Hochschule
Augsburg University of
Applied Sciences

The *ParaNut* Processor

Architecture Description and Reference Manual

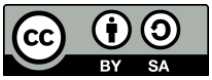
Gundolf Kiefer, Alexander Bahle

Hochschule Augsburg – University of Applied Sciences

gundolf.kiefer@hs-augsburg.de

Version 0.3.0

February 19, 2020



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

Contents

1. Introduction

The goal of the *ParaNut* project is to develop an open, scalable and practically usable 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-

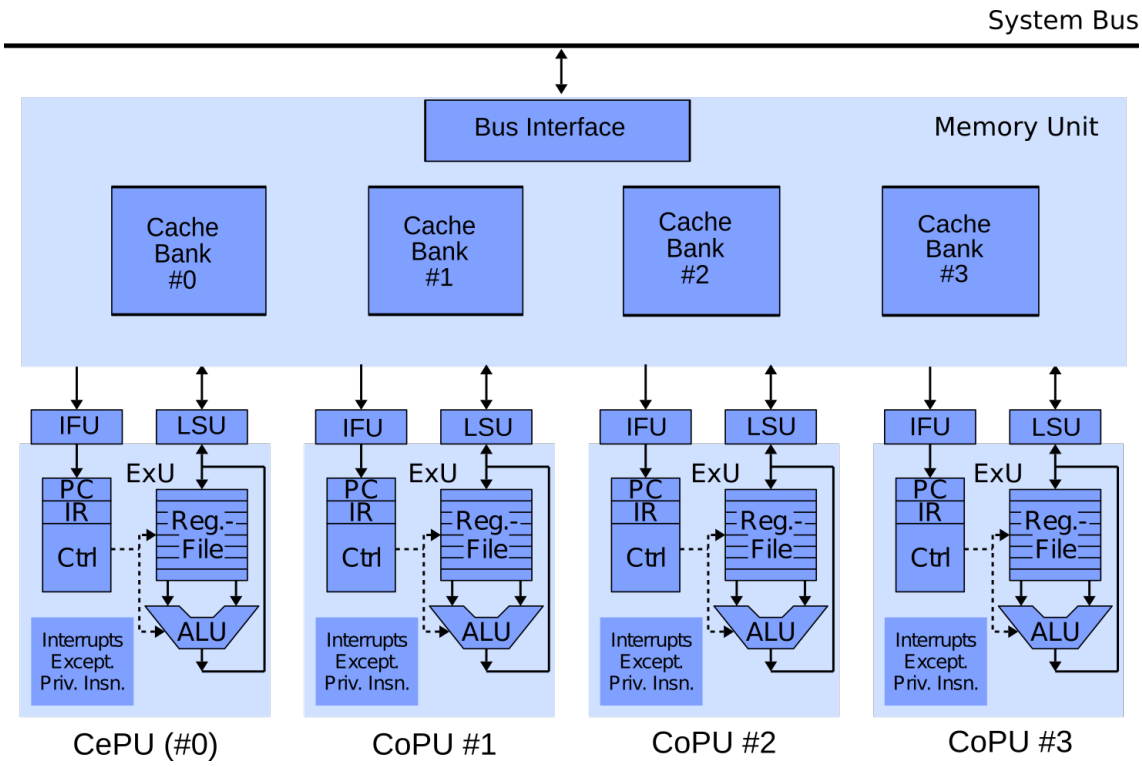


Figure 2.1.: A *ParaNut* instance with 4 cores

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].

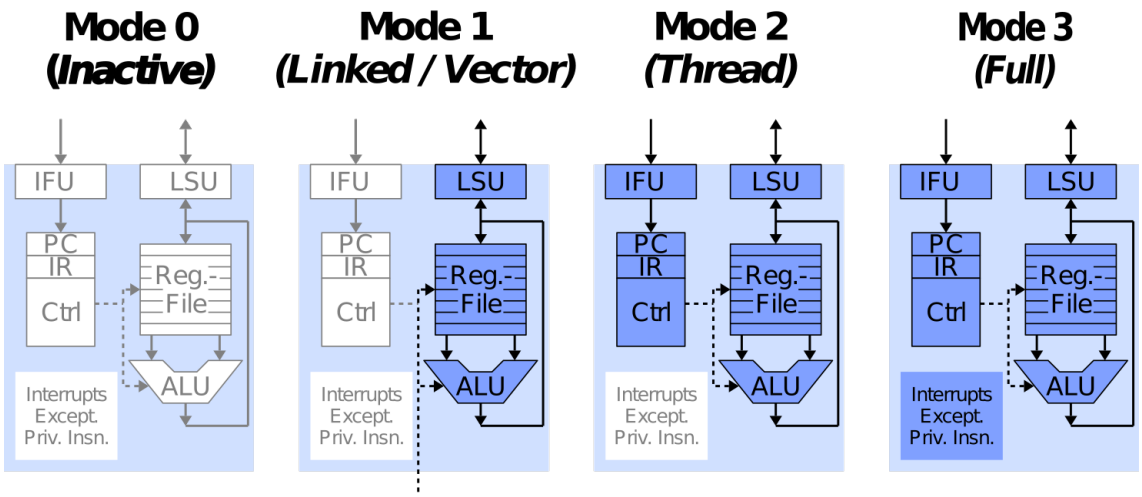


Figure 2.2.: *ParaNut* modes and required logic

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 classical 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 complete instruction set reference for the *ParaNut* architecture. For completeness, the descriptions of the RISC-V (RV32I) instructions and registers supported by a *ParaNut* processor have been copied from the RISC-V Instruction Set Manual Volume I [2]. Clarifications and deviations from the RISC-V specification are captured as comments in the following sections.
"A component is termed a *core* if it contains an independent instruction fetch unit. A RISC-V-compatible core might support multiple RISC-V-compatible hardware threads, or *harts*, through multithreading." [3] In context of the *ParaNut* architecture this means that Mode 3 and Mode 2 capable CPUs are *cores* as well as *harts* whereas Mode 1 CPUs can only be referred to as *harts*.

3.1. Base Instruction Formats

In the base RV32I ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 3.1. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction-address-misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. This exception is reported on the branch or jump instruction, not on the target instruction. No instruction-address-misaligned exception is generated for a conditional branch that is not taken.

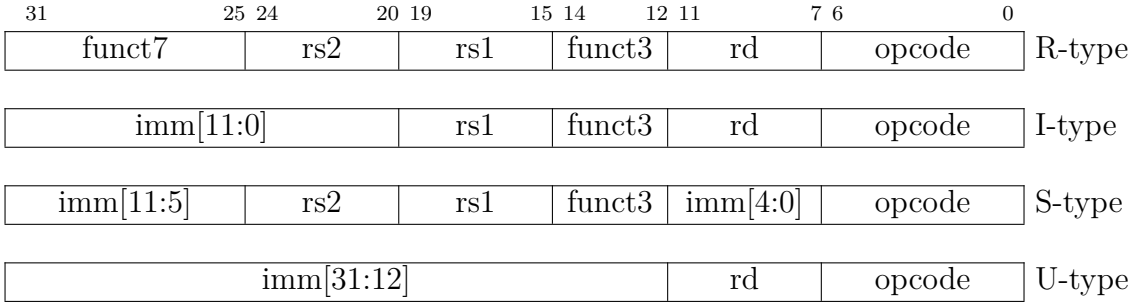


Figure 3.1.: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

The RISC-V ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions (Chapter **??**), immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

3.2. Immediate Encoding Variants

There are a further two variants of the instruction formats (B/J) based on the handling of immediates, as shown in Figure 3.2.

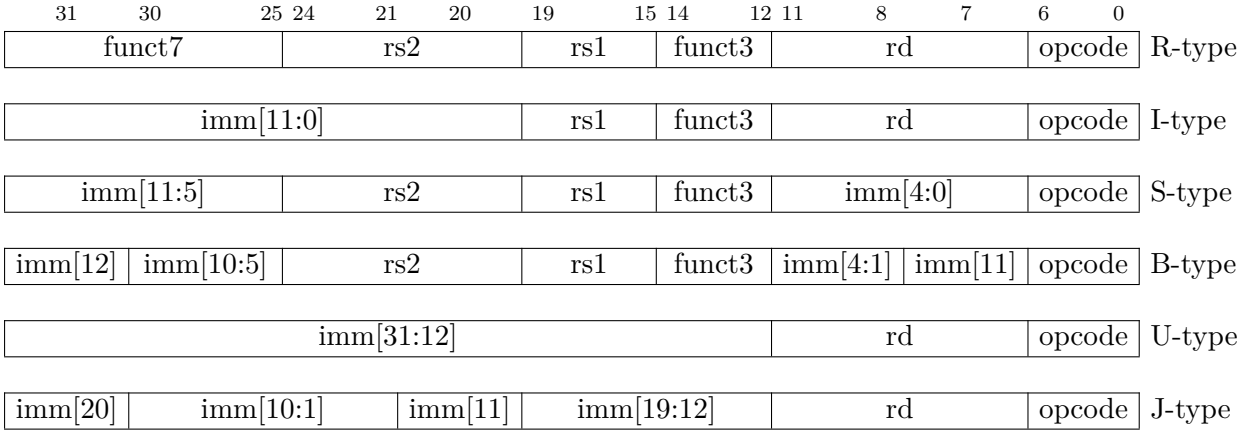


Figure 3.2.: RISC-V base instruction formats showing immediate variants.

The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.

Similarly, the only difference between the U and J formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

Figure 3.3 shows the immediates produced by each of the base instruction formats, and is labeled to show which instruction bit (inst[y]) produces each bit of the immediate value.

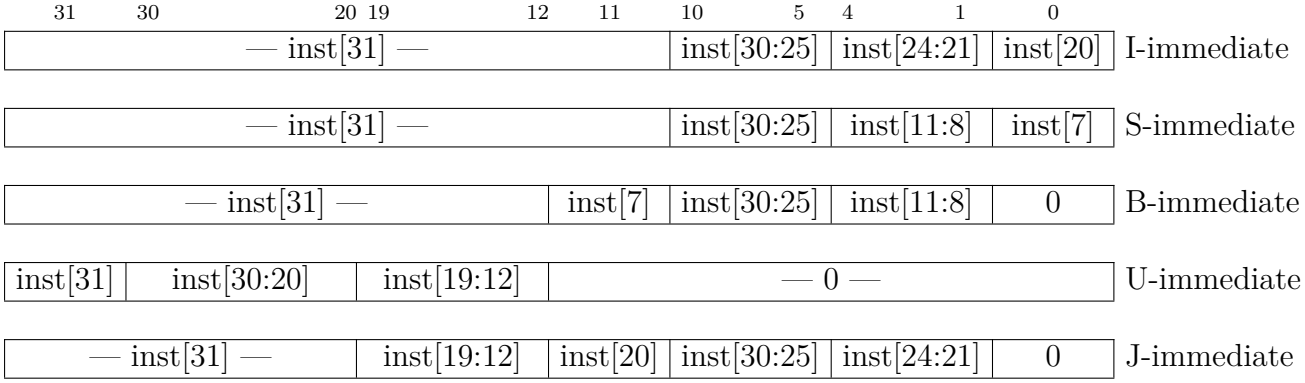


Figure 3.3.: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

3.3. Instructions

3.3.1. Integer Computational Instructions

Integer Register-Immediate Instructions

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

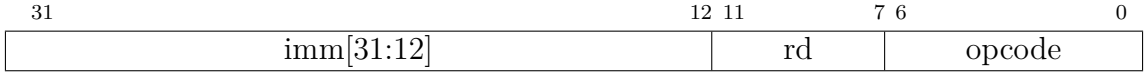
ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd, rs1, 0* is used to implement the MV *rd, rs1* assembler pseudoinstruction.

SLTI (set less than immediate) places the value 1 in register *rd* if register *rs1* is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to *rd*. SLTIU is similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to XLEN bits then treated as an unsigned number). Note, SLTIU *rd, rs1, 1* sets *rd* to 1 if *rs1* equals zero, otherwise sets *rd* to 0 (assembler pseudoinstruction SEQZ *rd, rs*).

ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*. Note, XORI *rd, rs1, -1* performs a bitwise logical inversion of register *rs1* (assembler pseudoinstruction NOT *rd, rs*).

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right shift type is encoded in bit 30. SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).



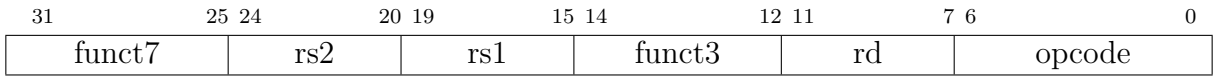
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.

AUIPC (add upper immediate to `pc`) is used to build `pc`-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the `pc` of the AUIPC instruction, then places the result in register *rd*.

Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.

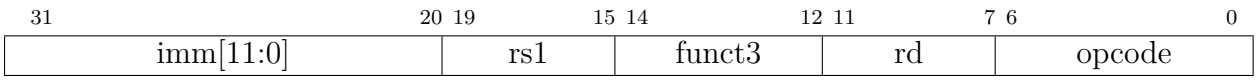


7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, *x0*, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd, rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

NOP Instruction



12	5	3	5	7
0	0	ADDI	0	OP-IMM

The NOP instruction does not change any architecturally visible state, except for advancing the `pc` and incrementing any applicable performance counters. NOP is encoded as ADDI *x0, x0, 0*.

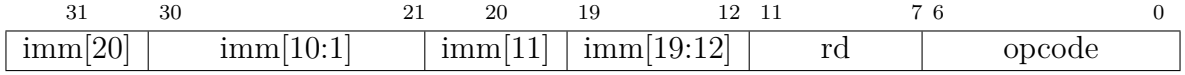
3.3.2. Control Transfer Instructions

RV32I provides two types of control transfer instructions: unconditional jumps and conditional branches. Control transfer instructions in RV32I do *not* have architecturally visible delay slots.

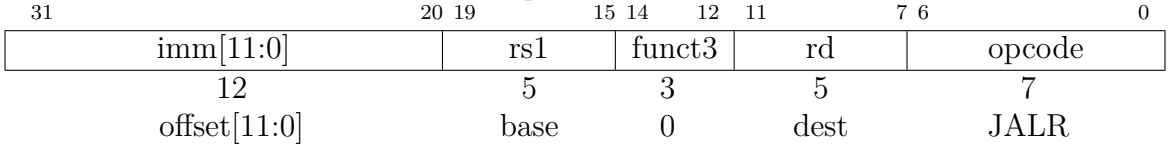
Unconditional Jumps

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the `pc` to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (`pc`+4) into register *rd*. The standard software calling convention uses `x1` as the return address register and `x5` as an alternate link register.

Plain unconditional jumps (assembler pseudoinstruction J) are encoded as a JAL with *rd*=`x0`.

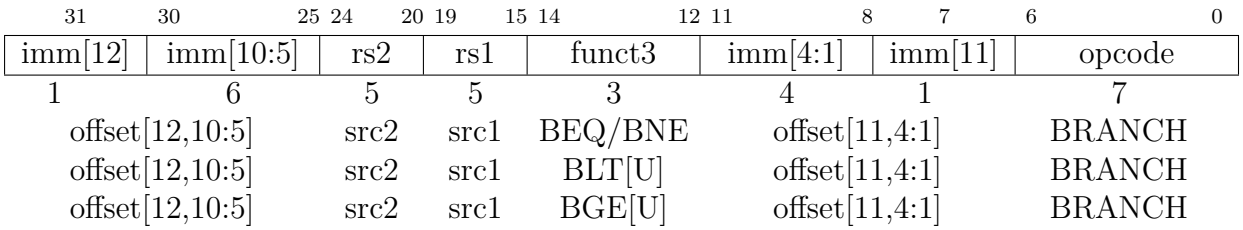


The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register *rs1*, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (`pc`+4) is written to register *rd*. Register `x0` can be used as the destination if the result is not required.



Conditional Branches

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2, and is added to the current `pc` to give the target address. The conditional branch range is ±4 KiB.



Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

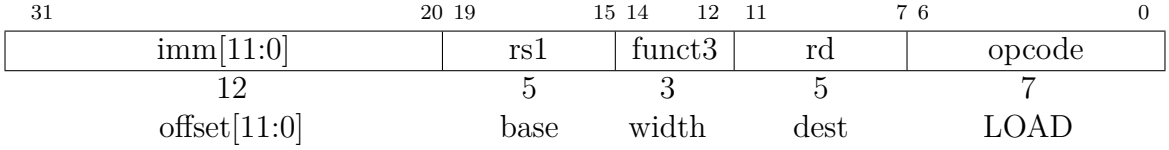
Software should be optimized such that the sequential code path is the most common path, with less-frequently taken code paths placed out of line. Software should also assume that backward branches will be predicted taken and forward branches as not taken, at least the first time they are encountered. Dynamic predictors should quickly learn any predictable branch behavior.

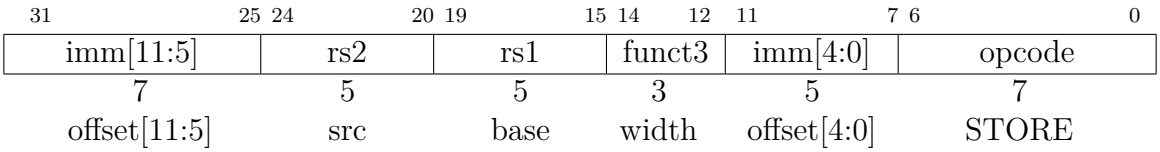
The present ParaNut *implementation does not feature any branch prediction, branches as well as jumps stall the instruction fetch until the condition and/or address is evaluated.*

Unlike some other architectures, the RISC-V jump (JAL with *rd*=`x0`) instruction should always be used for unconditional branches instead of a conditional branch instruction with an always-true condition. RISC-V jumps are also PC-relative and support a much wider offset range than branches, and will not pollute conditional-branch prediction tables.

3.3.3. Load and Store Instructions

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. RV32I provides a 32-bit address space that is byte-addressed and little-endian. The execution environment will define what portions of the address space are legal to access with which instructions (e.g., some addresses might be read only, or support word access only). Loads with a destination of `x0` must still raise any exceptions and action any other side effects even though the load value is discarded.





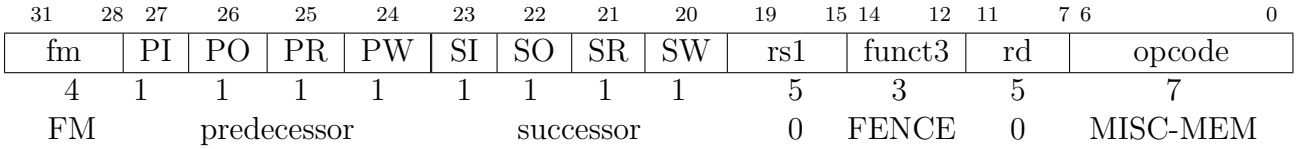
Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective byte address is obtained by adding register *rs1* to the sign-extended 12-bit offset. Loads copy a value from memory to register *rd*. Stores copy the value in register *rs2* to memory.

The LW instruction loads a 32-bit value from memory into *rd*. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in *rd*. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in *rd*. LB and LBU are defined analogously for 8-bit values. The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register *rs2* to memory.

For best performance, the effective address for all loads and stores should be naturally aligned for each data type (i.e., on a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses). The base ISA supports misaligned accesses, but these might run extremely slowly depending on the implementation. Furthermore, naturally aligned loads and stores are guaranteed to execute atomically, whereas misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.

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 won't cause any change in memory. Misaligned loads won't change the value of* rd.

3.3.4. Memory Ordering Instructions



The FENCE instruction is used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the *successor* set following a FENCE before any operation in the *predecessor* set preceding the FENCE.

The execution environment will define what I/O operations are possible, and in particular, which memory addresses when accessed by load and store instructions will be treated and ordered as device input and device output operations respectively rather than memory reads and writes. For example, memory-mapped I/O devices will typically be accessed with uncached loads and stores that are ordered using the I and O bits rather than the R and W bits. Instruction-set extensions might also describe new coprocessor I/O instructions that will also be ordered using the I and O bits in a FENCE.

The ParaNut *processor operates in order and the write buffer of the Load Store Units is emptied in order 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 one must use the special cache control instructions found in section 3.3.8.*

3.3.5. 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.

CSR Instructions

All CSR instructions atomically read-modify-write a single CSR, whose CSR specifier is encoded in the 12-bit *csr* field of the instruction held in bits 31–20. The immediate forms use a 5-bit zero-extended immediate encoded in the *rs1* field.

csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register *rd*. The initial value in *rs1* is written to the CSR. If *rd*=`x0`, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

The CSRRS (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).

The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected.

For both CSRRS and CSRRC, if *rs1*=`x0`, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs. Both CSRRS and CSRRC always read the addressed CSR and cause any read side effects regardless of *rs1* and *rd* fields. Note that if *rs1* specifies a register holding a zero value other than `x0`, the instruction will still attempt to write the unmodified value back to the CSR and will cause any attendant side effects. A CSRRW with *rs1*=`x0` will attempt to write zero to the destination CSR.

The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the *rs1* field instead of a value from an integer register. For CSRRSI and CSRRCI, if the uimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write. For CSRRWI, if *rd*=`x0`, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read. Both CSRRSI and CSRRCI will always read the CSR and cause any read side-effects regardless of *rd* and *rs1* fields.

Register operand				
Instruction	rd	rs1	read CSR?	write CSR?
CSRRW	`x0`	-	no	yes
CSRRW	`!x0`	-	yes	yes
CSRRS/C	-	`x0`	yes	no
CSRRS/C	-	`!x0`	yes	yes
Immediate operand				
Instruction	rd	uimm	read CSR?	write CSR?
CSRRWI	`x0`	-	no	yes
CSRRWI	`!x0`	-	yes	yes
CSRRS/CI	-	0	yes	no
CSRRS/CI	-	!0	yes	yes

Table 3.1.: Table showing whether a CSR instruction reads or writes a given CSR. The CSRRS and CSRRC instructions have same behavior so are shown as CSRRS/C in Table.

Table 3.1 summarizes the behavior of the CSR instructions with respect to whether they read and/or write the CSR.

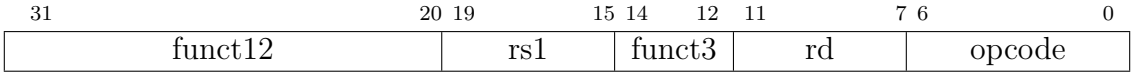
Some CSRs, such as the instructions-retired counter, `instret`, may be modified as side effects of instruction execution. In these cases, if a CSR access instruction reads a CSR, it reads the value prior to the execution of the instruction. If a CSR access instruction writes such a CSR, the write is done instead of the increment. In particular, a value written to `instret` by one instruction will be the value read by the following instruction.

The assembler pseudoinstruction to read a CSR, CSRR *rd, csr*, is encoded as CSRRS *rd, csr, x0*. The assembler pseudoinstruction to write a CSR, CSRW *csr, rs1*, is encoded as CSRRW *x0, csr, rs1*, while CSRWI *csr, uimm*, is encoded as CSRRWI *x0, csr, uimm*.

Further assembler pseudoinstructions are defined to set and clear bits in the CSR when the old value is not required: CSRS/CSRC *csr, rs1*; CSRSI/CSRCI *csr, uimm*.

ParaNut *does not raise an illegal instruction exception on reads or writes to non-existent CSRs (eg.* sval*). Instead a value of 0 is returned or the write is ignored. Only writes to read-only implemented CSRs (see Chapter 3.4) will cause an illegal instruction exception.*

3.3.6. Environment Call and Breakpoints



12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

There two instructions cause a precise requested trap to the supporting execution environment.

The ECALL instruction is used to make a service request to the execution environment. The execution environment will define how parameters for the service request are passed, but usually these will be in defined locations in the integer register file.

The EBREAK instruction is used to return control to a debugging environment.

3.3.7. Trap-Return Instructions

Instructions to return from trap are encoded under the PRIV minor opcode.

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
MRET/SRET/URET	0	PRIV	0	SYSTEM

To return after handling a trap, there are separate trap return instructions per privilege level: MRET, SRET, and URET. MRET is always provided. SRET must be provided if supervisor mode is supported, and should raise an illegal instruction exception otherwise. URET is only provided if user-mode traps are supported, and should raise an illegal instruction otherwise.

ParaNut *does not implement U- or S-mode so only MRET is supported.*

3.3.8. *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

The HALT instruction halts the current CPU by changing 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.

3.4. Control and Status Registers (CSR)

The control and status registers defined by the RISC-V architecture and supported by the *ParaNut* architecture are listed in Table 3.3. The addresses used are defined in the RISC-V Privileged Architecture Instruction Set Manual. All registers are 32 bits wide from software perspective. The *CoPU* column specifies if the register is readable in a *CoPU* (modes 1 and 2). CoPUs supporting mode 3 implement the same registers as CePUs.

Presently, a protected user mode is not defined. Illegal accesses according to the tables do not generate exceptions. They are either ignored (write accesses) or may return senseless data (read accesses).

The descriptions, tables and figures are copied from the RISC-V privileged ISA [3]. Clarifications or deviations from the specification are added as comments.

3.4.1. Privilege Levels

At any time, a RISC-V hardware thread (*hart*) is running at some privilege level encoded as a mode in one or more CSRs (control and status registers). Three RISC-V privilege levels are currently defined as shown in Table 3.2.

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

Table 3.2.: RISC-V privilege levels.

Privilege levels are used to provide protection between different components of the software stack, and attempts to perform operations not permitted by the current privilege mode will cause an exception to be raised. These exceptions will normally cause traps into an underlying execution environment.

The machine level has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine-mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. M-mode can be used to manage secure execution environments on RISC-V. User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and operating system usage respectively.

ParaNut *currently only implements the M-mode privilege level.*

Number	Privilege	CoPU	Name	Description
Machine Information Registers				
0xF11	MRO		mvendorid	Vendor ID.
0xF12	MRO		marchid	Architecture ID.
0xF13	MRO		mimpid	Implementation ID.
0xF14	MRO	X	mhartid	Hardware thread ID.
Machine Trap Setup				
0x300	MRW		mstatus	Machine status register.
0x301	MRO		misa	ISA and extensions
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 3.3.: Currently allocated standard RISC-V CSRs

3.4.2. CSR Field Specifications

The following definitions and abbreviations are used in specifying the behavior of fields within the CSRs.

Reserved Writes Preserve Values, Reads Ignore Values (WPRI)

Some whole read/write fields are reserved for future use. Software should ignore the values read from these fields, and should preserve the values held in these fields when writing values to other fields of the same register.

For forward compatibility, implementations that do not furnish these fields must hardwire them to zero. These fields are labeled **WPRI** in the register descriptions.

Write/Read Only Legal Values (WLRL)

Some read/write CSR fields specify behavior for only a subset of possible bit encodings, with other bit encodings reserved. Software should not write anything other than legal values to such a field, and should not assume a read will return a legal value unless the last write was of a legal value, or the register has not been written since another operation (e.g., reset) set the register to a legal value. These fields are labeled **WLRL** in the register descriptions.

Implementations are permitted but not required to raise an illegal instruction exception if an instruction attempts to write a non-supported value to a **WLRL** field. Implementations can return arbitrary bit patterns on the read of a **WLRL** field when the last write was of an illegal value, but the value returned should deterministically depend on the illegal written value and the value of the field prior to the write.

Write Any Values, Reads Legal Values (WARL)

Some read/write CSR fields are only defined for a subset of bit encodings, but allow any value to be written while guaranteeing to return a legal value whenever read. Assuming that writing the CSR has no other side effects, the range of supported values can be determined by attempting to write a desired setting then reading to see if the value was retained. These fields are labeled **WARL** in the register descriptions.

Implementations will not raise an exception on writes of unsupported values to a **WARL** field. Implementations can return any legal value on the read of a **WARL** field when the last write was of an illegal value, but the legal value returned should deterministically depend on the illegal written value and the value of the field prior to the write.

3.4.3. Machine Vendor ID Register `mvendorid`

The `mvendorid` CSR is a 32-bit read-only register providing the JEDEC manufacturer ID of the provider of the core. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented or that this is a non-commercial implementation.

ParaNut returns a fixed value of 0 indicating a non-commercial implementation.

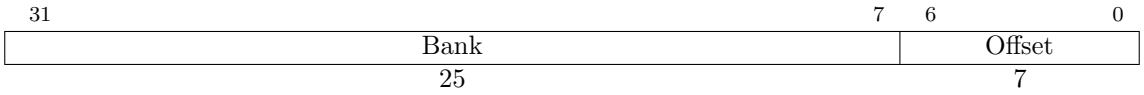


Figure 3.4.: Vendor ID register (`mvendorid`).

3.4.4. Machine Architecture ID Register `marchid`

The `marchid` CSR is an MXLEN-bit read-only register encoding the base microarchitecture of the hart. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented. The combination of `mvendorid` and `marchid` should uniquely identify the type of hart microarchitecture that is implemented.

ParaNut returns a fixed value of 0. Architecture ID is not yet requested from the RISC-V Foundation.

Figure 3.5.: Machine Architecture ID register (`marchid`).

3.4.5. Machine Implementation ID Register `mimpid`

The `mimpid` CSR provides a unique encoding of the version of the processor implementation. This register must be readable in any implementation, but a value of 0 can be returned to indicate that the field is not implemented. The Implementation value should reflect the design of the RISC-V processor itself and not any surrounding system.

ParaNut returns a fixed value based on the implementation version starting with a 0x2 as the most significant nibble. [[TBD: define more specific version (major.minor.revision)]]

Figure 3.6.: Machine Implementation ID register (`mimpid`).

3.4.6. 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. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but at least one hart must have a hart ID of zero. Hart IDs must be unique.

ParaNut harts are numbered contiguously and CePU is guaranteed to have an ID of zero. The maximum hart ID can be determined using the non-standard `pncpus` *CSR. (*`pncpus` *- 1 = highest hart ID)*

Figure 3.7.: Hart ID register (`mhartid`).

3.4.7. Machine Status Register `mstatus`

The `mstatus` register is an MXLEN-bit read/write register formatted as shown in Figure 3.8 for RV32. The `mstatus` register keeps track of and controls the hart's current operating state.

ParaNut currently only implements the MIE (Machine Interrupt Enable) and MPIE (Machine Privious Interrupt Enable) bits as writable registers. The other bits are all fixed to 0.
For a more accurate description of all the bits in this register please refer to the full RISC-V privileged specification.

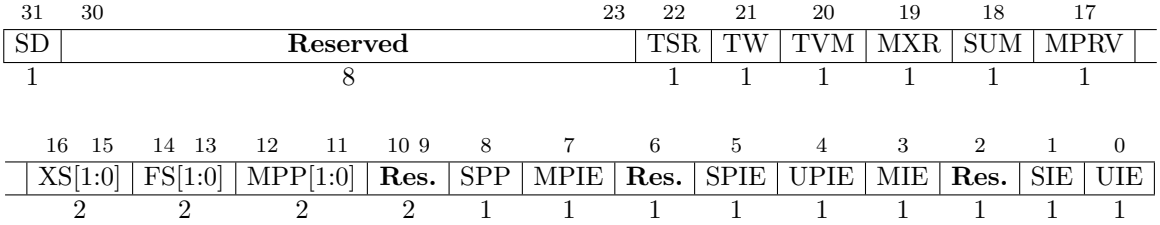


Figure 3.8.: Machine-mode status register (`mstatus`) for RV32.

3.4.8. Machine ISA Register `misa`

The `misa` CSR is a **WARL** read-write register reporting the ISA supported by the hart. This register must be readable in any implementation, but a value of zero can be returned to indicate the `misa` register has not been implemented, requiring that CPU capabilities be determined through a separate non-standard mechanism.

ParaNut returns a fixed value based on the activated RISC-V extensions at compile or synthesis time. Table 3.4 shows the encoding of the Extensions field. The MXL field is also fixed at 1 indicating the 32 Bit support.

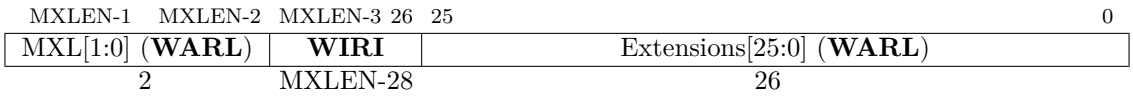


Figure 3.9.: Machine ISA register (`misa`).

Bit	Character	Description
0	A	Atomic extension
1	B	*Tentatively reserved for Bit-Manipulation extension*
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E base ISA
5	F	Single-precision floating-point extension
6	G	Additional standard extensions present
7	H	Hypervisor extension
8	I	RV32I/64I/128I base ISA
9	J	*Tentatively reserved for Dynamically Translated Languages extension*
10	K	*Reserved*
11	L	*Tentatively reserved for Decimal Floating-Point extension*
12	M	Integer Multiply/Divide extension
13	N	User-level interrupts supported
14	O	*Reserved*
15	P	*Tentatively reserved for Packed-SIMD extension*
16	Q	Quad-precision floating-point extension
17	R	*Reserved*
18	S	Supervisor mode implemented
19	T	*Tentatively reserved for Transactional Memory extension*
20	U	User mode implemented
21	V	*Tentatively reserved for Vector extension*
22	W	*Reserved*
23	X	Non-standard extensions present
24	Y	*Reserved*
25	Z	*Reserved*

Table 3.4.: Encoding of Extensions field in `misa`.

3.4.9. Machine Interrupt Registers `mip` and `mie`

The `mip` register is an MXLEN-bit read/write register containing information on pending interrupts, while `mie` is the corresponding MXLEN-bit read/write register containing interrupt enable bits. Only the bits corresponding to lower-privilege software interrupts (USIP, SSIP), timer interrupts (UTIP, STIP), and external interrupts (UEIP, SEIP) in `mip` are writable through this CSR address; the remaining bits are read-only.

The ParaNut *architecture defines interrupt support only for the CePU.*
In the current implementation external interrupts are still under construction, hence `mip` *and* `mie` *read fixed values of 0 and are implemented as* `WARL` *on writes. This is prone to change in future Versions.*
For a more accurate description of all the bits in this register please refer to the full RISC-V privileged specification.

The MEIP field in `mip` is a read-only bit that indicates a machine-mode external interrupt is pending. MEIP is set and cleared by a platform-specific interrupt controller.

MXLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI	MEIP	**WIRI**	SEIP	UEIP	MTIP	**WIRI**	STIP	UTIP	MSIP	**WIRI**	SSIP	USIP
MXLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.10.: Machine interrupt-pending register (`mip`).

MXLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI	MEIE	**WIRI**	SEIE	UEIE	MTIE	**WIRI**	STIE	UTIE	MSIE	**WIRI**	SSIE	USIE
MXLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.11.: Machine interrupt-enable register (`mie`).

The MEIE field in `mie` enables machine external interrupts when set.

The MEIE field in the `mie` CSR enables M-mode external interrupts.

3.4.10. Machine Trap-Vector Base-Address Register `mtvec`

The `mtvec` register is an MXLEN-bit read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

MXLEN-1	2 1	0
BASE[MXLEN-1:2] (**WARL**)	MODE (**WARL**)	
MXLEN-2	2	

Figure 3.12.: Machine trap-vector base-address register (`mtvec`).

The `mtvec` register must always be implemented, but can contain a hardwired read-only value. If `mtvec` is writable, the set of values the register may hold can vary by implementation. The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field.

Value	Name	Description
0	Direct	All exceptions set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	—	*Reserved*

Table 3.5.: Encoding of `mtvec` MODE field.

The encoding of the MODE field is shown in Table 3.5. When MODE=Direct, all traps into machine mode cause the `pc` to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into machine mode cause the `pc` to be set to the address in the BASE field, whereas interrupts cause the `pc` to be set to the address in the BASE field plus four times the interrupt cause number. For example, a machine-mode timer interrupt (see Table 3.6 on page 25) causes the `pc` to be set to BASE+`0x1c`.

ParaNut *only supports the direct mode and implements the* `mtvec` *register to enable a wide variety of software.*

3.4.11. Machine Scratch Register `mscratch`

The `mscratch` register is an MXLEN-bit read/write register dedicated for use by machine mode. Typically, it is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler.

Figure 3.13.: Machine-mode scratch register.

3.4.12. Machine Exception Program Counter `mepc`

`mepc` is an MXLEN-bit read/write register formatted as shown in Figure 3.14. The low bit of `mepc` (`mepc[0]`) is always zero. On implementations that support only IALIGN=32, the two low bits (`mepc[1:0]`) are always zero.

If an implementation allows IALIGN to be either 16 or 32 (by changing CSR `misa`, for example), then, whenever IALIGN=32, bit `mepc[1]` is masked on reads so that it appears to be 0. This masking occurs also for the implicit read by the MRET instruction. Though masked, `mepc[1]` remains writable when IALIGN=32.

`mepc` is a **WARL** register that must be able to hold all valid physical and virtual addresses. It need not be capable of holding all possible invalid addresses. Implementations may convert some invalid address patterns into other invalid addresses prior to writing them to `mepc`.

When a trap is taken into M-mode, `mepc` is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, `mepc` is never written by the implementation, though it may be explicitly written by software.

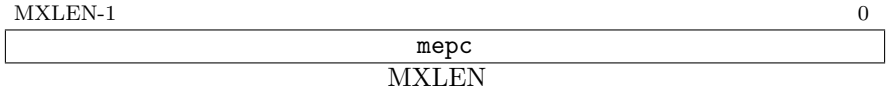


Figure 3.14.: Machine exception program counter register.

3.4.13. Machine Cause Register `mcause`

The `mcause` register is an MXLEN-bit read-write register formatted as shown in Figure 3.15. When a trap is taken into M-mode, `mcause` is written with a code indicating the event that caused the trap. Otherwise, `mcause` is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the `mcause` register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception. Table 3.6 lists the possible machine-level exception codes. The Exception Code is a **WLRL** field, so is only guaranteed to hold supported exception codes.

Note that ParaNut currently does not check the legality of a value on write through

software. Nonetheless, on exception only valid values will be written by hardware. Table 3.6 is modified to show the possible exceptions a ParaNut CPU currently can encounter in bold. Also the non-standard exception with code 16 for CoPU exceptions was added even though this number is reserved. More on this topic can be found in Section 3.6.

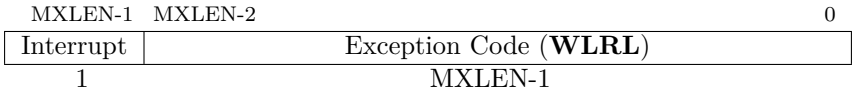


Figure 3.15.: Machine Cause register `mcause`.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	≥12	*Reserved*
0	0	**Instruction address misaligned**
0	1	Instruction access fault
0	2	**Illegal instruction**
0	3	**Breakpoint**
0	4	**Load address misaligned**
0	5	Load access fault
0	6	**Store/AMO address misaligned**
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	**Environment call from M-mode**
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16	**ParaNut CoPU exception** (*Reserved*)
0	≥17	*Reserved*

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

3.4.14. Machine Trap Value Register `mtval`

The `mtval` register is an MXLEN-bit read-write register formatted as shown in Figure 3.16. When a trap is taken into M-mode, `mtval` is either set to zero or written with exception-specific information to assist software in handling the trap. Otherwise, `mtval` is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set `mtval` informatively and which may unconditionally set it to zero.

When a hardware breakpoint is triggered, or an instruction-fetch, load, or store address-misaligned, access, or page-fault exception occurs, `mtval` is written with the faulting virtual address. On an illegal instruction trap, `mtval` may be written with the first XLEN or ILEN bits of the faulting instruction as described below. For other traps, `mtval` is set to zero, but a future standard may redefine `mtval`'s setting for other traps.

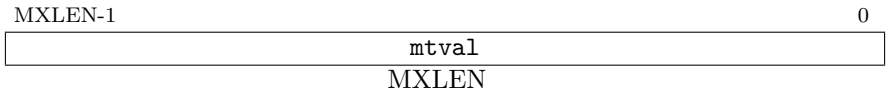


Figure 3.16.: Machine Trap Value register.

3.4.15. Hardware Performance Monitor

M-mode includes a basic hardware performance-monitoring facility. The `mcycle` CSR counts the number of clock cycles executed by the processor core on which the hart is running. The `minstret` CSR counts the number of instructions the hart has retired. The `mcycle` and `minstret` registers have 64-bit precision on all RV32 and RV64 systems.

The counter registers have an arbitrary value after system reset, and can be written with a given value. Any CSR write takes effect after the writing instruction has otherwise completed.

The hardware performance monitor includes 29 additional 64-bit event counters, `mhpmcounter3`–`mhpmcounter31`. The event selector CSRs, `mhpmevent3`–`mhpmevent31`, are MXLEN-bit **WARL** registers that control which event causes the corresponding counter to increment. The meaning of these events is defined by the platform, but event 0 is defined to mean "no event." All counters should be implemented, but a legal implementation is to hard-wire both the counter and its corresponding event selector to 0.

The hardware performance monitor counters can be configured in 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.*

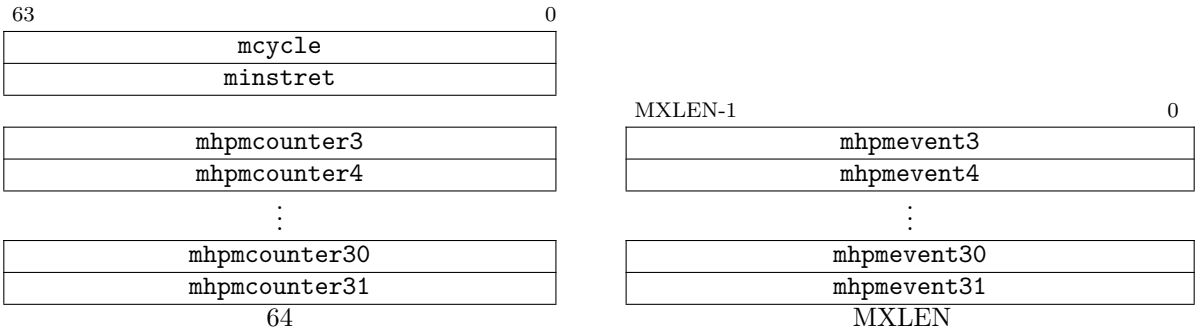


Figure 3.17.: Hardware performance monitor counters.

All of these counters have 64-bit precision on RV32 and RV64.

On RV32 only, reads of the `mcycle`, `minstret`, and `mhpmcounter`*n* CSRs return the low 32 bits, while reads of the `mcycleh`, `minstreth`, and `mhpmcounter`*n*`h` CSRs return bits 63–32 of the corresponding counter.

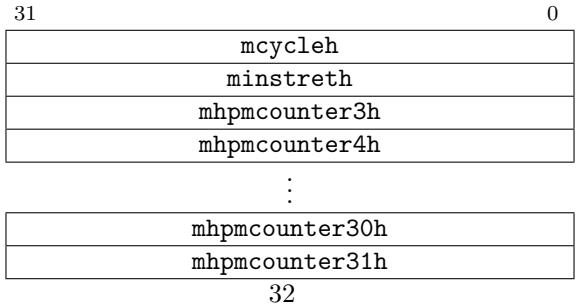


Figure 3.18.: Upper 32 bits of hardware performance monitor counters, RV32 only.

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.

3.4.16. Machine Timer Registers `mtime` and `mtimecmp`

Platforms provide a real-time counter, exposed as a memory-mapped machine-mode register, `mtime`. `mtime` must run at constant frequency, and the platform must provide a mechanism for determining the timebase of `mtime`.

The `mtime` register has a 64-bit precision on all RV32 and RV64 systems. Platforms provide a 64-bit memory-mapped machine-mode timer compare register (`mtimecmp`), which causes a timer interrupt to be posted when the `mtime` register contains a value greater than or equal to the value in the `mtimecmp` register. The interrupt remains posted until it is cleared by writing the `mtimecmp` register. The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the `mie` register.

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.*
Timers will be implemented in future versions according to this specification.

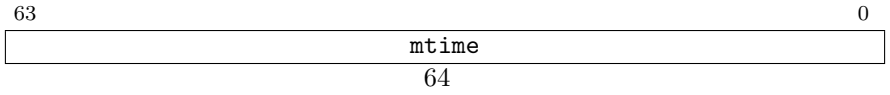


Figure 3.19.: Machine time register (memory-mapped control register).

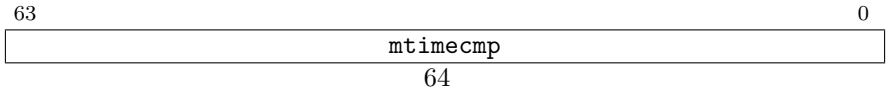


Figure 3.20.: Machine time compare register (memory-mapped control register).

3.5. *ParaNut* Control and Status Registers

Table 3.8 shows the *ParaNut* registers, which are used to query the hardware configuration and to set and query the status of the CPU array. These registers are only available on a CePU.

Number	Privilege	Name	Description
ParaNut R/W (Non-Standard R/W)			
0x7C0	MRW	pngrpsel	ParaNut CPU group select.
0x7C1	MRW	pnce	ParaNut CPU enable register.
0x7C2	MRW	pnlm	ParaNut CPU linked mode register.
0x7C3	MRW	pnxsel	ParaNut CoPU exception select register.
0x7C4	MRW	pncache	ParaNut Cache Control register.
ParaNut R (Non-Standard R)			
0xFC0	MRO	pncpus	ParaNut number of CPUs.
0xFC1	MRO	pnm2cp	ParaNut CPU capabilities register
0xFC2	MRO	pnx	ParaNut CoPU exception pending.
0xFC3	MRO	pncause	ParaNut CoPU trap cause ID.
0xFC4	MRO	pnepc	ParaNut CoPU exception program counter.
0xFC5	MRO	pncacheinfo	ParaNut cache information.
0xFC6	MRO	pncachesets	ParaNut number of cache sets.
0xFC7	MRO	pnclockinfo	ParaNut clock speed information.
0xFC8	MRO	pnmemsize	ParaNut memory size.

Table 3.8.: Currently allocated *ParaNut* CSRs

3.5.1. ParaNut CPU group select `pngrpsel`

The `pngrpsel` register is an MXLEN-bit read-write register formatted as shown in Figure 3.21. It only takes legal values (**WARL**) 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.

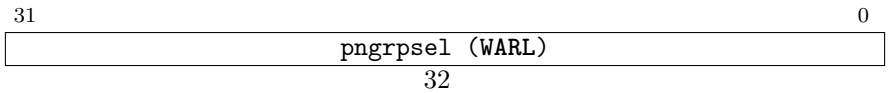


Figure 3.21.: ParaNut CPU group select.

3.5.2. ParaNut CPU enable register `pnce`

The `pnce` register is an MXLEN-bit read-write register formatted as shown in Figure 3.22. 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 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 moves into 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 hartID > 31.

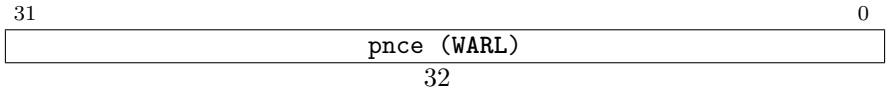


Figure 3.22.: ParaNut CPU enable register.

3.5.3. ParaNut CPU linked mode register `pnlm`

The `pnlm` register is an MXLEN-bit read-write register formatted as shown in Figure 3.23. 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 activate or deactivate CoPUs. 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 hartID > 32.

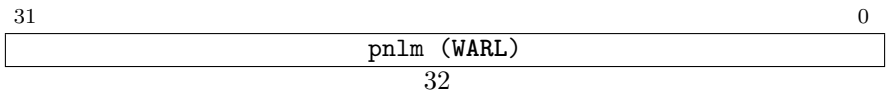


Figure 3.23.: ParaNut CPU linked mode register.

3.5.4. ParaNut CoPU exception select register `pnxsel`

The `pnxsel` register is an MXLEN-bit read-write register formatted as shown in Figure 3.23. 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 `mpncause` 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 hartID > 31.

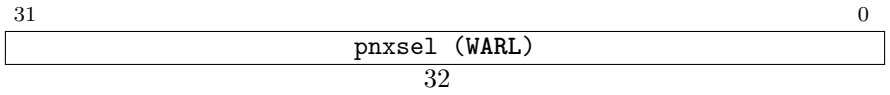


Figure 3.24.: ParaNut CoPU exception select register.

3.5.5. ParaNut Cache control register `pncache`

The `pncache` register is an MXLEN-bit read-write register formatted as shown in Figure 3.25. 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.

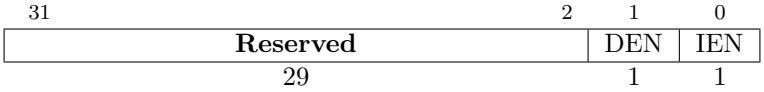


Figure 3.25.: ParaNut Cache control register.

When disabling the caches during normal program execution we advise to flush or writeback the cache beforehand.

3.5.6. ParaNut number of CPUs `pncpus`

The `pncpus` register is an MXLEN-bit read-only register formatted as shown in Figure 3.26. It holds the number of CPUs (including the CePU).

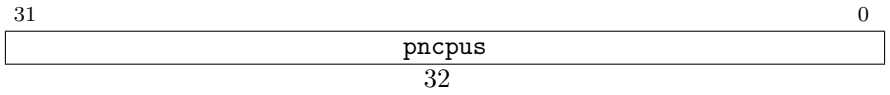


Figure 3.26.: ParaNut number of CPUs.

3.5.7. ParaNut CPU capabilities register `pnm2cp`

The `pnm2cp` register is an MXLEN-bit read-only register formatted as shown in Figure 3.27. 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 hartID > 31.

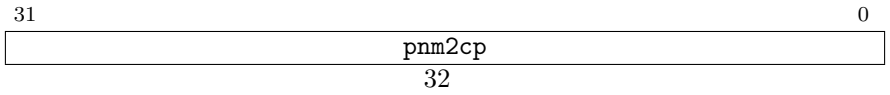


Figure 3.27.: ParaNut CPU capabilities register.

3.5.8. ParaNut CoPU exception pending `pnx`

The `pnx` register is an MXLEN-bit read-only register formatted as shown in Figure 3.28. 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 hartID > 31.

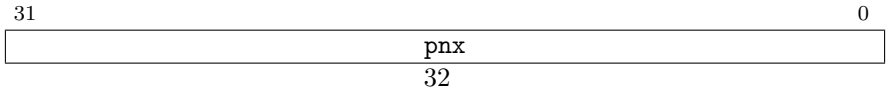


Figure 3.28.: ParaNut CoPU exception pending.

3.5.9. ParaNut CoPU trap cause ID `pncause`

The `pncause` register is an MXLEN-bit read-only register formatted as shown in Figure 3.29. It holds the cause of exception of the CoPU selected by `pnxsel` and `pngrpsel`. The CSR only holds legal values as defined in `mcause`.

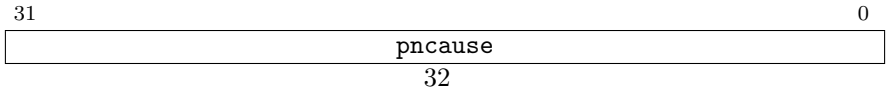


Figure 3.29.: ParaNut CoPU trap cause ID.

3.5.10. ParaNut CoPU exception program counter `pnepc`

The `pnepc` register is an MXLEN-bit read-only register formatted as shown in Figure 3.30. It holds the exception program counter of the CoPU selected by `pnxsel` and `pngrpsel`. The CSR only holds legal values as defined in `mepc`.

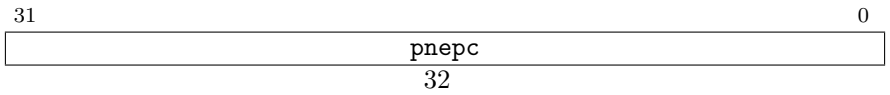


Figure 3.30.: ParaNut CoPU exception program counter.

3.5.11. ParaNut cache information register `pncacheinfo`

The `pncacheinfo` register is an MXLEN-bit read-only register formatted as shown in Figure 3.31. It holds information about the cache properties.

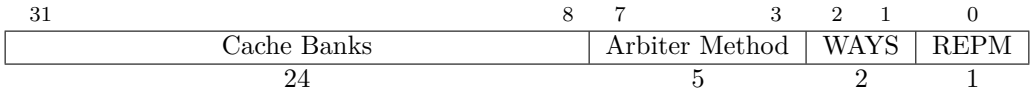


Figure 3.31.: ParaNut cache information register.

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.

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

3.5.12. ParaNut number of cache sets register `pncachesets`

The `pncachesets` register is an MXLEN-bit read-only register formatted as shown in Figure 3.32. It holds the number of cache sets.

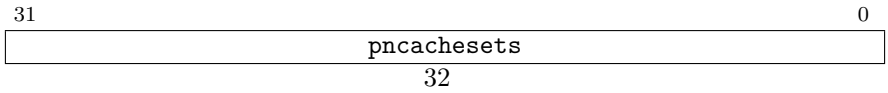


Figure 3.32.: ParaNut number of cache sets register.

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

3.5.13. ParaNut clock speed information register `pnclockinfo`

The `pnclockinfo` register is an MXLEN-bit read-only register formatted as shown in Figure 3.33. It holds the clock speed in Hz set at compile or synthesis time.

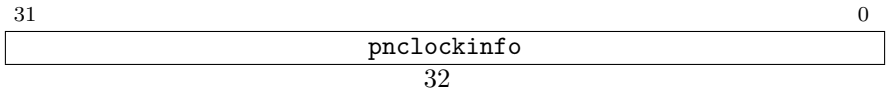


Figure 3.33.: ParaNut clock speed information register.

3.5.14. ParaNut memory size register `pnmemsize`

The `pnmemsize` register is an MXLEN-bit read-only register formatted as shown in Figure 3.34. It holds the memory size set at compile or synthesis time.

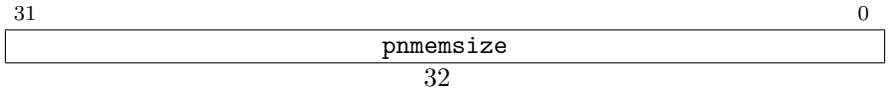


Figure 3.34.: ParaNut memory size register.

3.6. Exceptions

Table 3.6 lists the exceptions supported by the *ParaNut* architecture. All of the marked exceptions can occur in a mode 2 capable CoPU.

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. *... Exception handler executing ...*
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. *... Exception handler executing ...*
   - 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. [[TBD: Instead of "roll back" we may also specify: are restartable. This is easier to implement, e. g. loads which may for some CoPUs cause a page fault and for the other would then be executed twice without harm.]]

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. *... Exception handler executing ...*

   - 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

```
# Check for PARANUT_HOME (<ParaNut-Dir>/settings.sh)
ifndef PARANUT_HOME
$(error PARANUT_HOME is not set. Source <ParaNut-Dir>/settings.sh first)
endif

# 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)
```

```
19  OBJECTS = $(patsubst %.c,%.o,$(SOURCES))
20  HEADERS = $(wildcard *.h)
21
22  SYSTEMS_DIR = $(PARANUT_HOME)/systems
23  RISCV_COMMON_DIR = $(PARANUT_HOME)/sw/riscv_common
24
25  CFG_MARCH ?= rv32i
26
27  CFLAGS = -O2 -march=$(CFG_MARCH) -mabi=ilp32 -I$(RISCV_COMMON_DIR)
28  LDFLAGS = $(CFLAGS) -static -nostartfiles -lc $(RISCV_COMMON_DIR)/startup.S $
        (RISCV_COMMON_DIR)/syscalls.c -T $(RISCV_COMMON_DIR)/paranut.ld
29
30  # Software Targets
31  all: $(ELF) dump
32
33  $(ELF): $(OBJECTS)
34    $(CC) -o $@ $^ $(LDFLAGS)
35
36  %.o: %.c $(HEADERS)
37    $(CC) -c $(CFLAGS) $<
38
39
40  # ParaNut Targets
41  .PHONY: sim
42  sim: $(ELF)
43    +$(MAKE) -C $(PARANUT_HOME)/sysc paranut_tb
44    $(PARANUT_HOME)/sysc/paranut_tb -t0 $<
45
46  .PHONY: flash-%
47  flash-%: bin
48    paranut_flash -c -p $(ELF).bin $(SYSTEMS_DIR)/$*/hardware/build/system.hdf
          $(SYSTEMS_DIR)/$*/hardware/firmware/firmware.elf
49
50  flash-%-bit: bin
51    paranut_flash -c -b $(SYSTEMS_DIR)/$*/hardware/build/system.bit -p $(ELF).
          bin \
52    $(SYSTEMS_DIR)/$*/hardware/build/system.hdf $(SYSTEMS_DIR)/$*/hardware/
          firmware/firmware.elf
53
54  # Misc Targets
55  .PHONY: dump
56  dump: $(ELF).dump
57  $(ELF).dump: $(ELF)
58    $(OBJDUMP) -S -D $< > $@
59
60  .PHONY: bin
61  bin: $(ELF).bin
62  $(ELF).bin: $(ELF)
63    $(OBJCOPY) -S -O binary $< $@
64
```

```
65  .PHONY: clean
66  clean:
67    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 `PARANUT_HOME` (see settings.sh) 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:

```
$ ../../sysc/paranut_tb 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:

```
$ ../../sysc/paranut_tb -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:

```
$ ../../sysc/paranut_tb -d hello_newlib
```

In a new shell start OpenOCD and use the `tools/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_HOME/tools/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.