OpenRISC 1200

Supplementary Programmer's Reference Manual

Jeremy Bennett, Julius Baxter

Revision 0.2.1
23 Nov 2010

Copyright

Revision History

When updating this revision history, ensure the bookmark *revno* is updated to reside on the latest revision number and *revdate* to reside on the revision date.

Revision	Date	Author	Description
0.0.1	Apr 27, 2010	Jeremy Bennett	Initial draft, covering just PIC
0.0.2	Jun 2, 2010	Jeremy Bennett	Documentation around `lf.madd.s` and FPU SPRs clarified
0.0.3	Jun 9, 2010	Jeremy Bennett	Range exceptions clarified for `l.add`, `l.addc`, `l.addi` and `l.addic` instructions. Exception behavior of `l.div` and `l.divu` instructions clarified.
0.0.4	Jun 11, 2010	Jeremy Bennett	Syntax of `l.ff1` and `l.fl1` corrected. Bit pattern of `l.maci` corrected.
0.0.5	Jun 13, 2010	Jeremy Bennett	Details of `l.mul`, `l.muli` and `l.mul` clarified.
0.0.6	14 Jun 2010	Jeremy Bennett	Alignment behavior of `l.jalr` and `l.jr` clarified.
0.0.7	16 Jun 2010	Jeremy Bennett	Handling of immediate operand to `l.xori` changed.
0.0.8	16 Jun 2010	Jeremy Bennett	More explanation of `l.xori`.
0.1.0	13 Jul 2010	Jeremy Benentt	Version corresponds to Or1ksim 0.4.0. Clarify use of `r9` in `l.jalr` delay slot.
0.2.0	6 Aug 2010	Julius Baxter	Document now specific to latest OR1200 in repository, not just revision 2. Update description of 2 LSBs of PICMR.
0.2.1	23 Nov 2010	Julius Baxter	Updated headers, removing reference to OR1200 version 2. Clarify that clearing bit in PICSR involves writing '0'.

Table of Contents

Acronyms & Abbreviations

ETLA	Extended TLA
PIC	Programmable Interrupt Controller
PICMR	PIC Mask Register
PICSR	PIC Status Register
TLA	Three Letter Acryonym
UPR	Unit Present Register

Table 1-1. Acronyms and Abbreviations

References

1 Damjan Lampret, 6 Sep 01. *OpenRISC 1200 IP Core Specification*, rev 0.7. Available from www.opencores.org.

2 Igor Mohor, 14 Apr 04. *SoC Debug Interface, rev 3.0.* Available from www.opencores.org.

3 OpenCores, 25 Nov 05. *OpenRISC 1000 Architecture Manual*. Available from www.opencores.org.

1 About this Manual

1.1 Introduction

The OpenRISC 1000 system architecture manual [3] defines the architecture for a family of open-source, synthesizable RISC microprocessor cores. The OpenRISC 1200 is a 32-bit implementation of that architecture [1].

A new version of the OpenRISC 1200 was created by engineers from ORSoC AB in 2009 and published at www.opencores.org. This new version makes a number of improvements to the original implementation.

As the basis for design, the architecture and specification documents are fixed. This document exists to capture information about changes from those specifications and to give further detail and clarification where required.

This document is currently an OpenOffice document controlled under Subversion at www.opencores.org. Its dynamic nature means that it would be better represented in a Wiki, when that becomes available.

The chapters of the document match those in the architecture manual for convenience [3]. However this does mean that early revisions have a large number of empty chapters.

1.2 Authors and Contributors

The main authors are shown on the title page. However the contents of this document draw on the contributions of the wider OpenRISC community, who we list here in alphabetical order.

If you have contributed to this manual but your name isn't listed here, it is not meant as a slight—we simply don't know about it. Send an email to the author of the latest revision, and we'll correct the situation.

Name	Contribution
Julius Baxter	Advice on PIC operation
John Eaton	Advice on PIC operation
Raul Fajardo	Advice on PIC operation
Richard Herveille	Advice on instruction behavior

Table 1-2. Contributors to this manual.

1.3 Typography

In this manual, fonts are used as follows:

- Programming examples are shown in a fixed width font, `thus`.
- Emphasis is shown *thus*, strong emphasis, **thus**.
- UPPER CASE items may be either acronyms or register mode fields that can be written by software. Some common acronyms appear in the glossary.

- Square brackets [] indicate an addressed field in a register or a numbered register in a register file.

However users should take advantage of the OpenOffice styles which have been created to enforce this.

1.4 Conventions

`l.mnemonic`	Identifies an ORBIS32/64 instruction.
`lv.mnemonic`	Identifies an ORVDX32/64 instruction.
`lf.mnemonic`	Identifies an ORFPX32/64 instruction.
`0x`	Indicates a hexadecimal number.
`rA`	Instruction syntax used to identify a general purpose register
`REG[FIELD]`	Syntax used to identify specific bit(s) of a general or special purpose register. FIELD can be a name of one bit or a group of bits or a numerical range constructed from two values separated by a colon.
`X`	In certain contexts, this indicates a ‘don't care’.
`N`	In certain contexts, this indicates an undefined numerical value.
`Implementation`	An actual processor implementing the OpenRISC 1000 architecture.
`Unit`	Sometimes referred to as a coprocessor. An implemented unit usually with some special registers and controlling instructions. It can be defined by the architecture or it may be custom.
`Exception`	A vectored transfer of control to supervisor software through an exception vector table. A way in which a processor can request operating system assistance (division by zero, TLB miss, external interrupt etc).
`Privileged`	An instruction (or register) that can only be executed (or accessed) when the processor is in supervisor mode (when SR[SM]=1).

Table 1-3. Conventions

1.5 Numbering

All numbers are decimal or hexadecimal unless otherwise indicated. The prefix 0x indicates a hexadecimal number. Decimal numbers don't have a special prefix. Binary and other numbers are marked with their base.

2 Architecture Overview

3 Addressing Modes and Operand Conventions

4 Register Set

4.1 Features

4.2 Overview

4.3 Special Purpose Registers

Group 11 is shown as being reserved for the Floating Point Unit, although no registers are described (the floating point control status register, FPCSR, is in Group 0).

The floating point opcodes, lf.madd.d and lf.madd.s, in the original architecture manual refer to two floating point SPRs, FPMADDLO and FPMADDHI. However this appears to be an anomaly. This manual documents revised specifications, which are purely register based. See Section 5.4 for details.

4.4 General Purpose Registers

4.5 Support for Custom Number of GPRs

4.6 Supervision Register

4.7 Exception Program Counter Registers

4.8 Exception Effective Address Registers

4.9 Exception Supervision Registers

4.10 Next and Previous Program Counter

4.11 Floating Point Control Status Register

5 Instruction Set

5.1 Features

5.2 Overview

5.3 ORBIS32/64

l.add Add Signed l.add

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	9	8	7	.	.	4	3	.	.	0
opcode 0x38						D					A					B					res	0x0			reserved					opcode 0x0	
6 bits						5 bits					5 bits					5 bits					1	2 bits			4 bits					4 bits	

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.

32-bit Implementation

```
rD[31:0] ← rA[31:0] + rB[31:0]
SR[CY]   ← carry
SR[OV]   ← overflow
```

64-bit Implementation

```
rD[63:0] ← rA[63:0] + rB[63:0]
SR[CY]   ← carry
SR[OV]   ← overflow
```

Exceptions

`Range Exception` on overflow if `SR[OVE]` is set.

Note. This is a clarification of the original manual.

l.addc Add Signed and Carry l.addc

31 26	25 21	20 16	15 11	10	9 . . 8	7 4	3 0
opcode 0x38	D	A	B	res	0x0	reserved	opcode 0x1
6 bits	5 bits	5 bits	5 bits	1	2 bits	4 bits	4 bits

Format:

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

Description

The contents of general purpose register rA are added to the contents of general purpose register rB and carry, SR[CY], to form the result. The result is placed into general purpose register rD.

32-bit Implementation

```
rD[31:0] ← rA[31:0] + rB[31:0] + SR[CY]
SR[CY]   ← carry
SR[OV]   ← overflow
```

64-bit Implementation

```
rD[63:0] ← rA[63:0] + rB[63:0] + SR[CY]
SR[CY]   ← carry
SR[OV]   ← overflow
```

Exceptions

`Range Exception` on overflow if `SR[OVE]` is set.

Note. This is a clarification of the original manual.

l.addi Add Immediate Signed l.addi

31 26	25 21	20 16	15 11	10 . 8	7 0
opcode 0x27	D	A	I		
6 bits	5 bits	5 bits	16 bits		

Format:

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

Description

The immediate value is sign extended and added to the contents of general purpose register rA to form the result. The result is placed into general purpose register rD.

32-bit Implementation

```
rD[31:0] ← rA[31:0] + exts(Immediate)
SR[CY]   ← carry
SR[OV]   ← overflow
```

64-bit Implementation

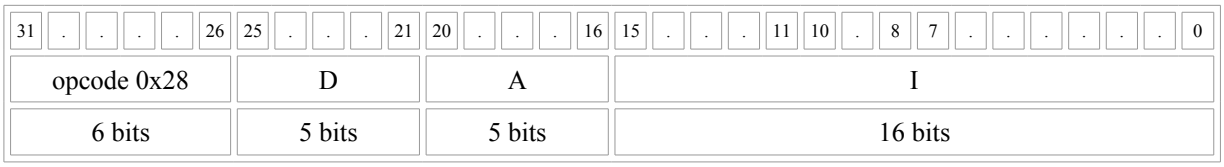
```
rD[63:0] ← rA[63:0] + exts(Immediate)
SR[CY]   ← carry
SR[OV]   ← overflow
```

Exceptions

`Range Exception` on overflow if `SR[OVE]` is set.

Note. This is a clarification of the original manual.

l.addic Add Immediate Signed and Carry l.addic



31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0x28	D	A	I		
6 bits	5 bits	5 bits	16 bits		

Format:

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

Description

The immediate value is sign extended and added to the contents of general purpose register rA and carry, SR[CY], to form the result. The result is placed into general purpose register rD.

32-bit Implementation

```
rD[31:0] ← rA[31:0] + exts(Immediate) + SR[CY]
SR[CY]   ← carry
SR[OV]   ← overflow
```

64-bit Implementation

```
rD[63:0] ← rA[63:0] + exts(Immediate) + SR[CY]
SR[CY]   ← carry
SR[OV]   ← overflow
```

Exceptions

`Range Exception` on overflow if `SR[OVE]` is set.

Note. This is a clarification of the original manual.

l.div Divide Signed l.div

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	.	0
opcode 0x38						D					A					B					res		0x3				reserved				opcode 0x9	
6 bits						5 bits					5 bits					5 bits					1		2 bits				4 bits				4 bits	

Format:

```
l.div  rD,rA,rB
```

Description

The contents of general purpose register rA are divided by the contents of general purpose register rB, to form the result. Both operands are treated as signed integers. The result is placed into general purpose register rD. The carry flag, SR[CY] is set when the divisor is zero.

Note. Change from original manual. Correct values shown for carry and overflow.

32-bit Implementation

```
rD[31:0] ← rA[31:0] / rB[31:0]
SR[CY]   ← rB[31:0] == 0
SR[OV]   ← 0
```

64-bit Implementation

```
rD[63:0] ← rA[63:0] / rB[63:0]
SR[CY]   ← rB[63:0] == 0
SR[OV]   ← 0
```

Exceptions

`Range exception` when divisor is zero if `SR[OVE]` is set.

Note. This is a clarification of the original manual, which does not specify that SR[OVE] must be set.

l.divu Divide Unsigned l.divu

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0x38	D	A	B	res	0x3	reserved	opcode 0xa
6 bits	5 bits	5 bits	/* Div by zero sets carry */5 bits	1	2 bits	4 bits	4 bits

Format:

```
l.div  rD,rA,rB
```

Description

The contents of general purpose register rA are divided by the contents of general purpose register rB, to form the result. Both operands are treated as unsigned integers. The result is placed into general purpose register rD. The carry flag, SR[CY] is set when the divisor is zero.

Note. Change from original manual. Correct values shown for carry and overflow.

32-bit Implementation

```
rD[31:0] ← rA[31:0] / rB[31:0]
SR[CY]   ← rB[31:0] == 0
SR[OV]   ← 0
```

64-bit Implementation

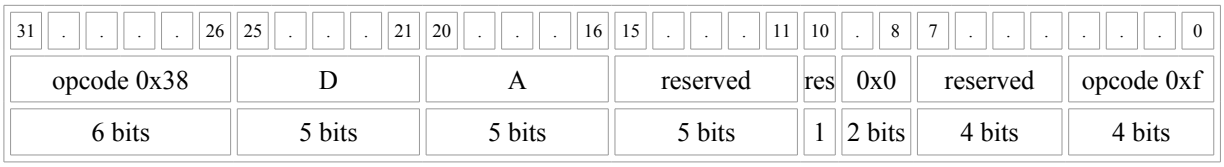
```
rD[63:0] ← rA[63:0] / rB[63:0]
SR[CY]   ← rB[63:0] == 0
SR[OV]   ← 0
```

Exceptions

`Range exception` when divisor is zero if `SR[OVE]` is set.

Note. This is a clarification of the original manual, which does not specify that SR[OVE] must be set.

l.ff1 Find First 1 l.ff1



31 26	25 . . . 21	20 . . . 16	15 . . . 11	10	. 8	7 0
opcode 0x38	D	A	reserved	res	0x0	reserved	opcode 0xf
6 bits	5 bits	5 bits	5 bits	1	2 bits	4 bits	4 bits

Format:

```
l.ff1  rD,rA
```

Description

Position of the first '1' bit is written into general-purpose register rD. Checking for bit '1' starts with bit 0 (LSB), and counting is incremented for every zero bit. If first '1' bit is discovered in LSB, one is written into rD, if first ‘1’ bit is discovered in MSB, 32 is written into rD. If there is no '1' bit, zero is written in rD.

32-bit Implementation

```
rD[31:0] ← rA[0] ? 1 : rA[1] ? 2 ... rA[31] ? 32 : 0
```

64-bit Implementation

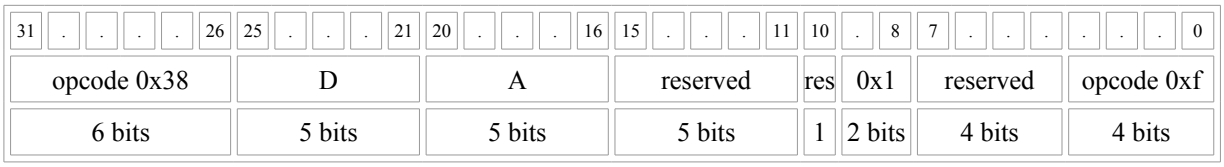
```
rD[63:0] ← rA[0] ? 1 : rA[1] ? 2 ... rA[63] ? 64 : 0
```

Exceptions

None.

Note. The original manual shows this opcode with three operands. The first reserved field would usually provide a rB, but this is not used here.

l.fl1 Find Last 1 l.fl1



31 26	25 21	20 16	15 11	10	. . 8	7 0
opcode 0x38	D	A	reserved	res	0x1	reserved	opcode 0xf
6 bits	5 bits	5 bits	5 bits	1	2 bits	4 bits	4 bits

Format:

```
l.fl1   rD,rA
```

Description

Position of the last '1' bit is written into general-purpose register rD. Checking for bit '1' starts with bit 0 (LSB), and counting is inccremented for every zero bit until the last ‘1’ bit is found nearing the MSB. If first '1' bit is discovered in bit 32(64) MSB, 32 (64) is written into rD, if first ‘1’ bit is discovered in LSB, one is written into rD. If there is no '1' bit, zero is written in rD.

32-bit Implementation

```
rD[31:0] ← rA[31] ? 32 : rA[30] ? 31 ... rA[0] ? 1 : 0
```

64-bit Implementation

```
rD[63:0] ← rA[63] ? 64 : rA[62] ? 63 ... rA[0] ? 1 : 0
```

Exceptions

None.

Note. The original manual shows this opcode with three operands. The first reserved field would usually provide a rB, but this is not used here.

l.jalr Jump and Link Register l.jalr

31 26	25 16	15 . . . 11	10 0
opcode 0x12	reserved	B	reserved
6 bits	10 bits	5 bits	11 bits

Format:

```
l.jalr  rB
```

Description

The contents of general-purpose register rB is the effective address of the jump. The program unconditionally jumps to EA with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register. It is not allowed to specify the link register as rB. The value of the link register, if read as an operand in the delay slot will be the *new* value, *not* the old value. If the link register is written in the delay slot, the value written will replace the value stored by the `l.jalr` instruction.

32-bit Implementation

```
PC[31:0] ← rB[31:0]
LR[31:0] ← DelayInsnAddr + 4
```

64-bit Implementation

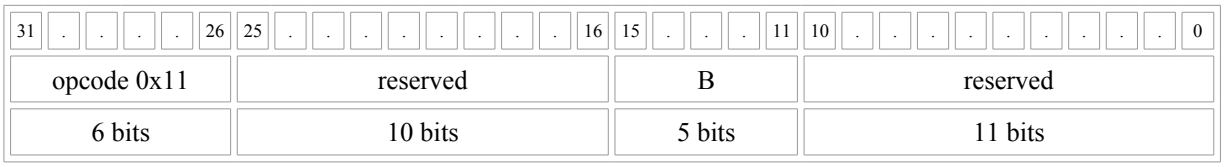
```
PC[63:0] ← rB[63:0]
LR[63:0] ← DelayInsnAddr + 4
```

Exceptions

`ALIGN Exception` if the address in rB is not word aligned. `ILLEGAL Exception` if the link register is specified as rB.

Note. The original manual makes no specification of exception behavior, nor does it specify the value that the link register will take if used as an operand in the delay slot. The assembler will generally prevent the link register being used as rB.

l.jr Jump Register l.jr



31 26	25 16	15 . . . 11	10 0
opcode 0x11	reserved	B	reserved
6 bits	10 bits	5 bits	11 bits

Format:

```
l.jr  rB
```

Description

The contents of general-purpose register rB is the effective address of the jump. The program unconditionally jumps to EA with a delay of one instruction.

32-bit Implementation

```
PC[31:0] ← rB[31:0]
```

64-bit Implementation

```
PC[63:0] ← rB[63:0]
```

Exceptions

`ALIGN Exception` if the address in rB is not word aligned.

Note. The original manual makes no specification of exception behavior.

l.maci Multiply Immediate Signed and Accumulate l.maci

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	.	.	.	.	.	.	.	.	.	.	0
opcode 0x13						reserved					A					I														
6 bits						5 bits					5 bits					16 bits														

Format:

```
l.maci  rA,I
```

Description

The immediate value and the contents of general-purpose register rA are multiplied, and the result is truncated to 32 bits, then sign extended and added to the special- purpose registers MACHI and MACLO. All operands are treated as signed integers.

32-bit Implementation

```
temp1[31:0]  ← MACLO[31:0]
temp1[63:32] ← MACHI[31:0]
temp2[31:0]  ← rA[31:0] * exts(Immediate)
temp1[63:0]  ← temp1[63:0] + exts(temp2[31:0])
MACLO[31:0]  ← temp1[31:0]
MACHI[31:0]  ← temp1[63:32]
```

64-bit Implementation

```
temp1[31:0]  ← MACLO[31:0]
temp1[63:32] ← MACHI[31:0]
temp2[31:0]  ← rA[63:0] * exts(Immediate)
temp1[63:0]  ← temp1[63:0] + exts(temp2[31:0])
MACLO[31:0]  ← temp1[31:0]
MACHI[31:0]  ← temp1[63:32]
```

Exceptions

None.

Note. The original manual shows this opcode with a different bit pattern. The first reserved field would usually provide rD, but this is not used here. Clarified that the results of multiplication is sign-extended before adding in.

l.mul Multiply Signed l.mul

31 26	25 21	20 16	15 11	10	9 8	7 . . . 4	3 . . 0
opcode 0x38	D	A	B	res	0x3	reserved	opcode 0x6
6 bits	5 bits	5 bits	5 bits	1	2 bits	4 bits	4 bits

Format:

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

Description

The contents of general-purpose register rA and the contents of general- purpose register rB are multiplied, and the result is truncated to destination register width and placed into general-purpose register rD. Both operands are treated as signed integers.

32-bit Implementation

```
rD[31:0] ← rA[31:0] * rB[31:0]
SR[CY]   ← carry
SR[OV]   ← overflow
```

64-bit Implementation

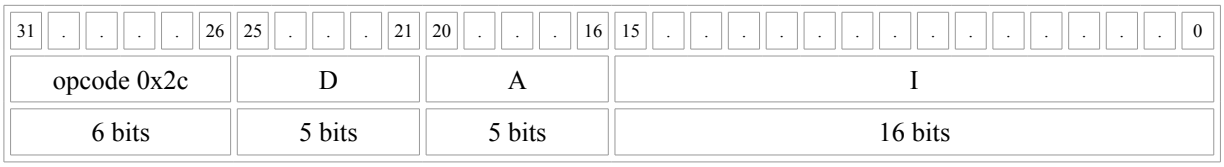
```
rD[63:0] ← rA[63:0] * rB[63:0]
SR[CY]   ← carry
SR[OV]   ← overflow
```

Exceptions

`Range Exception` on overflow if `SR[OVE]` is set.

Note. This is a clarification of the original manual.

l.muli Multiply Signed l.muli



31 26	25 . . . 21	20 . . . 16	15 0
opcode 0x2c	D	A	I
6 bits	5 bits	5 bits	16 bits

Format:

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

Description

```
SR[OV]   ← overflow
```

Exceptions

`Range Exception` on overflow if `SR[OVE]` is set.

Note. This is a clarification of the original manual.

The immediate value is sign extended and multiplied by the contents of general-purpose register rA. The result is truncated to destination register width and placed into general-purpose register rD. Both operands are treated as signed integers.

32-bit Implementation

```
rD[31:0] ← rA[31:0] * exts(Immediate)
SR[CY]   ← carry
SR[OV]   ← overflow
```

64-bit Implementation

```
rD[63:0] ← rA[63:0] + exts(Immediate)
SR[CY]   ← carry
SR[OV]   ← overflow
```

Exceptions

`Range Exception` on overflow if `SR[OVE]` is set.

Note. This is a clarification of the original manual.

l.mulu Multiply Unsigned l.mulu

31 26	25 21	20 16	15 11	10	9 8	7 . . . 4	3 . . 0
opcode 0x38	D	A	B	res	0x3	reserved	opcode 0xb
6 bits	5 bits	5 bits	5 bits	1	2 bits	4 bits	4 bits

Format:

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

Description

The contents of general-purpose register rA and the contents of general- purpose register rB are multiplied, and the result is truncated to destination register width and placed into general-purpose register rD. Both operands are treated as unsigned integers.

32-bit Implementation

```
rD[31:0] ← rA[31:0] * rB[31:0]
SR[CY]   ← carry
SR[OV]   ← 0
```

64-bit Implementation

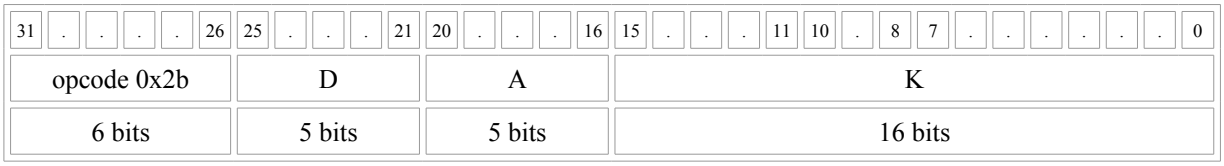
```
rD[63:0] ← rA[63:0] * rB[63:0]
SR[CY]   ← carry
SR[OV]   ← 0
```

Exceptions

None.

Note. This is a clarification of the original manual, which suggested setting the overflow flag and potentially raising a `Range Exception`.

l.xori Exclusive Or with Immediate Half Word l.xori



Format:

```
l.xori  rD,rA,K
```

Description

The immediate value is zero-extended and combined with the contents of general-purpose register rA in a bit-wise logical XOR operation. The result is placed into general-purpose register rD.

32-bit Implementation

```
rD[31:0] ← rA[31:0] XOR extz(Immediate)
```

64-bit Implementation

```
rD[63:0] ← rA[63:0] XOR extz(Immediate)
```

Exceptions

None.

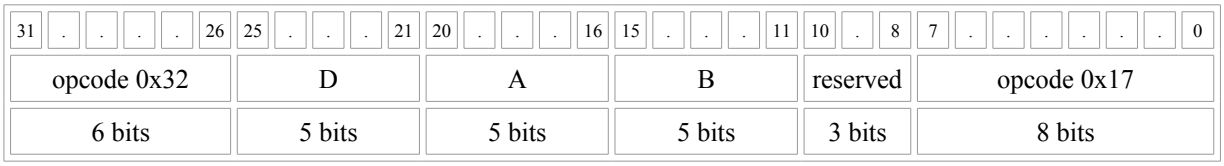
Note. The original manual specified that the immediate value was sign-extended, and inconsistency with `l.andi` and `l.ori`. This means that `l.xori rD,rA,-1` can be used in the absence of a NOT instruction. This change of functionality has a significant impact, because of the heavy use in this way by GCC. Thus this manual documents an aspiration for behavior, that is not yet provided in current implementations.

5.4 ORFPX32/64

This section is not separated from ORBIS32/64 in the original manual.

This manual documents changed instructions only.

lf.madd.d Multiply and Add Floating Point Double Precision lf.madd.d



31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0x32	D	A	B	reserved	opcode 0x17
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.madd.d  rD,rA,rB
```

Description

The contents of general purpose register rA are multipled by the contents of general purpose register rB, and added to the general purpose register rD.

32-bit Implementation

```
N/A
```

64-bit Implementation

```
rD[63:0] ← rA[63:0] * rB[63:0] + rD[63:0]
```

Exceptions

```
Floating Point
```

lf.madd.s Multiply and Add Floating Point Single Precision lf.madd.s

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	.	0
opcode 0x32						D					A					B					reserved			opcode 0x7							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

Format:

```
lf.madd.s  rD,rA,rB
```

Description

The contents of general purpose register rA are multipled by the contents of general purpose register rB, and added to the general purpose register rD.

32-bit Implementation

```
rD[31:0] ← rA[31:0] * rB[31:0] + rD[31:0]
```

64-bit Implementation

```
rD[31:0]  ← rA[31:0] * rB[31:0] + rD[31:0]
rD[63:32] ← 0
```

Exceptions

```
Floating Point
```

5.5 ORVDX64

This section is not separated from ORBIS32/64 in the original manual.

6 Exception Model

7 Memory Model

8 Memory Management

9 Cache Model and Cache Coherency

10 Debug Unit (Optional)

The debug interface from a programmer's perspective is unchanged in the OpenRISC 1200. However a new physical debug interface has been adopted [2]. This will affect those concerned with the representation of JTAG packets to drive the debug interface.

11 Performance Counters Unit (Optional)

12 Power Management (Optional)

13 Programmable Interrupt Controller (Optional)

The OpenRISC 1000 Architecture Manual [] describes an optional simple programmer interrupt controller (PIC) capable of handling up to 32 separate interrupt lines driving the Interrupt Exception. The presence of the PIC is indicated in the Unit Present Register bby UPR[PICP].

The PIC is controlled by two registers, PICMR and PICSR. PICMR is a mask for the incoming interrupts. PICSR is a status register indicating which interrupt lines have been asserted. By convention, interrupts 0 and 1 in PICMR are tied to 1, so that these interrupts lines are non-maskable.

There are two permanently unmasked interrupt lines, [1:0], with PICMR[1:0] fixed to one.

A simple block diagram of the PIC is shown in Figure 13-1.

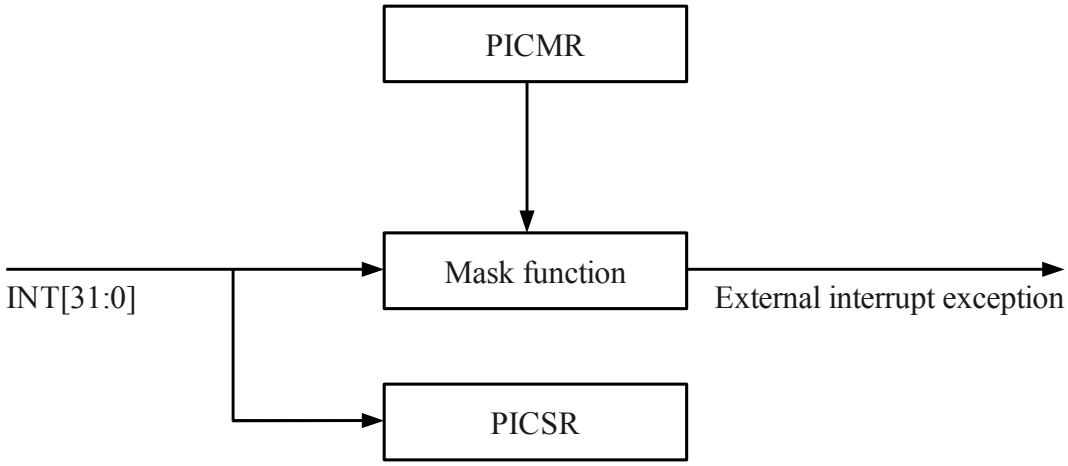


Figure 13-1: Testing

13.1 Functionality

In the OpenRISC 1200, the differs from the architecture specification. The PIC offers a latched level-sensitive interrupt.

Once an interrupt line is latched (i.e. its value appears in PICSR), no new interrupts can be triggered for that line until its bit in PICSR is cleared. The usual sequence for an interrupt handler is then as follows.

1 Peripheral asserts interrupt, which is latched and triggers handler.

2 Handler processes interrupt.

3 Handler notifies peripheral that the interrupt has been processed (typically via a memory mapped register).

4 Peripheral deasserts interrupt.

5 Handler clears corresponding bit in PICSR and returns.

It is assumed that the peripheral will deassert its interrupt promptly (within 1-2 cycles). Otherwise on exiting the interrupt handler, having cleared PICSR, the level sensitive interrupt will immediately retrigger.

13.2 PIC Mask Register (PICMR)

The interrupt controller mask register is a 32-bit special-purpose supervisor-level register accessible with the l.mtspr/l.mfspr instructions in supervisor mode.

PICMR is used to mask or unmask 30 programmable interrupt sources.

Bit	31-2	1-0
Identifier	IUM	FOM
Reset	0	0x3
R/W	R/W	R/O

IUM	Interrupt UnMask 0x00000000 All interrupts are masked 0x00000001 Interrupt input 0 is enabled, all others are masked … 0xFFFFFFFF All interrupt inputs are enabled
FOM	Fixed One Mask

Table 13-4. PICMR Field Descriptions

13.3 PIC Status Register (PICSR)

The atomic way to clear an interrupt source is by first clearing it at the external source, then by writing a '0' to the corresponding bit in the PICSR. This will clear the underlying latch for the edge-triggered source, iff the external line is also de-asserted.

14 Tick Timer Facility (Optional)

15 OpenRISC 1000 Implementations

16 Application Binary Interface

There have been some minor changes in the ABI used with the OpenRISC 1200 version 2.