Hochschule
Augsburg University of
Applied Sciences

The *ParaNut* Processor

Architecture Description and Reference Manual

Gundolf Kiefer

Hochschule Augsburg – University of Applied Sciences
gundolf.kiefer@hs-augsburg.de

Version 0.1.0

September 24, 2012

For internal use only!

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 level and data level based on multiple processing cores while keeping the design of the individual core itself as simple as possible.

The *ParaNut* introduces a unique concept for SIMD (single instruction, multiple data) vectorization. Whereas SIMD extentsions for workstation processors or embbeded 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 similar to thread programming.

The instruction set is kept compatible to the OpenRISC 1000 specification. Hence, the OpenRISC GCC tool chain and libraries/operation systems (newlib, Linux with some necessary extensions) can be used with the *ParaNut* .

Countless techniques for accelerating CPUs by means parallelism can be found in contemporary textbooks on computer architecture. In essence, the techniques can be grouped into three categories: *data-level parallelism (DLP)*, *instruction-level parallelism (ILP)*, and *thread-level parallelism (TLP)* [1]. SIMD (single instruction, multiple data) extensions, such as Intel's SSE or the ARM SIMD media extensions, fall into the DLP category. Such extensions often contain quite specialized instructions leading to an inherently bad compiler support. Hence, in practice, such extensions can only be fully leveraged by time-consuming assembly programming or by using special libraries. The ILP category includes all the techniques that improve the throughput of single general-purpose instruction stream, such as (deep) pipelining, out-of-order execution, scoreboards, or VLIW (very long instruction word) architectures. The acceleration comes at certain cost in terms of chip area, since special hardware structures are needed to avoid pipeline conflicts and to perform the instruction scheduling. These hardware structures may require an enormous amount chip area and cause power consumption, but they do not perform the operations originally defined by the program. Finally, the TLP category comprises multi-core architectures as well as all techniques for simultaneous multi-threading.

From the hardware perspective, the mentioned DLP and TLP techniques are preferable, since the ILP techniques requires extra hardware which does not perform real workload. However, from the software developers perspective, ILP is preferred. DLP techniques require time-consuming assembly programming, leading to platform-dependent code. TLP requires to write parallel code and to deal with potential race conditions. However, in the present era of multi-core processors even in Embedded Systems, parallel programming becomes more and more common.

2. The *ParaNut* Architecture

2.1. Overview

The general structure of *ParaNut* is depicted in 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 also 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 adress 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.

[[TBD: Overview OpenRISC architecture]]

2.2. Execution Modes

A CPU (both CePU and CoPU) 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 are handled by this CPU.

Typically, the CePU always runs in Mode 3 (or Mode 0, if the whole core is in a sleep state). The mode of the CoPUs is controlled by the CePU. The following sections describe typical configurations for the complete *ParaNut* to illustrate how SIMD vectorization or

[[TBD]]

Figure 2.1.: The *ParaNut* architecture

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

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

void add_arrays_parallel () {
  int n, cpu_no;

  pn_begin_linked (4);          // activate 3 (=4-1) CoPUs in the "Linked" s

  cpu_no = pn_get_cpu_no();     // get the number of this CPU
  for (n = 0; n < 100; n += 4)
    s[n + cpu_no] = a[n + cpu_no] + b[n + cpu_no];
// performs 4 additions in parallel

  pn_end_linked ();             // now, only the CePU is active again.
}
```

Figure 2.2.: Example of a vectorized loop

multi-threading can be performed. It should be noted, however, that the *ParaNut* is not restricted to these configurations, but that e. g. combinations of multi-threading and vectorization are also possible by putting the CoPUs into the respective states.

2.2.1. 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, with only a single, clearly-defined exception: Conditional branches are executed based on the outcome of the condition on the CoPU. Hence, SIMD code can be easily written using a standard compiler. 2.2 shows an example of a vectorized loop.

An often requested feature for SIMD extensions are scatter-gather operations [[...]]

2.2.2. Multi-Threading

To perform classical simultanous 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.

[[TBD: reference to pthreads, OpenMP, ...]]

2.2.3. Hardware Customization

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 3 and 4 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, if a CoPU is customized for mode m, all modes $\leq m$ are supported.

- A Mode-1-CoPU only contains very little logic besides the ALU and the register file. Hence, a *ParaNut* with only Mode-1-CoPUs does not require much more area than a normal SIMD processor.
- A Mode-2-CoPU additionally contains an instruction fetch unit and eventually one more read port at the *Memory Unit (MemU)* for it.
- A Mode-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.

2.3. Software Support

[[TBD: OpenRISC tool chain, newlib, Linux]]

2.4. Operating System Support

[[TBD]]

[[

The question may arise, what the differences are compared to CoPUs which only support Mode 2, and whether the advantages of a Mode-3-CoPU are relevant for an embedded system.

- In a Mode-2-CoPU, a lot of logic for exception and interrupt handling can be saved [[what else? SRs?]].
- A multi-processor operating system needs full-featured processors. With Mode-2-CoPUs, the operating system only "sees" the CePU. [[...]]

]]

3. Hardware Implementation

[[TBD]]

3.1. Overview

[[TBD]]

3.2. Instruction Fetch Unit (IFU)

[[TBD]]

3.3. Execution Unit (EXU)

[[TBD]]

3.4. Load-Store-Unit (LSU)

[[TBD]]

3.5. Memory Unit (MemU)

[[TBD]]

3.6. SystemC Model & Testbench

[[TBD]]

3.7. VHDL Model & Testbench

[[TBD]]

[[TBD]]

Figure 3.1.: MemU block diagram

4. Software Development and Support Library

[[TBD]]

4.1. Basic Support Library: *libparanut*

4.1.1. Overview

4.1.2. Mode Setting

[[TBD: set modes, begin/end vectorization, begin/end threading]]

4.1.3. Helpers

[[TBD: helpers, e.g. conditional moves to replace if-constructs in linked mode]]

4.1.4. Other Functions

[[TBD / remove]]

4.2. POSIX Threads (*pthreads*)

[[TBD]]

4.3. Linux Support

[[TBD]]

5. Conclusion

[[TBD]]

A. Instruction Set Reference

A.1. Instructions

A.1.1. ALU Instructions

l.add – Add

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	----	0000

Format: `l.add rD, rA, rB`

Description: The contents of the general-purpose registers rA and rB are added. The result is placed into rD.

Operation:
```
rD <- rA + rB
SR[CY] <- Carry
SR[OV] <- Overflow
```

Exceptions: Range Exception

l.addc – Add with Carry

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	----	0001

Format: `l.addc rD, rA, rB`

Description: The contents of the general-purpose registers rA, rB, and the carry flag are added. The result is placed into rD.

Operation:
```
rD <- rA + rB + SR[CY]
SR[CY] <- Carry
SR[OV] <- Overflow
```

Exceptions: Range Exception

l.sub – Subtract

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	----	0010

Format: `l.sub rD, rA, rB`

Description: The contents of the general-purpose register rB is subtracted from rA. The result is placed into rD.

Note: The OR1k specification does not clearly specify whether the carry flag is affected or not.

Operation:
```
rD <- rA - rB
SR[CY] <- Carry
SR[OV] <- Overflow
```

Exceptions: Range Exception

l.and – Logical AND

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	----	0011

Format: `l.and rD, rA, rB`

Description: A bit-wise logical AND operation is performed on the contents of the general-purpose registers rA and rB. The result is placed into rD.

Operation: `rD <- rA and rB`

Exceptions: None

l.or – Logical OR

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	----	0100

Format: `l.or rD, rA, rB`

Description: A bit-wise logical OR operation is performed on the contents of the general-purpose registers rA and rB. The result is placed into rD.

Operation: `rD <- rA or rB`

Exceptions: None

l.xor – Logical XOR

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	----	0101

Format: `l.xor rD, rA, rB`

Description: A bit-wise logical XOR operation is performed on the contents of the general-purpose registers rA and rB. The result is placed into rD.

Operation:
```
rD <- rA xor rB
```

Exceptions: None

l.sll – Shift Left Logical

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	00--	1000

Format: `l.sll rD, rA, rB`

Description: The contents of register rA are shifted left by the number of bit positions specified in register rB. Low-order bits are filled with 0. The result is placed into rD.

Operation:
```
rD[31:rB[4:0]] <- rA[31-rB[4:0]:0]
rD[rB[4:0]-1:0] <- 0
```

Exceptions: None

l.srl – Shift Right Logical

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	01--	1000

Format: `l.srl rD, rA, rB`

Description: The contents of register rA are shifted right by the number of bit positions specified in register rB. High-order bits are filled with 0. The result is placed into rD.

Operation:
```
rD[31-rB[4:0]:0] <- rA[31:rB[4:0]]
rD[31:32-rB[4:0]] <- 0
```

Exceptions: None

l.sra – Shift Right Arithmetic

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	10--	1000

Format: `l.sra rD, rA, rB`

Description: The contents of register rA are shifted right by the number of bit positions specified in register rB. High-order bits are filled with rA[31]. The result is placed into rD.

Operation:
```
rD[31-rB[4:0]:0] <- rA[31:rB[4:0]]
rD[31:32-rB[4:0]] <- rA[31]
```

Exceptions: None

l.cmov – Conditional Move

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	----	1110

Format: `l.cmov rD, rA, rB`

Description: If SR[F] is set, general-purpose register rA is placed into register rD. Otherwise, register rB is placed into rD.

Operation:
```
rD[31:0] < - SR[F] ? rA[31:0] :  rB[31:0]
```

Exceptions: None

l.mul – Multiply Signed

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	11	----	0110

Format: `l.mul rD, rA, rB`

Description: The contents of registers rA and rB are multiplied. The result is truncated to 32 bit and placed into register rD. Both operands are treated as *signed* integers.

None (*Note:* In contrast to the OR1k specification, the flags CY and OV are not affected, and no range exception can be generated.

Operation: `rD <- rA * rB`

Exceptions: None (OR1k: Range Exception)

l.mulu – Multiply Unsigned

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	11	----	1011

Format: `l.mulu rD, rA, rB`

Description: The contents of registers rA and rB are multiplied. The result is truncated to 32 bit and placed into register rD. Both operands are treated as *unsigned* integers.

None (*Note:* In contrast to the OR1k specification, the flags CY and OV are not affected.

Operation: `rD <- rA * rB`

Exceptions: None (OR1k: Range Exception)

l.sfeq – Set Flag if Equal

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111001	00000	aaaaa	bbbbb	-	--	----	----

Format: `l.sfeq rA, rB`

Description: The contents of registers rA and rB are compared. The flag SR[F] is set, if they are equal, and unset otherwise.

Operation: `SR[F] <- (rA == rB)`

Exceptions: None

l.sfne – Set Flag if Not Equal

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111001	00001	aaaaa	bbbbb	-	--	----	----

Format: `l.sfne rA, rB`

Description: The contents of registers rA and rB are compared. The flag SR[F] is set, if they are different, and unset otherwise.

Operation: `SR[F] <- (rA != rB)`

Exceptions: None

l.sfgtu – Set Flag if Greater Than Unsigned

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111001	00010	aaaaa	bbbbb	-	--	----	----

Format: `l.sfgtu rA, rB`

Description: The contents of registers rA and rB are interpreted as unsigned numbers and compared. The flag SR[F] is set, if rA $>$ rB, and unset otherwise.

Operation: `SR[F] <- (rA > rB)`

Exceptions: None

l.sfgeu – Set Flag if Greater or Equal Unsigned

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111001	00011	aaaaa	bbbbb	-	--	----	----

Format: `l.sfgeu rA, rB`

Description: The contents of registers rA and rB are interpreted as unsigned numbers and compared. The flag SR[F] is set, if rA >= rB, and unset otherwise.

Operation: `SR[F] <- (rA >= rB)`

Exceptions: None

l.sfltu – Set Flag Less Than Unsigned

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111001	00100	aaaaa	bbbbb	-	--	----	----

Format: `l.sfltu rA, rB`

Description: The contents of registers rA and rB are interpreted as unsigned numbers and compared. The flag SR[F] is set, if rA < rB, and unset otherwise.

Operation: `SR[F] <- (rA < rB)`

Exceptions: None

l.sfleu – Set Flag if Less or Equal Unsigned

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111001	00101	aaaaa	bbbbb	-	--	----	----

Format: `l.sfleu rA, rB`

Description: The contents of registers rA and rB are interpreted as unsigned numbers and compared. The flag SR[F] is set, if rA <= rB, and unset otherwise.

Operation: `SR[F] <- (rA <= rB)`

Exceptions: None

l.sfgts – Set Flag if Greater Than Signed

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111001	01010	aaaaa	bbbbb	-	--	----	----

Format: `l.sfgts rA, rB`

Description: The contents of registers rA and rB are interpreted as signed numbers and compared. The flag SR[F] is set, if rA > rB, and unset otherwise.

Operation: `SR[F] <- (rA > rB)`

Exceptions: None

l.sfges – Set Flag if Greater or Equal Signed

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111001	01011	aaaaa	bbbbb	-	--	----	----

Format: `l.sfges rA, rB`

Description: The contents of registers rA and rB are interpreted as signed numbers and compared. The flag SR[F] is set, if rA >= rB, and unset otherwise.

Operation: `SR[F] <- (rA >= rB)`

Exceptions: None

l.sflts – Set Flag Less Than Signed

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111001	01100	aaaaa	bbbbb	-	--	----	----

Format: `l.sflts rA, rB`

Description: The contents of registers rA and rB are interpreted as signed numbers and compared. The flag SR[F] is set, if rA < rB, and unset otherwise.

Operation: `SR[F] <- (rA < rB)`

Exceptions: None

l.sfles – Set Flag if Less or Equal Signed

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111001	01101	aaaaa	bbbbb	-	--	----	----

Format: `l.sfles rA, rB`

Description: The contents of registers rA and rB are interpreted as signed numbers and compared. The flag SR[F] is set, if rA <= rB, and unset otherwise.

Operation:
```
SR[F] <- (rA <= rB)
```

Exceptions: None

l.addi – Add Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
100111	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.addi rD, rA, I`

Description: The contents of the general-purpose registers rA and the sign-extended immediate value I are added. The result is placed into rD.

Operation:
```
rD <- rA + exts(I)
SR[CY] <- Carry
SR[OV] <- Overflow
```

Exceptions: None

l.addic – Add Immediate with Carry

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101000	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.addic rD, rA, I`

Description: The contents of the general-purpose registers rA, the sign-extended immediate value I, and the carry flag are added. The result is placed into rD.

Operation:
```
rD <- rA + exts(I) + SR[CY]
SR[CY] <- Carry
SR[OV] <- Overflow
```

Exceptions: None

l.andi – Logical AND with Immediate Half Word

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101001	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.andi rD, rA, I`

Description: A bit-wise logical AND operation is performed on the contents of the general-purpose registers rA and the zero-extended immediate value I. The result is placed into rD.

Operation: `rD <- rA and extz(I)`

Exceptions: None

l.ori – Logical OR with Immediate Half Word

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101010	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.ori rD, rA, I`

Description: A bit-wise logical OR operation is performed on the contents of the general-purpose registers rA and the zero-extended immediate value I. The result is placed into rD.

Operation: `rD <- rA or extz(I)`

Exceptions: None

l.xori – Logical XOR with Immediate Half Word

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101011	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.xori rD, rA, I`

Description: A bit-wise logical XOR operation is performed on the contents of the general-purpose registers rA and the sign-extended immediate value I. The result is placed into rD.

Note: In the OR1200 implementation, the immediate value is zero-extended, whereas *ParaNut* sticks to the original OR1k specification. This allows a 32-bit NOT operation to be implemented as `l.xori rA, rB, -1`.

Operation: `rD <- rA xor exts(I)`

Exceptions: None

l.muli – Multiply Immediate Signed

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101100	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.muli rD, rA, I`

Description: The contents of the register rA and the immediate value I are multiplied. The result is truncated to 32 bit and placed into register rD. Both operands are treated as signed integers.

None (*Note:* In contrast to the OR1k specification, the flags CY and OV are not affected, and no range exception can be generated.

Operation: `rD <- rA * exts(I)`

Exceptions: None (OR1k: Range Exception)

l.sfeqi – Set Flag if Equal Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101111	00000	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.sfeqi rA, I`

Description: The contents of the register rA and the immediate value I are compared. The flag SR[F] is set, if they are equal, and unset otherwise.

Operation: `SR[F] <- (rA == I)`

Exceptions: None

l.sfnei – Set Flag if Not Equal Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101111	00001	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.sfnei rA, I`

Description: The contents of the register rA and the immediate value I are compared. The flag SR[F] is set, if they are different, and unset otherwise.

Operation: `SR[F] <- (rA != I)`

Exceptions: None

l.sfgtui – Set Flag if Greater Than Unsigned Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101111	00010	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.sfgtui rA, I`

Description: The contents of the register rA and the immediate value I are interpreted as unsigned numbers and compared. The flag SR[F] is set, if rA $>$ I, and unset otherwise.

Operation: `SR[F] <- (rA > I)`

Exceptions: None

l.sfgeui – Set Flag if Greater or Equal Unsigned Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101111	00011	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.sfgeui rA, I`

Description: The contents of the register rA and the immediate value I are interpreted as unsigned numbers and compared. The flag SR[F] is set, if rA >= I, and unset otherwise.

Operation: `SR[F] <- (rA >= I)`

Exceptions: None

l.sfltui – Set Flag Less Than Unsigned Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101111	00100	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.sfltui rA, I`

Description: The contents of the register rA and the immediate value I are interpreted as unsigned numbers and compared. The flag SR[F] is set, if rA < I, and unset otherwise.

Operation: `SR[F] <- (rA < I)`

Exceptions: None

l.sfleui – Set Flag if Less or Equal Unsigned Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101111	00101	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.sfleui rA, I`

Description: The contents of the register rA and the immediate value I are interpreted as unsigned numbers and compared. The flag SR[F] is set, if rA <= I, and unset otherwise.

Operation: `SR[F] <- (rA <= I)`

Exceptions: None

l.sfgtsi – Set Flag if Greater Than Signed Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101111	01010	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.sfgtsi rA, I`

Description: The contents of the register rA and the immediate value I are interpreted as signed numbers and compared. The flag SR[F] is set, if rA > I, and unset otherwise.

Operation: `SR[F] <- (rA > I)`

Exceptions: None

l.sfgesi – Set Flag if Greater or Equal Signed Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101111	01011	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.sfgesi rA, I`

Description: The contents of the register rA and the immediate value I are interpreted as signed numbers and compared. The flag SR[F] is set, if rA >= I, and unset otherwise.

Operation: `SR[F] <- (rA >= I)`

Exceptions: None

l.sfltsi – Set Flag Less Than Signed Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101111	01100	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.sfltsi rA, I`

Description: The contents of the register rA and the immediate value I are interpreted as signed numbers and compared. The flag SR[F] is set, if rA < I, and unset otherwise.

Operation: `SR[F] <- (rA < I)`

Exceptions: None

l.sflesi – Set Flag if Less or Equal Signed Immediate

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101111	01101	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.sflesi rA, I`

Description: The contents of the register rA and the immediate value I are interpreted as signed numbers and compared. The flag SR[F] is set, if rA <= I, and unset otherwise.

Operation:
```
SR[F] <- (rA <= I)
```

Exceptions: None

l.movhi – Move Immediate High

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
000110	ddddd	----0	iiiii	i	ii	iiii	iiii

Format: `l.movhi rD, I`

Description: The immediate value I is placed into the high-order 16 bits of register rD. The low-order bits of rD are cleared.

Operation:
```
rD[31:16] <- I
rD[15:0] <- 0
```

Exceptions: None

A.1.2. Load & Store Instructions

l.lwz – Load Word and Extend with Zero

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
100001	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.lwz rD, I(rA)`

Description: A word is loaded from memory and placed into register rD. The effective address is determined by adding the contents of rA to the sign-extended immediate value I.

Note: For *ParaNut* , the instructions l.lwz and l.lws are equivalent.

Operation: `rD <- Mem (rA + exts(I)) [31:0]`

Exceptions: Alignment
TLB miss
Page fault
Bus error

l.lws – Load Word and Extend with Sign

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
100010	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.lws rD, I(rA)`

Description: A word is loaded from memory and placed into register rD. The effective address is determined by adding the contents of rA to the sign-extended immediate value I.

Note: For *ParaNut* , the instructions l.lwz and l.lws are equivalent.

Operation: `rD <- Mem (rA + exts(I)) [31:0]`

Exceptions: Alignment
TLB miss
Page fault
Bus error

l.lbz – Load Byte and Extend with Zero

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
100011	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.lbz rD, I(rA)`

Description: A single byte is loaded from memory, zero-extended, and then placed into register rD. The effective address is determined by adding the contents of rA to the sign-extended immediate value I.

Operation: `rD <- extz ( Mem (rA + exts(I)) [7:0] )`

Exceptions: TLB miss
Page fault
Bus error

Exceptions: None

l.lbs – Load Byte and Extend with Sign

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
100100	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.lbs rD, I(rA)`

Description: A single byte is loaded from memory, sign-extended, and then placed into register rD. The effective address is determined by adding the contents of rA to the sign-extended immediate value I.

Operation: `rD <- exts ( Mem (rA + exts(I)) [7:0] )`

Exceptions: TLB miss
Page fault
Bus error

l.lhz – Load Half Word and Extend with Zero

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
100101	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.lhz rD, I(rA)`

Description: A half word is loaded from memory, zero-extended, and then placed into register rD. The effective address is determined by adding the contents of rA to the sign-extended immediate value I.

Operation: `rD <- extz ( Mem (rA + exts(I)) [15:0] )`

Exceptions:
Alignment
TLB miss
Page fault
Bus error

l.lhs – Load Half Word and Extend with Sign

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
100110	ddddd	aaaaa	iiiii	i	ii	iiii	iiii

Format: `l.lhs rD, I(rA)`

Description: A half word is loaded from memory, sign-extended, and then placed into register rD. The effective address is determined by adding the contents of rA to the sign-extended immediate value I.

Operation: `rD <- exts ( Mem (rA + exts(I)) [15:0] )`

Exceptions:
Alignment
TLB miss
Page fault
Bus error

l.sw – Store Word

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
110101	iiiii	aaaaa	bbbbb	i	ii	iiii	iiii

Format: `l.sw I(rA), rB`

Description: The contents of register rB are stored as a word. The effective address is determined by adding the contents of rA to the sign-extended immediate value I.

Operation: `Mem (rA + exts(I)) <- rB`

Exceptions: Alignment
TLB miss
Page fault
Bus error

l.sb – Store Byte

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
110110	iiiii	aaaaa	bbbbb	i	ii	iiii	iiii

Format: `l.sb I(rA), rB`

Description: The low-order bits of register rB are stored as a byte. The effective address is determined by adding the contents of rA to the sign-extended immediate value I.

Operation: `Mem (rA + exts(I)) <- rB[7:0]`

Exceptions: TLB miss
Page fault
Bus error

l.sw – Store Half Word

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
110111	iiiii	aaaaa	bbbbb	i	ii	iiii	iiii

Format: `l.sh I(rA), rB`

Description: The low-order bits of register rB are stored as a half word. The effective address is determined by adding the contents of rA to the sign-extended immediate value I.

Operation: `Mem (rA + exts(I)) <- rB[15:0]`

Exceptions: Alignment
TLB miss
Page fault
Bus error

A.1.3. Control Flow Instructions

l.j – Jump

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
000000	nnnnn	nnnnn	nnnnn	n	nn	nnnn	nnnn

Format: `l.j N`

Description: The instruction jumps unconditionally with a delay of one instruction. The target adress is determined by adding an immediate constant offset to the current PC, which refers the address of the jump instruction. The immediate offset is determined by multiplying the sign-extended 26-bit immediate value I by 4.

Operation: `PC <- PC + 4 * exts(N)`

Exceptions: None

l.jal – Jump and Link

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
000001	nnnnn	nnnnn	nnnnn	n	nn	nnnn	nnnn

Format: `l.jal N`

Description: The instruction jumps unconditionally with a delay of one instruction, and the address of the instruction after the delay slot is placed into the link register. The target adress is determined by adding an immediate constant offset to the current PC, which refers the address of the jump instruction. The immediate offset is determined by multiplying the sign-extended 26-bit immediate value I by 4.

Operation:
```
LR <- PC + 8
R9 <- PC + 4 * exts(N)
```

Exceptions: None

l.bnf – Branch if No Flag

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
000011	nnnnn	nnnnn	nnnnn	n	nn	nnnn	nnnn

Format: `l.bnf N`

Description: If the flag SR[F] is not set, the instruction jumps with a delay of one instruction. The target adress is determined by adding an immediate constant offset to the current PC, which refers the address of the jump instruction. The immediate offset is determined by multiplying the sign-extended 26-bit immediate value I by 4.

Operation:
```
if (SR[F] == 0) PC <- PC + 4 * exts(N)
```

Exceptions: None

l.bnf – Branch if Flag

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
000100	nnnnn	nnnnn	nnnnn	n	nn	nnnn	nnnn

Format: `l.bf N`

Description: If the flag SR[F] is set, the instruction jumps with a delay of one instruction. The target adress is determined by adding an immediate constant offset to the current PC, which refers the address of the jump instruction. The immediate offset is determined by multiplying the sign-extended 26-bit immediate value I by 4.

Operation: `if (SR[F] == 1) PC <- PC + 4 * exts(N)`

Exceptions: None

l.nop – No Operation

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
000101	01---	-----	kkkkk	k	kk	kkkk	kkkk

Format: `l.nop K`

Description: In general, the instruction does nothing. However, the OR1K simulator, certain values for K may trigger special actions.

The instruction *l.nop 1* is handled as a HALT instruction. [[TBD: Define own HALT?]]

Note: Different from the OR1k specification, the execution time may also be zero.

Operation: `(None)`

Exceptions: None

l.jr – Jump Register

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
010001	-----	-----	bbbbb	-	--	----	----

Format: `l.jr rB`

Description: The instruction jumps unconditionally with a delay of one instruction. The contents of general-purpose register rB are used as the target adress.

Operation:
```
PC <- rB
```

Exceptions: None

l.jalr – Jump and Link Register

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
010010	-----	-----	bbbbb	-	--	----	----

Format: `l.jalr rB`

Description: The instruction jumps unconditionally with a delay of one instruction, and the address of the instruction after the delay slot is placed into the link register. The contents of general-purpose register rB are used as the target adress.

Operation:
```
R9 <- PC + 8
PC <- rB
```

Exceptions: None

l.sys – System Call

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
001000	00000	00000	kkkkk	k	kk	kkkk	kkkk

Format: `l.sys K`

Description: Execution of this instruction results in the system call exception. The system calls exception is a request to the operating system to provide operating system services. The immediate value can be used to specify which system service is requested, alternatively a GPR defined by the ABI can be used to specify system service.

Operation:
```
EPCR <- NPC
ESR <- SR
PC <- 0xc00
```

Exceptions: System call

l.rfe – Return from Exception

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
001001	-----	-----	-----	-	--	----	----

Format: `l.rfe`

Description: Execution of this instruction partially restores the state of the processor prior to the exception. This instruction does not have a delay slot.

Operation:
```
PC <- EPCR
SR <- ESR
```

Exceptions: None

A.1.4. Special Instructions

l.mfspr – Move from Special Purpose Register

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
101101	ddddd	aaaaa	kkkkk	k	kk	kkkk	kkkk

Format: `l.mfspr rD, rA, K`

Description: The contents of the special register, defined by contents of register rA logically ORed with the immediate value, are moved into register rD.

Operation: `rD <- SR(rA or K)`

Exceptions: None

l.mfspr – Move from Special Purpose Register

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
110000	kkkkk	aaaaa	bbbbb	k	kk	kkkk	kkkk

Format: `l.mtspr rA, rB, K`

Description: The contents of the general-purpose register rB are moved into the special register defined by contents of register rA logically ORed with the immediate value.

Operation: `SR(rA or K) <- rD`

Exceptions: None

A.1.5. *ParaNut* Extensions

A.2. Special-Purpose Registers

This section describes all supported registers inside the *ParaNut* core. Shifting the group number *GRP* 11 bits left and adding the register number *REG* computes the address of each special-purpose register. All registers are 32 bits wide from software perspective. *CePU* and *CoPU* specify the valid access types for each register in a *CePU* and a *CoPU* (modes 1 and 2), respectively. R stands for read access and W stands for write access. CoPUs supporting mode 3 implement the same registers as CePUs.

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

The *ParaNut* does not implement the cache writeback/invalidate/flush registers. Instead, to allow a smaller hardware implementation, the new instructions *p.cwriteback*,

p.cinvalidate, and *p.cflush* were introduced for the same functionality in the group of load/store operations (see Section A.1.5).

Group 24 contains the *ParaNut* registers, which are used to query the hardware configuration and to set and query the status of the CPU array.

PNCPUS Number of CPUs (including the CePU).

PNM2CAP Each bit corresponds to one CPU. If the bit is set, the respective CPU supports *Mode 2* (unlinked mode) or higher. If unset, the respective CPU supports only *Mode 0* (halt) and *Mode 1* (linked).

PNCE Each bit corresponds to one CPU. Bit 0 represents the CePU and cannot be set to 0. By writing into this register, the CePU can activate or deactive CoPUs. By reading the register, the CePU can determine whether the CoPU is actually active. Deactivation may take some time until the CoPU moves into a stable state.

PNLM Each bit corresponds to one CPU. 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). By writing into this register, the CePU can switch the mode of the CoPUs. Mode switching is allowed only if the CoPU is inactive. If a bit is changed in the PNLM register and the respective PNCE bit is 1, undefined behavior will result!

PNX Each bit corresponds to one CPU. If set, an exception condition has occured. The bits are reset automatical when the register is read.

PNXIDn The exception ID of CPU #n. ([[PNXID0 may be undefined]])

A.2.1. Supervision Register (SR)

The fields of the Supervision Register (SR) are listed in Table A.2.

A.2.2. Version Register (VR)

The Version Register (VR) can be read to determine the core version according to the OpenRISC specification. The fields of the register are listed in Table A.3. The configuration field is presently not used. The *ParaNut* -specific configuration, such as the number of CePUs and their supported modes, can be determined through the *ParaNut* -specific registers.

A.2.3. Unit Present Register (UPR)

The fields of the Unit Present Register (UPR) are listed in Table A.4.

A.2.4. CPU Configuration Register (CPUCFGR)

The fields of the CPU Configuration Register (CPUCFGR) are listed in Table A.5. The *ParaNut* can be configured to have either 16 or 32 general purpose registers per CPU. If CGF=1, the number of registers is exactly 16 (this is different from the OR1k specification, which just states that the number of registers is less than 32).

GRP	*REG*	*Name*	*CePU*	*CoPU*	*Description*
0	0	VR	R	–	Version register
0	1	UPR	R	–	Unit Present register
0	2	CPUCFGR	R	–	CPU Configuration register
0	3	DMMUCFGR	R	–	Data MMU Configuration register
0	4	IMMUCFGR	R	–	Instruction MMU Configuration register
0	5	DCCFGR	R	–	Data Cache Configuration register
0	6	ICCFGR	R	–	Instruction Cache Configuration register
0	7	DCFGR	R	–	Debug Configuration register
0	8	PCCFGR	R	–	Performance Counters Configuration register
0	16	PC	R	–	PC mapped to SPR space (*Note:* According to the OR1k specification, NPC should go here! The OR1200 uses PC.)
0	17	SR	RW	RW	Supervision Register
0	18	PPC	R	–	PPC (Previous PC) mapped to SPR space
0	19	FPCSR	R	–	FP Control/Status Register
0	32..47	EPCR0..EPCR15	R	–	Exception PC Registers (all mapped to a single register)
0	48..63	EEAR0-EEAR15	R	–	Exception EA Registers (all mapped to a single register)
0	64..79	ESR0-ESR15	R	–	Exception SR Registers (all mapped to a single register)
0	1024..1055	GPR0..GPR31	RW	–	GPRs mapped to SPR space
3	0	DCCR	R	–	(Data) Cache Control Register
3	2	DCBFR	–	–	DC Block Flush Register
3	3	DCBIR	–	–	DC Block Invalidate Register
3	4	DCBWR	–	–	DC Block Write-back Register
4	(all registers are mapped to the corresponding registers of group 3)				
24	0	PNCPUS	R	–	*ParaNut* Number of CPUs
24	1	PNM2CAP	R	–	*ParaNut* Mode-2 Capability Mask
24	4	PNCE	RW	–	*ParaNut* CPU Enable
24	5	PNLM	RW	–	*ParaNut* Link Mode
24	8	PNX	R	–	*ParaNut* Exception triggered
24	32..64	PNXID0..PNXID31	R	–	*ParaNut* Exception ID

Table A.1.: Special Purpose Registers

Bit(s)	*Name*	*CePU*	*CoPU*	*Reset Value*	*Description*
0	SM	R	–	1	Supervisor Mode
1	TEE	R	–	0	Tick Timer Exception Enabled (*Note:* This bit cannot be set, a tick timer interrupt is not supported)
2	IEE	RW	–	0	Interrupt Exception Enabled
3	DCE	RW	–	0	Data Cache Enable
4	ICE	RW	–	0	Instruction Cache Enable *Note:* This bit is mapped to DCE. To activate the common cache, both DCE and ICE have to be set.
5	DME	R	–	0	Data MMU Enable
6	IME	R	–	0	Instruction MMU Enable
7	LEE	R	–	0	Little Endian Enable
8	CE	R	–	0	CID and shadow register enable
9	F	RW	RW	0	Flag (for conditional branches)
10	CY	RW	RW	0	Carry flag
11	OV	RW	RW	0	Overflow flag
12	OVE	R	–	0	Overflow Exception Enable
13	DSX	R	–	–	Delay Slot Exception 0: EPCR points to instruction outside a delay slot 1: EPCR points to instruction in a delay slot
14	EPH	R	–	0	Exception Prefix High 0: Exceptions vectors are located in memory area starting at 0x0 1: Exception vectors are located in memory area starting at 0xF0000000
15	FO	R	R	1	Fixed One (this bit is alway set)
16	SUMRA	R	–	0	SPRs User Mode Read Access 0: All SPRs are inaccessible in user mode 1: Certain SPRs can be read in user mode
31:28	CID	R	–	0	Context ID (optional)

Table A.2.: Supervision Register (SR)

Bit(s)	*Name*	*Mode*	*Value*	*Description*
0	UP	R	0x1f	Version (0x1f = *ParaNut*)
23:16	CFG	R	0	Configuration (reserved for future use)
15:6	–	R	0	(reserved)
5:0	REV	R	0..63	Revision

Table A.3.: Version Register (VR)

Bit(s)	*Name*	*Mode*	*Reset value*	*Description*
0	UP	R	1	UPR Present
1	DCP	R	1	Data Cache Present
2	ICP	R	1	Instruction Cache Present
3	DMP	R	0	Data MMU Present
4	IMP	R	0	Instruction MMU Present
5	MP	R	0	MAC Present
6	DUP	R	0..1	Debug Unit Present
7	PCUP	R	0	Performance Counters Unit Present
8	PMP	R	0..1	Power Management Present
9	PICP	R	0..1	Programmable Interrupt Controller Present
10	TTP	R	0..1	Tick Timer Present
31:24	CUP	R	0	Custom Units Present

Table A.4.: Unit Present Register (UPR)

Bit(s)	*Name*	*Mode*	*Value*	*Description*
3:0	NSGF	R	0	Number of Shadow GPR Files
4	CGF	R	0..1	Custom GPR File 0: GPR file has 32 registers 1: GPR file has 16 registers
5	OB32S	R	1	ORBIS32 Supported
6	OB64S	R	0	ORBIS64 Supported
7	OF32S	R	0	ORFPX32 Supported
8	OF64S	R	0	ORFP64P Supported
9	OV64S	R	0	ORVDX64 Supported

Table A.5.: CPU Configuration Register (CPUCFGR)

Bit(s)	*Name*	*Mode*	*Value*	*Description*
2:0	NCW	R	0..2	Number of Cache Ways 0: Cache is direct-mapped (one-way) 1. Cache is 2-way set-associative 2: Cache is 4-way set-associative
6:3	NCS	R	0..15	Number of Cache Sets (cache blocks per way) 0: Cache has one set 15: Cache has $2^{15} = 32768$ sets
7	BS	R	0..1	Cache Block Size 0: Cache block has 16 *or fewer* bytes (*OR1k: exactly* 16) 1: Cache block has 32 *or more* bytes (*OR1k: exactly 32*)
8	CWS	R	1	Cache Write Strategy 0: Write-through 1: Write-back
9	CCRI	R	0	Cache Control Register Implemented
10	CBIRI	R	0	Cache Block Invalidate Register Implemented
11	CBPRI	R	0	Cache Block Prefetch Register Implemented
12	CBLRI	R	0	Cache Block Lock Register Implemented
13	CBFRI	R	0	Cache Block Flush Register Implemented
14	CBWBRI	R	0	Cache Block Write-Back Register Implemented

Table A.6.: Data Cache Configuration Register (DCCFGR)

A.2.5. Data Cache Configuration Register (DCCFGR)

The fields of the Data Cache Configuration Register (DCCFGR) are listed in Table A.5. Since the *ParaNut* has a unified cache for data and instructions, the Instruction Cache Configuration Registers (ICCFGR) as specified by the OR1k architecture is mapped to the DCCFGR.

A.3. Exceptions

Table A.7 lists the exceptions supported by the *ParaNut* architecture. Exceptions labelled "(optional)" may not be supported by a particular implementation. The column CoPU indicates whether the exception can occure inside a CoPU.

The *ParaNut* does not support fast context switching. Hence, only one set of exception registers (EPCR, EEAR, ESR) exits. If an exception occurs in the CePU, the following steps are performed:

1. The return adress is stored in register EPCR. If an instruction causes an exeption, it has either completed (e. g. in the case of a system call) or can be restarted (e. g. in the case of a page fault). Depending on this, either the adress of the instruction, or its successor are stored. Special care has to be taken in the following cases:
   - If the exception is caused by an instruction in a delay slot, either the branch target adress (completed instruction) or the address of the branch instruction (restartable instruction) is stored in EPCR.
   - In the case of an "Illegal Instruction" exception, the address of the offending instruction is placed into EEAR, and EPCR points to the next instruction to be executed.
2. In the case of a page fault, the effective address is stored in EEAR.
3. The current SR is stored in ESR.
4. Interrupts are disabled: SR[IEE] = 0.
5. Excecution is continued at the address given by the exception ID multiplied by 0x100.

If an exception occurs inside a CoPU, the following steps occur:

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 abort it. If this is not ensured, the interrupted code is not restartable. [[TBD: Instead of "abort" 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. Inside the CoPU, the registers EPCR (not necessary for mode 1), EEAR and ESR are set as described above.
3. The exception ID is placed into the PN Exception ID register (PNXIDn).
4. The ParaNut Mode Enable register PNME is saved in EPNME.
5. All CoPUs change into the "halt" mode (PNME = 1, only the CePU remains active), and the CePU waits until they actually stop.
6. A CoPU exception is triggered for the CePU.

The exception handler ends by restoring the state of the PNME register and executing the *l.rfe* instruction. This former instruction lets all CPUs start from the CePU's current PC position. Now, they all concurrently execute *l.rfe* and return to the place where they were interrupted.

Name	*ID*	*CoPU*	*Restartable*	*Description*
Reset	0x1	–	–	Caused by hardware reset.
Bus Error	0x2	√	√	The causes are implementation-specific, but typically they are related to bus errors and attempts to access invalid physical address. *Note:* This exception is never asserted in the present version of *ParaNut* .
Data Page Fault	0x3	√	√	*(optional, requires MMU)* No matching page table entry found or page protection violation for load/store operations
Instruction Page Fault	0x4	√	√	*(optional, requires MMU)* No matching page table entry found or page protection violation for instruction fetch operations
Tick Timer	0x5	–	√	*(optional)* Tick timer interrupt asserted. (OR1200: Low priority external interrupt)
Alignment	0x6	√	–	Load/store access to naturally not aligned location.
Illegal Instruction	0x7	√	√	Illegal instruction in the instruction stream.
External Interrupt	0x8	–	√	External interrupt asserted. (OR1200: High priority external interrupt)
D-TLB Miss	0x9	√	√	*(optional, requires MMU)* No matching entry in DTLB (DTLB miss).
I-TLB Miss	0xA	√	√	*(optional, requires MMU)* No matching entry in ITLB (ITLB miss).
Range	0xB	√	–	*(optional)* Asserted, if a) an Overflow occured and SR[OVE] was set, or b) a non-existing general-purpose register has been accessed, if less than 32 GPRs exist.
System Call	0xC	–	√	System call initiated by software.
Trap	0xE	√	√	*(optional)* Caused by the l.trap instruction or by debug unit.
CoPU	0xF	–	(sometimes)	An exception occured inside a CoPU.

Table A.7.: Supported Exceptions

A.4. Testing

l.add – Add Signed

Code:

31 26	25 21	20 16	15 11	10	9 8	7 4	3 0
111000	ddddd	aaaaa	bbbbb	-	00	----	0000

Format: l.add rD, rA, rB

Description: The contents of general-purpose register rA are added to the contents of general-purpose register rB to form the result. The result is placed into general-purpose register rD.

Operation: rD $\leftarrow$ rA + rB

Flags affected: None

Exceptions: None

B. Software Library Reference

The bibliography on the following page was created with the Bibliography environment.

Bibliography

[1] John. L. Hennessy, David A. Patterson: "Computer Architecture: A Quantitative Approach", 4th Edition, Elsevier, 2007

[2] opencores.org: "OpenRISC 1000 Architecture Manual", 2006, www.opencores.org

The above bibliography is created from a BibTEX-database.

Index