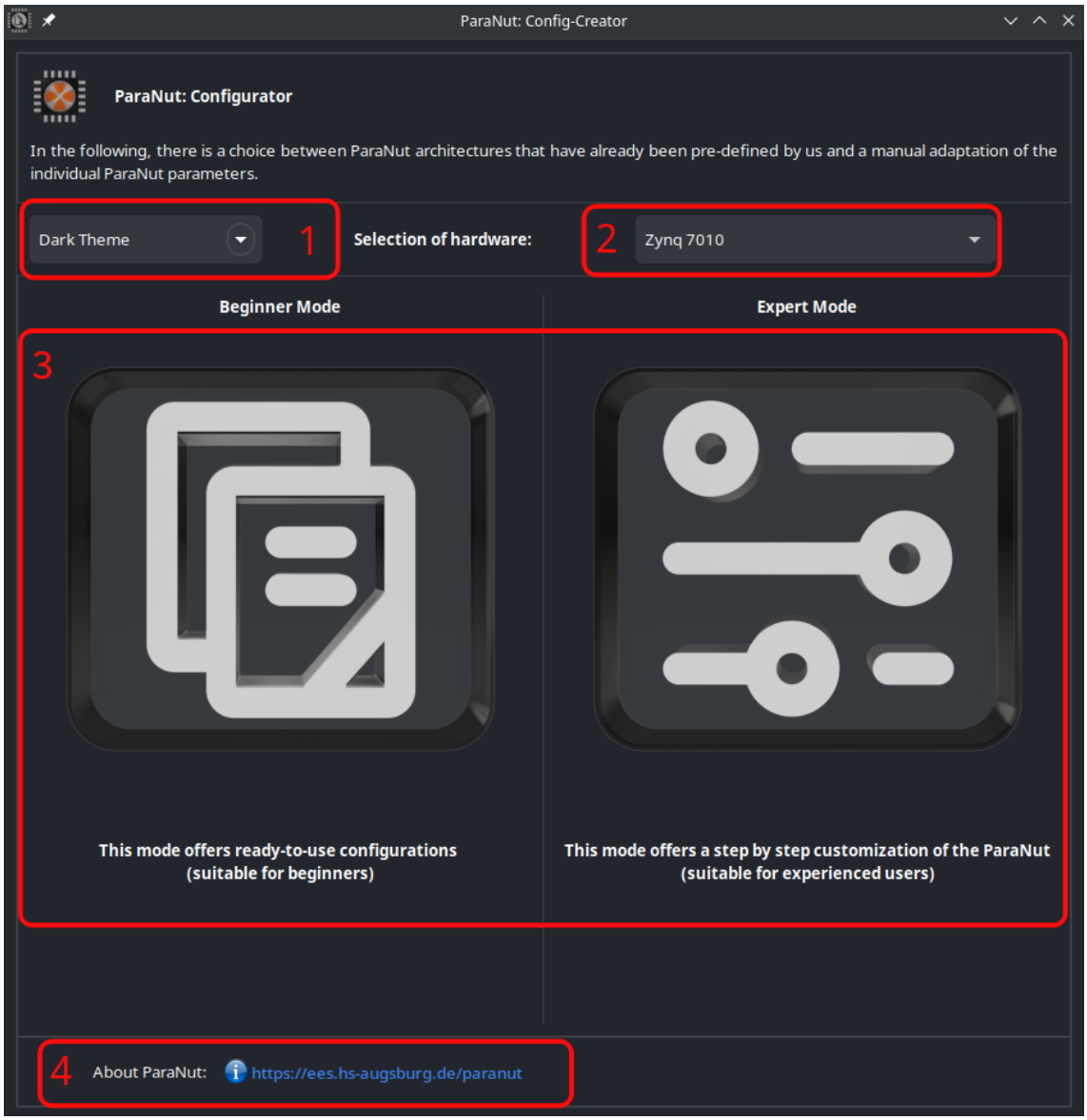


Figure 6.1.: Screenshot of the starting page

1. Select desired theme. You can choose between light and dark theme.

2. Select your desired hardware to continue with the configuration progress.

3. Choose between predefined configurations and a manual configuration of the ParaNut. Beginner mode: Contains a list of already „ready-to-use“ configurations and your saved presets. Expert mode: Contains a step by step configuration process, where every module can be adjusted by certain parameters.

4. These are links leading to an overview article of the ParaNut project from the embedded world exposition in 2020 and to the EES-ParaNut project GitHub.

6.1.2. Beginner mode

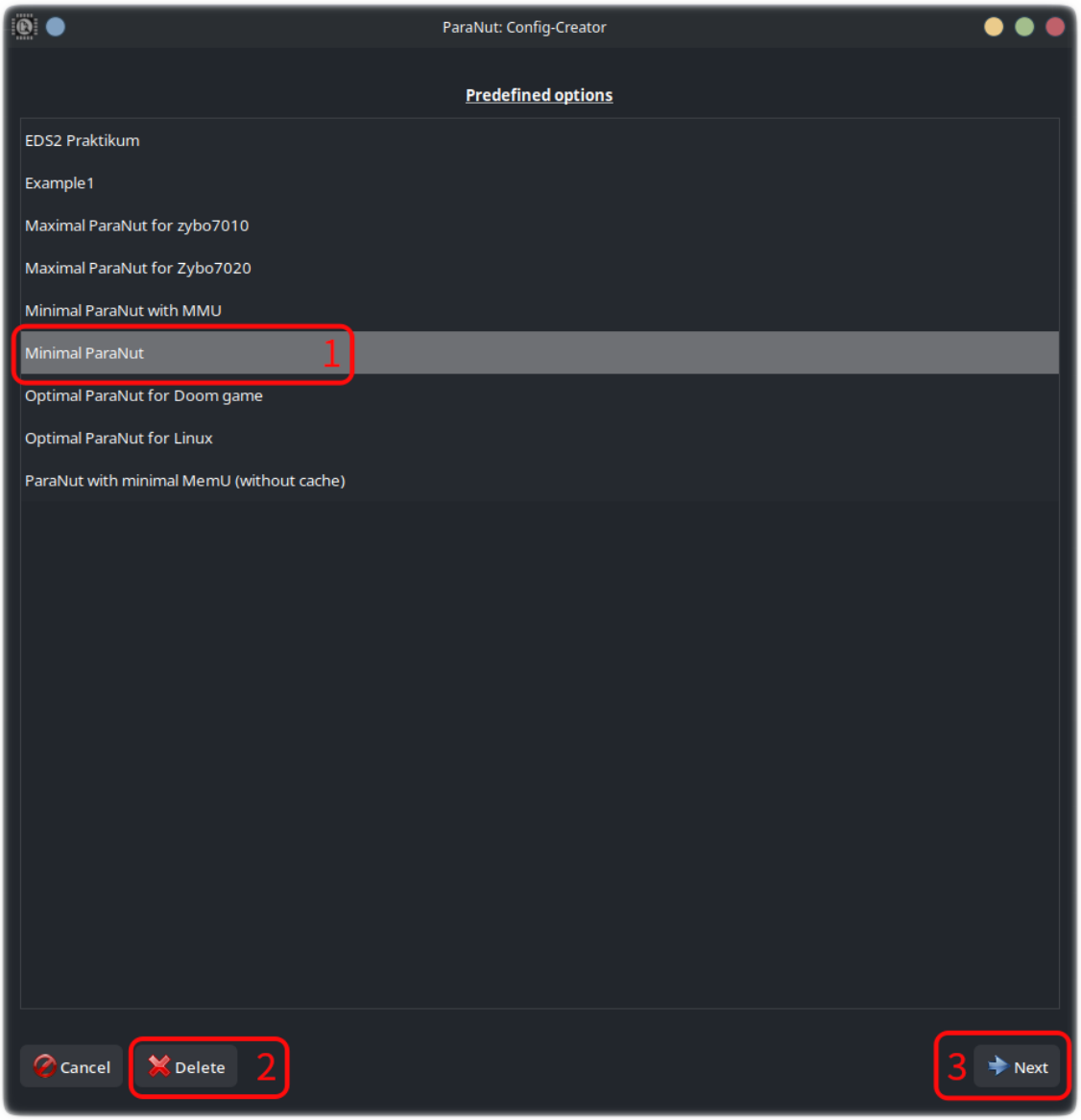


Figure 6.2.: Screenshot of the beginner mode page

1. Each section represents either a predefined system or one of your saved presets.
2. Pressing the „Cancel“ button will redirect you to the starting page.

6.1.3. Expert mode

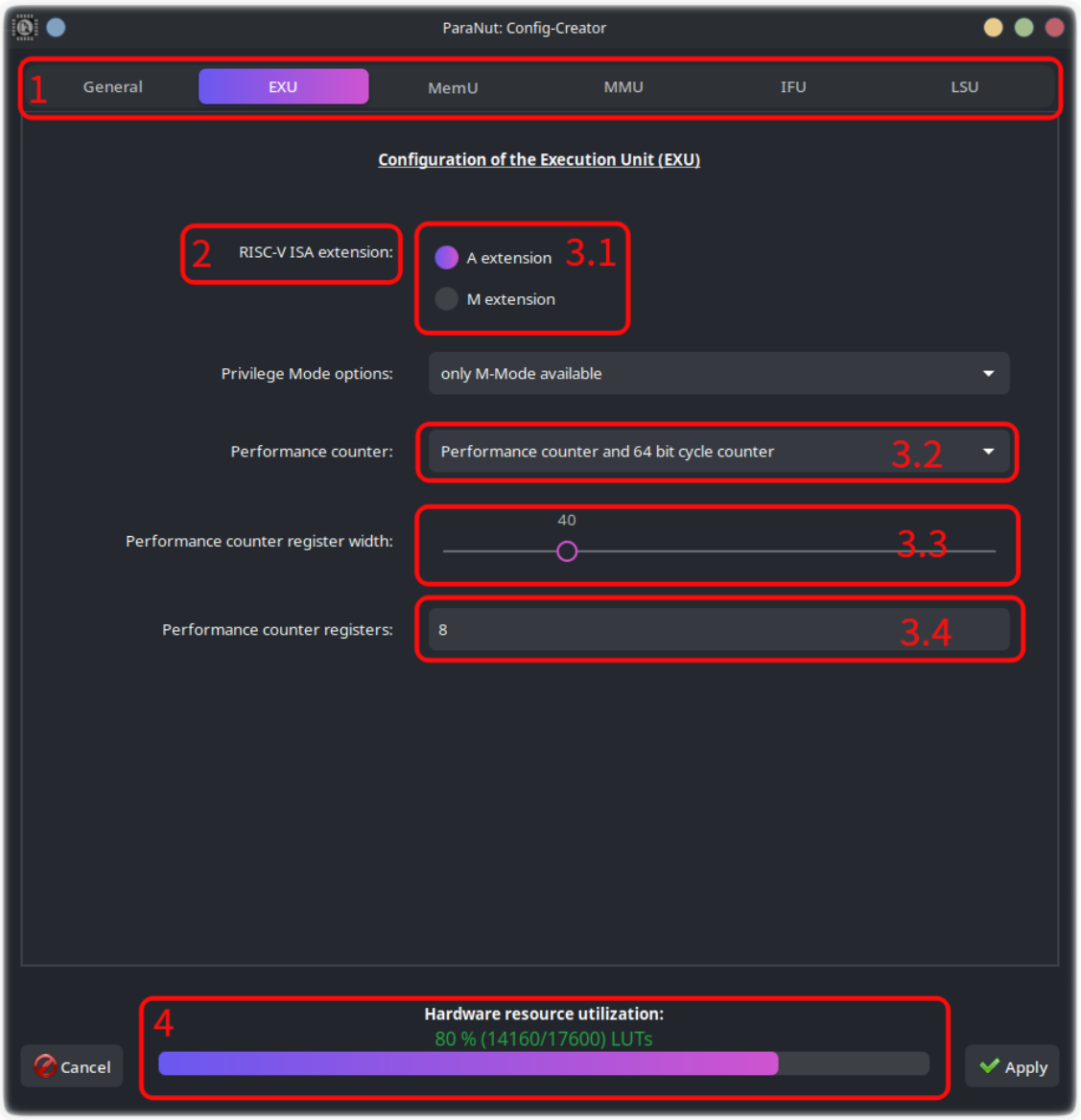


Figure 6.3.: Screenshot of the expert mode page

1. The manual configuration page has a navigation bar, which is used to jump between the individual modules.
2. Each module has its own parameters which reveal tips and explanations when hovering over them.
3. The configuration process consists of three different input types:
   1 Checkboxes, which decide whether content should be added.
   2 Comboboxes, which have drop-downs containing the different options to choose from.

3 Slider, which allows to select an accurate value.

4 Simple integer entries, resembling the desired value of the associated parameter. (invalid entries will return warnings or errors.)

4. The hardware resource utilization progress bar resembles the used hardware resource capacity relative to its maximum resources.

   Pressing the „Apply “ button will commit your entered values, while jumping to the overview page. Pressing the „Cancel “ button will terminate the manual configuration process and you will be sent to the starting page.

6.1.4. Overview page

This page is the last stop before the finished configuration file. On this page all your manually selected or predefined options will be summarized, giving you the chance to make changes.

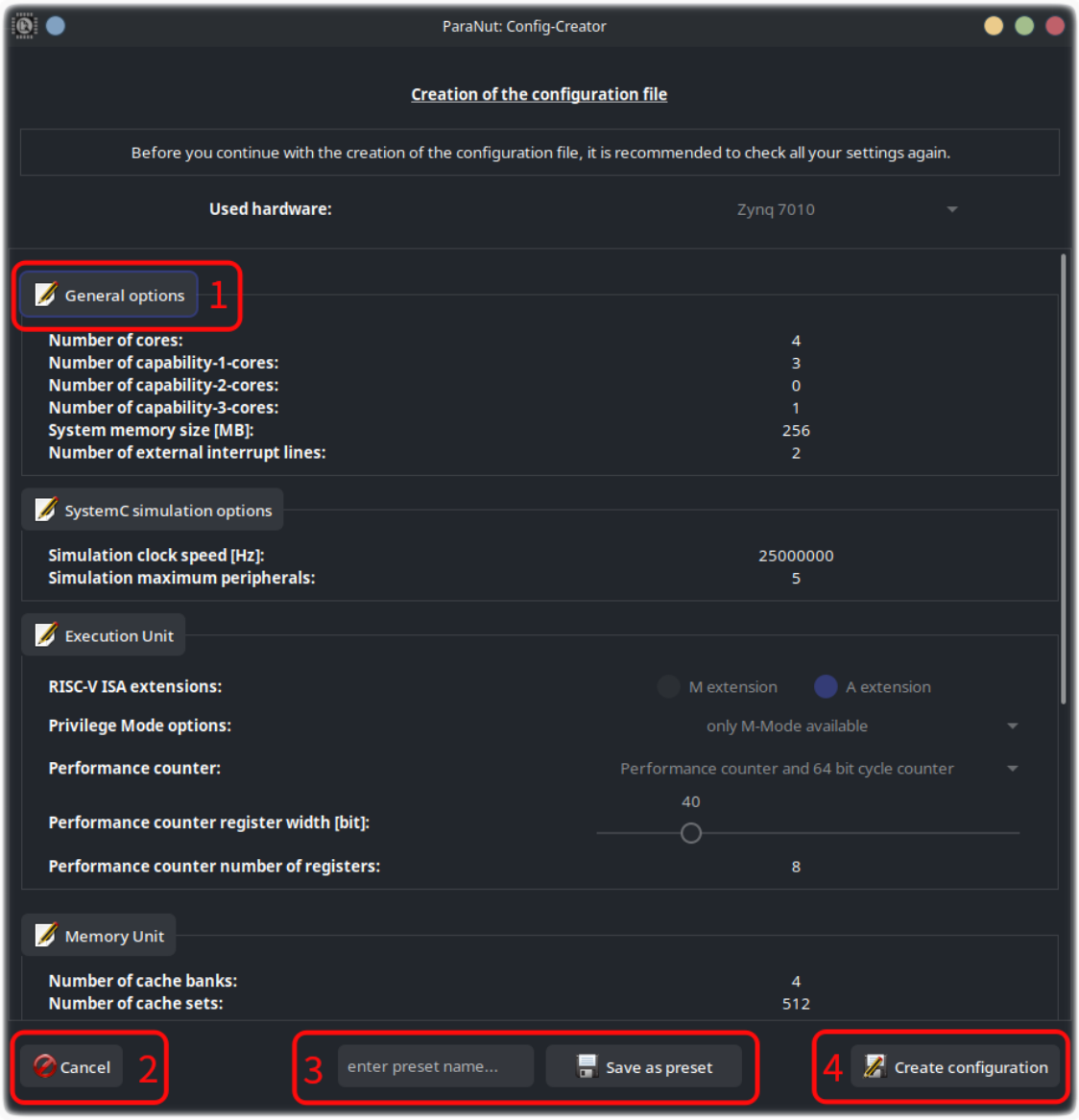


Figure 6.4.: Screenshot of the overview page

1. Each different module has ist own section with an „Edit“ button, which leads to the corresponding module page in the manual configuration.
2. The „Cancel“ button will lead you back to the starting page.
3. Your configuration will be saved as a preset. From now on it will be shown in your Beginner mode section of the tool.
4. Your configuration will be saved as a finished config.mk file on your computer.

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

[5] Christian H. Meyer, "A Memory Management Unit for the ParaNut", 2022

[6] "The rustup book: 2.Concepts: 2.1.Channels", https://rust-lang.github.io/rustup/concepts/channels.html

[7] "The rustc book: 6.Platfrom Support", https://doc.rust-lang.org/rustc/platform-support.html

[8] "The Cargo book", https://doc.rust-lang.org/cargo/

[9] "The Cargo book: 3.Cargo Reference: 3.6.Configuration", https://doc.rust-lang.org/cargo/reference/config.html

[10] "Embedonomicon: 1. The smallest #![no_std] program", https://docs.rust-embedded.org/embedonomicon/smallest-no-std.html

[11] "The Embedded Rust Book: 10.Interoperability: 10.2.A little Rust with your C", https://docs.rust-embedded.org/book/interoperability/rust-with-c.html

[12] "The Rustonomicon: 12.Beneath std: 12.1.#[panic_handler]", https://doc.rust-lang.org/nomicon/panic-handler.html

[13] "crate.io: bindgen",https://crates.io/crates/bindgen

[14] "Geeksforgeeks: Rust - Creating a Library", https://www.geeksforgeeks.org/rust-creating-a-library/

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

```
# 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
```

```
19
20 ELF = hello_newlib
21 SOURCES = $(wildcard *.c)
22 OBJECTS = $(patsubst %.c,%.o,$(SOURCES))
23 HEADERS = $(wildcard *.h)
24
25 PN_SYSTEMS_DIR = $(PARANUT)/systems
26 RISCV_COMMON_DIR = $(PARANUT)/sw/riscv_common
27
28 CFG_MARCH ?= rv32i
29
30 CFLAGS = -O2 -march=$(CFG_MARCH) -mabi=ilp32 -I$(RISCV_COMMON_DIR)
31 LDFLAGS = $(CFLAGS) -static -nostartfiles -lc $(RISCV_COMMON_DIR)/startup.S $(
       RISCV_COMMON_DIR)/syscalls.c -T $(RISCV_COMMON_DIR)/paranut.ld
32
33 # Software Targets
34 all: $(ELF) dump
35
36 $(ELF): $(OBJECTS)
37 $(CC) -o $@ $^ $(LDFLAGS)
38
39 %.o: %.c $(HEADERS)
40 $(CC) -c $(CFLAGS) $<
41
42
43 # ParaNut Targets
44 .PHONY: sim
45 sim: $(ELF)
46 +$(MAKE) -C $(PARANUT)/hw/sim pn-sim
47 $(PARANUT)/hw/sim/pn-sim -t0 $<
48
49 # Generic Flash targets (set PN_* accordingly)
50 .PHONY: flash flash-bit
51 flash: bin
52 pn-flash -c -p $(ELF).bin $(PN_SYSTEM_HDF) $(PN_FIRMWARE_ELF)
53
54 flash-bit: bin
55 pn-flash -c -b $(PN_SYSTEM_BIT) -p $(ELF).bin $(PN_SYSTEM_HDF) $(
       PN_FIRMWARE_ELF)
56
57
58 # Special System Flash targets for testing inside the source repository
59 .PHONY: flash-%
60 flash-%: bin
61 if [ ! -d $(PN_SYSTEMS_DIR) ]; then echo; echo "INFO: The flash targets are
       only for testing inside the source repository!"; echo; exit 1; fi
62 pn-flash -c -p $(ELF).bin $(PN_SYSTEMS_DIR)/$*/hardware/build/system.hdf $(
       PN_SYSTEMS_DIR)/$*/hardware/firmware/firmware.elf
63
64 flash-%-bit: bin
```

```
65 if [ ! -d $(PN_SYSTEMS_DIR) ]; then echo "INFO: The flash targets are only for
       testing inside the source repository!"; exit 1; fi
66 pn-flash -c -b $(PN_SYSTEMS_DIR)/$*/hardware/build/system.bit -p $(ELF) \
67 $(PN_SYSTEMS_DIR)/$*/hardware/build/system.hdf $(PN_SYSTEMS_DIR)/$*/hardware/
       firmware/firmware.elf
68
69
70 # Misc Targets
71 .PHONY: dump
72 dump: $(ELF).dump
73 $(ELF).dump: $(ELF)
74 $(OBJDUMP) -S -D $< > $@
75
76 .PHONY: bin
77 bin: $(ELF).bin
78 $(ELF).bin: $(ELF)
79 $(OBJCOPY) -S -O binary $< $@
80
81 .PHONY: clean
82 clean:
83 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. Installing GDB

This chapter explains how to install a version of GDB in which the text user interface(tui) is enabled.

Prerequisites:

- libncurses-dev to be able to compile gdb with tui
- The packages listed here: https://github.com/riscv-collab/riscv-gnu-toolchain
- a symbolic link python that points to python3

Information:

- Buildtime with the -j4 option: 22 minutes
- Size required for the repository: 11 GB
- Size required for installed Toolchain: 1.4 GB
- Installs the complete riscv-gnu-toolchain an the multilib libraray

To do this follow these instructions:
Install libncurses.

```
$ sudo apt install libncurses-dev
```

If the "python" command is not available on your system you need to create a symbolic link named "python" pointing to the "python3" executable.

Download the source code from the GitHub repository.

```
$ git clone https://github.com/riscv/riscv-gnu-toolchain
```

Change directory into the newly downloaded repository.

```
$ cd riscv-gnu-toolchain
```

Checkout the Version "2023.01.04" with the following command

```
$ git checkout 2023.01.04
```

Configuring the toolchain with the following command (remove the \from the command):

```
$ ./configure --prefix=/home/<username>/toolchain \

--with-multilib-generator="rv32i-ilp32--;rv32im-ilp32--;rv32ima-ilp32--"
```

Note that the given prefix path is only a sugestion and can be changed by the user.
Compile and install the toolchain.

```
$ make
```

Now you have installed a complete toolchain from which only GDB is needed. To only use the GDB from the compiled toolchain you can rename the GDB version from the old toolchain and create a symbolic link that points to the new version in its place. Alternatifly you can call this version of GDB by it's full path.

A.3. Installing OpenOCD

This chapter explains how to install a OpenOCD version which supports the switching of memory access modes.
Information:

- Buildtime with the -j4 option: 1 minute
- Size required for the repository: 227 MB
- Size required for installed Toolchain: 23 MB
- Installs OpenOCD in Version 0.12.0

To do this follow these instructions:
Download the source code from the GitHub repository.

```
$ git clone https://git.code.sf.net/p/openocd/code openocd-code
```

Change directory into the newly downloaded repository.
```
$ cd openocd-code
```

Checkout the Version "v0.12.0" with the following command
```
$ git checkout v0.12.0
```

Configuring the toolchain with the following commands:
```
$ ./bootstrap
```

```
$ ./configure --prefix=/home/<username>/openocd --enable-remote-bitbang
```
Note that the given prefix path is only a sugestion and can be changed by the user.

Compile and install the toolchain.
```
$ make
```

```
$ make install
```

To use the newly installed version of OpenOCD you need to add the folder of the executable to the systems path variable. You can either add, the following command, to your .bashrc or run it before using OpenOCD.
```
$ export PATH="<Path to OpenOCD bin folder>:$PATH"
```

A.4. 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:

Run the command for the local repository from the root of the repository.
e.g. local repository for hello_newlib:
```
$ /<Path to Reposetory>/hw/sim/pn-sim -d <Path to Executable>/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`*

Run the command for the local repository from the root of the repository.
local repository:

```
$ openocd -f /<Path to Reposetory>/tools/etc/openocd-sim.cfg
```

installed:

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

> *To prevent error messages regarding csr register access please uncomment the `riscv expose_csrs` line for the right privivleg level. In the file `openocd-sim.cfg` located in tools/etc of the* ParaNut *repository*

> *To use OpenOCD in a more efficent manner please install OpenOCD 0.12.0 as discribed in A.3 and uncomment the following line `riscv set_mem_access abstract progbuf` in the `openocd-sim.cfg` file. Located in the folder tools/etc of the* ParaNut *repository.*

In yet another shell start the RISC-V GDB debug session:
For this you need to change into the directory containing the executable.
e.g for hello_newlib:

```
$ cd /<Path to Reposetory>/sw/hello_newlib
```

run gdb for hello_newlib:

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

Then connect to OpenOCD as remote target:

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

Load the register configuration for the current privilege level:
Replace the x with the privilege level configured in config.mk
Note: This path to the file is given for the case that gdb was started in the root of the repository. If GDB was not started in that folder you need to ajust the path

```
(gdb) set tdesc filename /<Path to Reposetory>/tools/etc/gdb_csr_priv<x>.xml
```

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
```

Additionally when you installed GDB as discribed in A.2 then you can use commands like the following to switch into the tui mode:

```
(gdb) layout regs
```

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.5. 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`*

Run the command for the local repository from the root of the repository.

local repository:
```
$ openocd -f /<Path to Reposetory>/tools/etc/openocd-board.cfg
```

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

> *To prevent error messages regarding csr register access please uncomment the* `riscv expose_csrs` *line for the right privivleg level. In the file* `openocd-board.cfg` *in tools/etc of the* ParaNut *repository.*

> *To use OpenOCD in a more efficent manner please install OpenOCD 0.12.0 as discribed in A.3 and uncomment the following line* `riscv set_mem_access abstract progbuf` *in the* `openocd-board.cfg` *file. Located in the folder tools/etx of the* ParaNut *repository.*

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

eg for hello_newlib

```
$ riscv64-unknown-elf-gdb /<Path to Reposetory>/refdesign/software/hello_newlib
```

Lastly connect to OpenOCD as remote target:

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

Load the register configuration for the current privilege level:
Replace the x with the privilege level configured in config.mk located in the root of the repository.
Note: This path to the file is given for the case that gdb was started in the root of the repository. If GDB was not started in tht folder you need to ajust the path
E.g:

```
(gdb) set tdesc filename /<Path to Reposetory>/tools/etc/gdb_csr_priv<x>.xml
```

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
```

Additionally when you installed GDB as discribed in A.2 then you can use commands like the following to switch into the tui mode:

```
(gdb) layout regs
```

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.6. 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.6.1. AXI compatible modules

A.6.2. 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.