

# OpenRISC 1000 Architecture Manual<sup>1</sup>

Architecture Version 1.3

Document Revision 1

May 26, 2019

<sup>1</sup>Copyright © 2000-2019 OPENRISC.IO and Authors

This document is free; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

# Table of Contents

<b>1 ABOUT THIS MANUAL.....</b>	<b>10</b>
1.1 INTRODUCTION.....	10
1.2 AUTHORS.....	10
1.3 DOCUMENT REVISION HISTORY.....	12
1.4 WORK IN PROGRESS.....	15
1.5 FONTS IN THIS MANUAL.....	15
1.6 CONVENTIONS.....	16
1.7 NUMBERING.....	16
<b>2 ARCHITECTURE OVERVIEW.....</b>	<b>17</b>
2.1 FEATURES.....	17
2.2 INTRODUCTION.....	18
2.3 ARCHITECTURE VERSION INFORMATION.....	18
<b>3 ADDRESSING MODES AND OPERAND CONVENTIONS.....</b>	<b>19</b>
3.1 MEMORY ADDRESSING MODES.....	19
3.2 MEMORY OPERAND CONVENTIONS.....	20
<b>4 REGISTER SET.....</b>	<b>23</b>
4.1 FEATURES.....	23
4.2 OVERVIEW.....	23
4.3 SPECIAL-PURPOSE REGISTERS.....	23
4.4 GENERAL-PURPOSE REGISTERS (GPRS).....	28
4.5 SUPPORT FOR CUSTOM NUMBER OF GPRS.....	28
4.6 SUPERVISION REGISTER (SR).....	29
4.7 EXCEPTION PROGRAM COUNTER REGISTERS (EPCR0 - EPCR15).....	31
4.8 EXCEPTION EFFECTIVE ADDRESS REGISTERS (EEAR0-EEAR15).....	31
4.9 EXCEPTION SUPERVISION REGISTERS (ESR0-ESR15).....	32
4.10 CORE IDENTIFICATION REGISTERS (COREID AND NUMCORES).....	32
4.11 NEXT AND PREVIOUS PROGRAM COUNTER (NPC AND PPC).....	32
4.12 FLOATING POINT CONTROL STATUS REGISTER (FPCSR).....	32
<b>5 INSTRUCTION SET.....</b>	<b>35</b>
5.1 FEATURES.....	35
5.2 OVERVIEW.....	35
5.3 ORBIS32/64.....	37
5.4 ORFPX32/64.....	136
5.5 ORFPX64A32.....	175
5.6 ORVDX64.....	176
<b>6 EXCEPTION MODEL.....</b>	<b>267</b>
6.1 INTRODUCTION.....	267

6.2 EXCEPTION CLASSES.....	267
6.3 EXCEPTION PROCESSING.....	269
6.4 FAST CONTEXT SWITCHING (OPTIONAL).....	270
<b>7 MEMORY MODEL.....</b>	<b>273</b>
7.1 MEMORY.....	273
7.2 MEMORY ACCESS ORDERING.....	273
7.3 ATOMICITY.....	274
<b>8 MEMORY MANAGEMENT.....</b>	<b>275</b>
8.1 MMU FEATURES.....	275
8.2 MMU OVERVIEW.....	275
8.3 MMU EXCEPTIONS.....	277
8.4 MMU SPECIAL-PURPOSE REGISTERS.....	277
8.5 ADDRESS TRANSLATION MECHANISM IN 32-BIT IMPLEMENTATIONS.....	290
8.6 ADDRESS TRANSLATION MECHANISM IN 64-BIT IMPLEMENTATIONS.....	293
8.7 MEMORY PROTECTION MECHANISM.....	296
8.8 PAGE TABLE ENTRY DEFINITION.....	297
8.9 PAGE TABLE SEARCH OPERATION.....	298
8.10 PAGE HISTORY RECORDING.....	299
8.11 PAGE TABLE UPDATES.....	299
<b>9 CACHE MODEL &amp; CACHE COHERENCY.....</b>	<b>300</b>
9.1 CACHE SPECIAL-PURPOSE REGISTERS.....	300
9.2 CACHE MANAGEMENT.....	302
9.3 CACHE/MEMORY COHERENCY.....	306
<b>10 MULTICORE SUPPORT.....</b>	<b>309</b>
10.1 INTRODUCTION.....	309
10.2 INTER PROCESSOR COMMUNICATION.....	309
10.3 TEMPORARY STORAGE.....	312
10.4 MULTICORE BOOTSTRAPPING.....	312
10.5 TIMER SYNCHRONIZATION.....	312
<b>11 DEBUG UNIT (OPTIONAL).....</b>	<b>313</b>
11.1 FEATURES.....	313
11.2 DEBUG VALUE REGISTERS (DVR0-DVR7).....	314
11.3 DEBUG CONTROL REGISTERS (DCR0-DCR7).....	314
11.4 DEBUG MODE REGISTER 1 (DMR1).....	315
11.5 DEBUG MODE REGISTER 2(DMR2).....	317
11.6 DEBUG WATCHPOINT COUNTER REGISTER (DWCR0-DWCR1).....	318
11.7 DEBUG STOP REGISTER (DSR).....	319
11.8 DEBUG REASON REGISTER (DRR).....	320
<b>12 PERFORMANCE COUNTERS UNIT (OPTIONAL).....</b>	<b>323</b>
12.1 FEATURES.....	323
12.2 PERFORMANCE COUNTERS COUNT REGISTERS (PCCR0-PCCR7).....	323
12.3 PERFORMANCE COUNTERS MODE REGISTERS (PCMR0-PCMR7).....	324
<b>13 POWER MANAGEMENT (OPTIONAL).....</b>	<b>326</b>

13.1 FEATURES.....	326
13.2 POWER MANAGEMENT REGISTER (PMR).....	327
<b>14 PROGRAMMABLE INTERRUPT CONTROLLER (OPTIONAL).....</b>	<b>328</b>
14.1 FEATURES.....	328
14.2 PIC MASK REGISTER (PICMR).....	328
14.3 PIC STATUS REGISTER (PICSR).....	329
<b>15 TICK TIMER FACILITY (OPTIONAL).....</b>	<b>330</b>
15.1 FEATURES.....	330
15.2 TIMER INTERRUPTS.....	331
15.3 TIMER MODES.....	331
15.4 TICK TIMER MODE REGISTER (TTMR).....	332
15.5 TICK TIMER COUNT REGISTER (TTCR).....	333
<b>16 OPENRISC 1000 IMPLEMENTATIONS.....</b>	<b>334</b>
16.1 OVERVIEW.....	334
16.2 VERSION REGISTER (VR).....	334
16.3 UNIT PRESENT REGISTER (UPR).....	335
16.4 CPU CONFIGURATION REGISTER (CPUCFGR).....	336
16.5 DMMU CONFIGURATION REGISTER (DMMUCFGR).....	338
16.6 IMMU CONFIGURATION REGISTER (IMMUCFGR).....	339
16.7 DC CONFIGURATION REGISTER (DCCFGR).....	340
16.8 IC CONFIGURATION REGISTER (ICCFGR).....	341
16.9 DEBUG CONFIGURATION REGISTER (DCFGR).....	342
16.10 PERFORMANCE COUNTERS CONFIGURATION REGISTER (PCCFGR).....	343
16.11 VERSION REGISTER 2 (VR2).....	343
16.12 ARCHITECTURE VERSION REGISTER (AVR).....	344
16.13 EXCEPTION VECTOR BASE ADDRESS REGISTER (EVBAR).....	344
16.14 ARITHMETIC EXCEPTION CONTROL REGISTER (AECR).....	345
16.15 ARITHMETIC EXCEPTION STATUS REGISTER (AESR).....	346
16.16 IMPLEMENTATION-SPECIFIC REGISTERS (ISR0-7).....	347
<b>17 APPLICATION BINARY INTERFACE.....</b>	<b>348</b>
17.1 DATA REPRESENTATION.....	348
17.2 FUNCTION CALLING SEQUENCE.....	351
17.3 OPERATING SYSTEM INTERFACE.....	354
17.4 POSITION-INDEPENDENT CODE.....	357
17.5 ELF.....	357
<b>18 MACHINE CODE REFERENCE.....</b>	<b>359</b>
<b>19 INDEX.....</b>	<b>376</b>

## Table Of Figures

FIGURE 3-1. REGISTER INDIRECT WITH DISPLACEMENT ADDRESSING.....	19
FIGURE 3-2. PC RELATIVE ADDRESSING.....	20
FIGURE 5-1. INSTRUCTION SET.....	35
FIGURE 8-1. TRANSLATION OF EFFECTIVE TO PHYSICAL ADDRESS – SIMPLIFIED BLOCK DIAGRAM FOR 32-BIT PROCESSOR IMPLEMENTATIONS.....	276
FIGURE 8-2. MEMORY DIVIDED INTO L1 AND L2 PAGES.....	290
FIGURE 8-3. ADDRESS TRANSLATION MECHANISM USING TWO-LEVEL PAGE TABLE.....	291
FIGURE 8-4. ADDRESS TRANSLATION MECHANISM USING ONLY L1 PAGE TABLE.....	292
FIGURE 8-5. MEMORY DIVIDED INTO L0, L1 AND L2 PAGES.....	293
FIGURE 8-6. ADDRESS TRANSLATION MECHANISM USING THREE-LEVEL PAGE TABLE .....	294
FIGURE 8-7. ADDRESS TRANSLATION MECHANISM USING TWO-LEVEL PAGE TABLE.....	295
FIGURE 8-8. SELECTION OF PAGE PROTECTION ATTRIBUTES FOR DATA ACCESSES.....	297
FIGURE 8-9. SELECTION OF PAGE PROTECTION ATTRIBUTES FOR INSTRUCTION FETCH ACCESSES.....	297
FIGURE 8-10. PAGE TABLE ENTRY FORMAT.....	298
FIGURE 10-1: MULTICORE INTERCONNECT WITH OMPIC.....	310
FIGURE 11-1. BLOCK DIAGRAM OF DEBUG SUPPORT.....	314
FIGURE 14-1. PROGRAMMABLE INTERRUPT CONTROLLER BLOCK DIAGRAM.....	328
FIGURE 15-1. TICK TIMER BLOCK DIAGRAM.....	330
FIGURE 17-1. BYTE ALIGNED, SIZEOF IS 1.....	349
FIGURE 17-2. NO PADDING, SIZEOF IS 8.....	350
FIGURE 17-3. PADDING, SIZEOF IS 16.....	350
FIGURE 17-4. STORAGE UNIT SHARING AND ALIGNMENT PADDING, SIZEOF IS 12.....	351

## Table Of Tables

TABLE 1. ACRONYMS AND ABBREVIATIONS.....	9
TABLE 1-1. AUTHORS OF THIS MANUAL.....	11
TABLE 1-2. REVISION HISTORY.....	15
TABLE 1-3. CONVENTIONS.....	16
TABLE 2-1: ARCHITECTURE VERSION INFORMATION.....	18
TABLE 3-1. MEMORY OPERANDS AND THEIR SIZES.....	21
TABLE 3-2. DEFAULT BIT AND BYTE ORDERING IN HALFWORDS.....	21
TABLE 3-3. DEFAULT BIT AND BYTE ORDERING IN SINGLEWORDS AND SINGLE PRECISION FLOATS.....	21
TABLE 3-4. DEFAULT BIT AND BYTE ORDERING IN DOUBLEWORDS, DOUBLE PRECISION FLOATS AND ALL VECTOR TYPES.....	22
TABLE 3-5. MEMORY OPERAND ALIGNMENT.....	22
TABLE 4-1. GROUPS OF SPRS.....	24
TABLE 4-2. LIST OF ALL SPECIAL-PURPOSE REGISTERS.....	27
TABLE 4-3. GENERAL-PURPOSE REGISTERS.....	28
TABLE 4-4. SR FIELD DESCRIPTIONS.....	30
TABLE 4-5. EPCR FIELD DESCRIPTIONS.....	31
TABLE 4-6. EEAR FIELD DESCRIPTIONS.....	31
TABLE 4-7. ESR FIELD DESCRIPTIONS.....	32
TABLE 4-8. FPCSR FIELD DESCRIPTIONS.....	34
TABLE 5-1. OPENRISC 1000 INSTRUCTION CLASSES.....	36
TABLE 6-1. EXCEPTION CLASSES.....	267
TABLE 6-2. EXCEPTION TYPES AND CAUSAL CONDITIONS.....	268
TABLE 6-3. VALUES OF EPCR AND EEAR AFTER EXCEPTION.....	270
TABLE 8-1. MMU EXCEPTIONS.....	277
TABLE 8-2. LIST OF MMU SPECIAL-PURPOSE REGISTERS.....	279
TABLE 8-3. DMMUCR FIELD DESCRIPTIONS.....	279
TABLE 8-4. DMMUPR FIELD DESCRIPTIONS.....	280
TABLE 8-5. IMMUCR FIELD DESCRIPTIONS.....	281
TABLE 8-6. IMMUPR FIELD DESCRIPTIONS.....	282
TABLE 8-7. XTLBEIR FIELD DESCRIPTIONS.....	282
TABLE 8-8. XTLBMR FIELD DESCRIPTIONS.....	283
TABLE 8-9. DTLBTR FIELD DESCRIPTIONS.....	285
TABLE 8-10. ITLBWYTR FIELD DESCRIPTIONS.....	286

TABLE 8-11. XATBMR FIELD DESCRIPTIONS.....	287
TABLE 8-12. DATBTR FIELD DESCRIPTIONS.....	288
TABLE 8-13. IATBTR FIELD DESCRIPTIONS.....	289
TABLE 8-14. PROTECTION ATTRIBUTES.....	296
TABLE 8-15. PTE FIELD DESCRIPTIONS.....	298
TABLE 9-1. CACHE REGISTERS.....	301
TABLE 9-2. DCCR FIELD DESCRIPTIONS.....	301
TABLE 9-3. ICCR FIELD DESCRIPTIONS.....	302
TABLE 9-4. DCBPR FIELD DESCRIPTIONS.....	302
TABLE 9-5. DCBFR FIELD DESCRIPTIONS.....	303
TABLE 9-6. DCBIR FIELD DESCRIPTIONS.....	304
TABLE 9-7. DCBWR FIELD DESCRIPTIONS.....	304
TABLE 9-8. DCBLR FIELD DESCRIPTIONS.....	305
TABLE 9-9. ICBPR FIELD DESCRIPTIONS.....	305
TABLE 9-10. ICBIR FIELD DESCRIPTIONS.....	306
TABLE 9-11. ICBLR FIELD DESCRIPTIONS.....	306
TABLE 10-1. OMPIC CONTROL FIELD DESCRIPTIONS.....	311
TABLE 10-2. OMPIC STATUS FIELD DESCRIPTIONS.....	311
TABLE 11-1. DVR FIELD DESCRIPTIONS.....	314
TABLE 11-2. DCR FIELD DESCRIPTIONS.....	315
TABLE 11-3. DMR1 FIELD DESCRIPTIONS.....	317
TABLE 11-4. DMR2 FIELD DESCRIPTIONS.....	318
TABLE 11-5. DWCR FIELD DESCRIPTIONS.....	319
TABLE 11-6. DSR FIELD DESCRIPTIONS.....	320
TABLE 11-7. DRR FIELD DESCRIPTIONS.....	322
TABLE 12-1. PCCR0 FIELD DESCRIPTIONS.....	324
TABLE 12-2. PCMR FIELD DESCRIPTIONS.....	325
TABLE 13-1. PMR FIELD DESCRIPTIONS.....	327
TABLE 14-1. PICMR FIELD DESCRIPTIONS.....	329
TABLE 14-2. PICSR FIELD DESCRIPTIONS.....	329
TABLE 15-1. TTMR FIELD DESCRIPTIONS.....	332
TABLE 15-2. TTCR FIELD DESCRIPTIONS.....	333
TABLE 16-1. VR FIELD DESCRIPTIONS.....	335
TABLE 16-2. UPR FIELD DESCRIPTIONS.....	336
TABLE 16-3. CPUCFGR FIELD DESCRIPTIONS.....	337
TABLE 16-4. DMMUCFGR FIELD DESCRIPTIONS.....	339
TABLE 16-5. IMMUCFGR FIELD DESCRIPTIONS.....	340

TABLE 16-6. DCCFGR FIELD DESCRIPTIONS.....	341
TABLE 16-7. ICCFGR FIELD DESCRIPTIONS.....	342
TABLE 16-8. DCFGR FIELD DESCRIPTIONS.....	342
TABLE 16-9. PCCFGR FIELD DESCRIPTIONS.....	343
TABLE 16-10. VR2 FIELD DESCRIPTIONS.....	344
TABLE 16-11. AVR FIELD DESCRIPTIONS.....	344
TABLE 16-12. EVBAR FIELD DESCRIPTIONS.....	344
TABLE 16-13. EACR FIELD DESCRIPTIONS.....	345
TABLE 16-14. EASR FIELD DESCRIPTIONS.....	347
TABLE 17-1. SCALAR TYPES.....	348
TABLE 17-2. VECTOR TYPES.....	349
TABLE 17-3. BIT-FIELD TYPES AND RANGES.....	350
TABLE 17-4. GENERAL-PURPOSE REGISTERS.....	352
TABLE 17-5. STACK FRAME.....	353
TABLE 17-6. HARDWARE EXCEPTIONS AND SIGNALS.....	355
TABLE 17-7. VIRTUAL ADDRESS CONFIGURATION.....	356
TABLE 17-8. <i>E_IDENT</i> FIELD VALUES.....	357
TABLE 17-9. <i>E_FLAGS</i> FIELD VALUES.....	358



## Acronyms & Abbreviations

ALU	Arithmetic Logic Unit
ATB	Area Translation Buffer
BIU	Bus Interface Unit
BTC	Branch Target Cache
CPU	Central Processing Unit
DC	Data Cache
DMMU	Data MMU
DTLB	Data TLB
DU	Debug Unit
EA	Effective address
FPU	Floating-Point Unit
GPR	General-Purpose Register
IC	Instruction Cache
IMMU	Instruction MMU
ITLB	Instruction TLB
MMU	Memory Management Unit
OR1K	OpenRISC 1000 Architecture
ORBIS	OpenRISC Basic Instruction Set
ORFPX	OpenRISC Floating-Point eXtension
ORVDX	OpenRISC Vector/DSP eXtension
PC	Program Counter
PCU	Performance Counters Unit
PIC	Programmable Interrupt Controller
PM	Power Management
PTE	Page Table Entry
R/W	Read/Write
RISC	Reduced Instruction Set Computer
SMP	Symmetrical Multi-Processing
SMT	Simultaneous Multi-Threading
SPR	Special-Purpose Register
SR	Supervision Register
TLB	Translation Lookaside Buffer

**Table 1. Acronyms and Abbreviations**

# 1 About this Manual

## 1.1 Introduction

The OpenRISC 1000 system architecture manual defines the architecture for a family of open-source, synthesizable RISC microprocessor cores. The OpenRISC 1000 architecture allows for a spectrum of chip and system implementations at a variety of price/performance points for a range of applications. It is a 32/64-bit load and store RISC architecture designed with emphasis on performance, simplicity, low power requirements, and scalability. The OpenRISC 1000 architecture targets medium and high performance networking and embedded computer environments.

This manual covers the instruction set, register set, cache management and coherency, memory model, exception model, addressing modes, operands conventions, and the application binary interface (ABI).

This manual does not specify implementation-specific details such as pipeline depth, cache organization, branch prediction, instruction timing, bus interface etc.

## 1.2 Authors

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 maintainer(s), and we'll correct the situation.

Name	E-mail	Contribution
Damjan Lampret	<a href="mailto:damjanl@opencores.org">damjanl@opencores.org</a>	Initial document
Chen-Min Chen	<a href="mailto:jimmy@ee.nctu.edu.tw">jimmy@ee.nctu.edu.tw</a>	Some notes
Marko Mlinar	<a href="mailto:markom@opencores.org">markom@opencores.org</a>	Fast context switches
Johan Rydberg	<a href="mailto:jrydberg@opencores.org">jrydberg@opencores.org</a>	ELF section
Matan Ziv-Av	<a href="mailto:matan@svgalib.org">matan@svgalib.org</a>	Several suggestions
Chris Ziomkowski	<a href="mailto:chris@opencores.org">chris@opencores.org</a>	Several suggestions
Greg McGary	<a href="mailto:greg@mcgary.org">greg@mcgary.org</a>	l.cmov, trap exception
Bob Gardner		Native Speaker Check
Rohit Mathur	<a href="mailto:rohitmathurs@opencores.org">rohitmathurs@opencores.org</a>	Technical review and corrections
Maria Bolado	<a href="mailto:mbolado@teisa.unican.es">mbolado@teisa.unican.es</a>	Technical review and corrections
ORSoC Yann Vernier	<a href="mailto:yannv@opencores.org">yannv@opencores.org</a>	Technical review and corrections
Julius Baxter	<a href="mailto:julius@opencores.org">julius@opencores.org</a>	Architecture revision information
Stefan Kristiansson	<a href="mailto:stefan.kristiansson@saunalahti.fi">stefan.kristiansson@saunalahti.fi</a>	Atomic instructions
Stefan Wallentowitz	<a href="mailto:stefan@wallentowitz.de">stefan@wallentowitz.de</a>	Multicore and corrections

Name	E-mail	Contribution
Stafford Horne	<a href="mailto:shorne@gmail.com">shorne@gmail.com</a>	Multicore and FPU Contributions
Andrey Bacherov	<a href="mailto:bandvig@gmail.com">bandvig@gmail.com</a>	FPU Contributions

**Table 1-1. Authors of this Manual**

## 1.3 Document Revision History

The revision history of this manual is presented in the table below.

Revision Date	By	Modifications	Arch. Ver (Maj.Min) – Doc Rev
15/Mar/2000	Damjan Lampret	Initial document	0.0-0
7/Apr/2001	Damjan Lampret	First public release	0.0-1
22/Apr/2001	Damjan Lampret	Incorporated changes from Johan and Matan	0.0-2
16/May/2001	Damjan Lampret	Changed SR, Debug, Exceptions, TT, PM. Added l.cmov, l.ff1, etc.	0.0-3
23/May/2001	Damjan Lampret	Added SR[SUMRA], configuration registerc etc.	0.0-4
24/May/2001	Damjan Lampret	Changed virtually almost all chapters in some way – major change is addition of configuration registers.	0.0-5
28/May/2001	Damjan Lampret	Changed addresses of some SPRs, removed group SPR group 11, added DCR[CT]=7.	0.0-6
24/Jan/2002	Marko Mlinar	Major check and update	0.0-7
9/Apr/2002	Marko Mlinar	PICPR register removed; l.sys convention added; mtspr/mfspr now use bitwise OR instead of sum	0.0-8
28/July/2002	Jeanne Wiegmann	First overall review & layout adjustment	0.0-9
20/Sep/2002	Rohit Mathur	Second overall review	0.0-10
12/Jan/2003	Damjan Lampret	Synchronization with or1ksim and OR1200 RTL. Not all chapters have been checked.	0.0-11
26/Jan/2003	Damjan Lampret	Synchronization with or1ksim and OR1200 RTL. From this revision on the manual carries revision number 1.0 and parts of the architecture that are implemented in OR1200 will no longer change because OR1200 is being implemented in silicon. Major parts that are not implemented in OR1200 and could change in the future include ORFPX, ORVDX, PCU, fast context switching, and 64-bit extension.	0.0-12

Revision Date	By	Modifications	Arch. Ver (Maj.Min) – Doc Rev
26/Jun/2004	Damjan Lampret	Fixed typos in instruction set description reported by Victor Lopez, Giles Hall and Luís Vitório Cargnini. Fixed typos in various chapters reported by Matjaz Breskvar. Changed description of PICSR. Updated ABI chapter based on agreed ABI from the openrisc mailing list. Removed DMR1[ETE], clearly defined watchpoints&breakpoint, split long watchpoint chain into two, removed WP10 and removed DMR1[DXFW], updated DMR2. Fixed FP definition (added FP exception. FPCSR register).	0.0-13
3/Nov/2005	Damjan Lampret	Corrected description of l.ff1, added l.fl1 instruction, corrected encoding of l.maci and added more description of tick timer.	0.0-14
15/Nov/2005	Damjan Lampret	Corrected description of l.sfXXui (arch manual had a wrong description compared to behavior implemented in or1ksim/gcc/or1200). Removed Atomicity chapter.	0.0-15
22/Mar/2011	ORSoC Yann Vernier	Converted to OpenDocument, ABI review, added instruction index and machine code reference table, added ORFPX and ORVDX headings, corrected descriptions for l.div, l.divu, l.ff1, l.fl1, l.mac*, l.mulu, l.msb, l.sub, lv.cmp_*.h, lv.muls.h, lv.pack.h, lv.subus.b, TLBTR, OF64S, specified link register for l.jal and l.jalr, PPN sizes, adjusted instruction classes, various typographical cleanups, clarified delay slot and exception interaction for l.j* and l.sys, removed empty 32-bit implementation for lv.pack/unpack to prevent blank pages	0.0-16
6/Aug/2011	Julius Baxter	Added architecture revision information.	0.0-17

Revision Date	By	Modifications	Arch. Ver (Maj.Min) – Doc Rev
05/Dec/2012	Julius Baxter	<p>Architecture version update</p> <p>Clarify unimplemented SPR space to be read as zero, writing to have no effect</p> <p>Clarify GPR0 implementation and use</p> <p>Remove l.trap instruction's conditional execution function</p> <p>Update ABI statement on returning structures by value</p> <p>Fix typo in register width description of l.sflw instruction</p> <p>Add UVRP bit in VR</p> <p>Add description of SPR VR2</p> <p>Add description of SPR AVR</p> <p>Add description of SPR EVBAR</p> <p>Mention implication of EVBAR in appropriate sections</p> <p>Add description of ISR SPRs</p> <p>Add presence bits for AVR, EVBAR, ISRs to CPUCFGR</p> <p>Add ND bit to CPUCFGR and mention optional delay slot in appropriate sections</p> <p>Mention exceptions possible for all branch/jump instructions</p> <p>Add description of SPRs AECR, AESR</p> <p>Add presence bits for AECR and AESR to CPUCFGR</p> <p>Clarify overflow exception behavior for appropriate unsigned and signed arithmetic instructions (l.add, l.addi, l.addc, l.addic, l.mul, l.muli, l.mulu, l.div, l.divu, l.sub, l.mac, l.maci, l.msb)</p> <p>Remove “signed” from name of addition and subtraction instructions, as they are used for both unsigned and signed arithmetic</p> <p>Add l.macu and l.msbl instructions for performing unsigned MAC operations</p> <p>Add l.muld and l.muldu for performing multiplication and allowing the 64-bit result to be accessible on 32-bit implementations</p>	1.0-0
21/Apr/2014	Stefan Kristiansson	<p>Add atomicity chapter.</p> <p>Add l.lwa and l.swa instructions.</p>	1.1-0
3/Mar/2015	Stefan Wallentowitz	<p>Corrections to multiple instruction encodings.</p>	1.2-0

Revision Date	By	Modifications	Arch. Ver (Maj.Min) – Doc Rev
19/Aug/2017	Stafford Horne	Add reservation of R10 for TLS. Add COREID and NUMCORES. Add atomic clarification on overlapping stores.	1.2-1
12/May/2019	Stafford Horne Andrey Bacherov	Add l.lf, l.adrp, lf.sfun*, lf.stod.d, lf.dtos.d instructions. Document ORFPX64A32 instructions. Clarifications on floating point. Assign addresses for FPMADD* and VMAC* SPRs.	1.3-1

Table 1-2. Revision History

## 1.4 Work in Progress

This document is *work in progress*. Anything in the manual could change until we have made our first silicon. The latest version is always available from revision control (Github as of this writing). See details about how to get it on [www.openrisc.io](http://www.openrisc.io).

We are currently looking for people to work on and maintain this document. If you would like to contribute, please send an email to one of the authors.

## 1.5 Fonts in this Manual

In this manual, fonts are used as follows:

- ✓ Typewriter font is used for programming examples.
- ✓ **Bold** font is used for emphasis.
- ✓ 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.

## 1.6 Conventions

<code>l.mnemonic</code>	Identifies an ORBIS32/64 instruction.
<code>lv.mnemonic</code>	Identifies an ORVDX32/64 instruction.
<code>lf.mnemonic</code>	Identifies an ORFPX32/64 instruction.
<code>0x</code>	Indicates a hexadecimal number.
<code>rA</code>	Instruction syntax used to identify a general purpose register
<code>REG[FIELD]</code>	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.
<code>X</code>	In certain contexts, this indicates a 'don't care'.
<code>N</code>	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 <code>SR[SM]=1</code> ).

**Table 1-3. Conventions**

## 1.7 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

This chapter introduces the OpenRISC 1000 architecture and describes the general architectural features.

### 2.1 Features

The OpenRISC 1000 architecture includes the following principal features:

- ✓ A completely free and open architecture.
- ✓ A linear, 32-bit or 64-bit logical address space with implementation-specific physical address space.
- ✓ Simple and uniform-length instruction formats featuring different instruction set extensions:
  - OpenRISC Basic Instruction Set (ORBIS32/64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 32- and 64-bit data
  - OpenRISC Vector/DSP eXtension (ORVFX64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 8-, 16-, 32- and 64-bit data
  - OpenRISC Floating-Point eXtension (ORFPX32/64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 32- and 64-bit data
- ✓ Two simple memory addressing modes, whereby memory address is calculated by:
  - addition of a register operand and a signed 16-bit immediate value
  - addition of a register operand and a signed 16-bit immediate value followed by update of the register operand with the calculated effective address
- ✓ Two register operands (or one register and a constant) for most instructions who then place the result in a third register
- ✓ Shadowed or single 32-entry or narrow 16-entry general purpose register file
- ✓ Optional branch delay slot for keeping the pipeline as full as possible
- ✓ Support for separate instruction and data caches/MMUs (Harvard architecture) or for unified instruction and data caches/MMUs (Stanford architecture)
- ✓ A flexible architecture definition that allows certain functions to be performed either in hardware or with the assistance of implementation-specific software
- ✓ Number of different, separated exceptions simplifying exception model
- ✓ Fast context switch support in register set, caches, and MMUs

## 2.2 Introduction

The OpenRISC 1000 architecture is a completely open architecture. It defines the architecture of a family of open source, RISC microprocessor cores. The OpenRISC 1000 architecture allows for a spectrum of chip and system implementations at a variety of price/performance points for a range of applications. It is a 32/64-bit load and store RISC architecture designed with emphasis on performance, simplicity, low power requirements, and scalability. OpenRISC 1000 targets medium and high performance networking and embedded computer environments.

Performance features include a full 32/64-bit architecture; vector, DSP and floating-point instructions; powerful virtual memory support; cache coherency; optional SMP and SMT support, and support for fast context switching. The architecture defines several features for networking and embedded computer environments. Most notable are several instruction extensions, a configurable number of general-purpose registers, configurable cache and TLB sizes, dynamic power management support, and space for user-provided instructions.

The OpenRISC 1000 architecture is the predecessor of a richer and more powerful next generation of OpenRISC architectures.

The full source for implementations of the OpenRISC 1000 architecture is available at [www.openrisc.io](http://www.openrisc.io) and [github.com/openrisc](https://github.com/openrisc) and is supported with GNU software development tools and a behavioral simulator. Most OpenRISC implementations are designed to be modular and vendor-independent. They can be interfaced with other open-source cores available at [www.opencores.org](http://www.opencores.org).

We encourage third parties to design and market their own implementations of the OpenRISC 1000 architecture and to participate in further development of the architecture.

## 2.3 Architecture Version Information

It is anticipated that revisions of the OR1K architecture will come about as architectural modifications are made over time. This document shall be valid for the latest version stated in it. Each implementation should indicate the minimum revision it supports in the Architecture Version Register (AVR).

The following table lists the versions and their release date.

Version	Date	Summary
0.0	November 2005	Initial architecture specification.
1.0	December 2012	First version.
1.1	April 2014	Atomic instructions additions.
1.2	April 2015	SPRs in user mode, multicore.
1.3	May 2019	New floating point instructions, l.adrp.

Table 2-1: Architecture Version Information

## 3 Addressing Modes and Operand Conventions

This chapter describes memory-addressing modes and memory operand conventions defined by the OpenRISC 1000 system architecture.

### 3.1 Memory Addressing Modes

The processor computes an effective address when executing a memory access instruction or branch instruction or when fetching the next sequential instruction. If the sum of the effective address and the operand length exceeds the maximum effective address in logical address space, the memory operand wraps around from the maximum effective address through effective address 0.

#### 3.1.1 Register Indirect with Displacement

Load/store instructions using this address mode contain a signed 16-bit immediate value, which is sign-extended and added to the contents of a general-purpose register specified in the instruction.

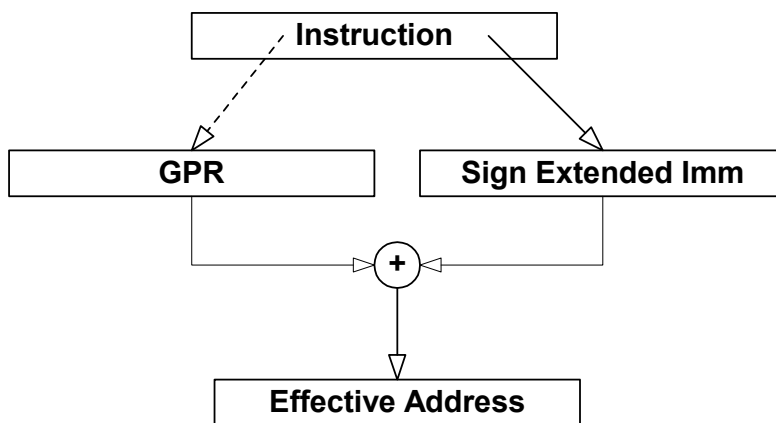


Figure 3-1. Register Indirect with Displacement Addressing

3-1 shows how an effective address is computed when using register indirect with displacement addressing mode.

### 3.1.2 PC Relative

Branch instructions using this address mode contain a signed 26-bit immediate value that is sign-extended and added to the contents of a Program Counter register. Before the execution at the destination PC, instruction in delay slot is executed if the ND bit in CPU Configuration Register (CPUCFGR) is set.

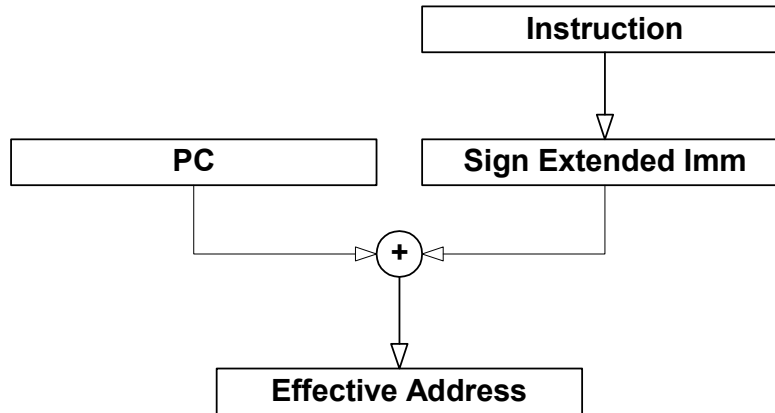


Figure 3-2. PC Relative Addressing

3-2 shows how an effective address is generated when using PC relative addressing mode.

## 3.2 Memory Operand Conventions

The architecture defines an 8-bit byte, 16-bit halfword, a 32-bit word, and a 64-bit doubleword. It also defines IEEE-754 compliant 32-bit single precision float and 64-bit double precision float storage units. 64-bit vectors of bytes, 64-bit vectors of halfwords, 64-bit vectors of singlewords, and 64-bit vectors of single precision floats are also defined.

Type of Data	Length in Bytes	Length in Bits
Byte	1	8
Halfword (or half)	2	16
Singleword (or word)	4	32
Doubleword (or double)	8	64
Single precision float	4	32
Double precision float	8	64
Vector of bytes	8	64
Vector of halfwords	8	64
Vector of singlewords	8	64

Type of Data	Length in Bytes	Length in Bits
Vector of single precision floats	8	64

**Table 3-1. Memory Operands and their sizes**

### 3.2.1 Bit and Byte Ordering

Byte ordering defines how the bytes that make up halfwords, singlewords and doublewords are ordered in memory. To simplify OpenRISC implementations, the architecture implements Most Significant Byte (MSB) ordering – or big endian byte ordering by default. But implementations can support Least Significant Byte (LSB) ordering if they implement byte reordering hardware. Reordering is enabled with bit SR[LEE].

The figures below illustrate the conventions for bit and byte numbering within various width storage units. These conventions hold for both integer and floating-point data, where the most significant byte of a floating-point value holds the sign and at least significant byte holds the start of the exponent.

3-2 shows how bits and bytes are ordered in a halfword.

Bit 15	Bit 8	Bit 7	Bit 0
MSB		LSB	
Byte address 0		Byte address 1	

**Table 3-2. Default Bit and Byte Ordering in Halfwords**

3-3 shows how bits and bytes are ordered in a singleword.

Bit 31	Bit 24	Bit 23	Bit 16	Bit 15	Bit 8	Bit 7	Bit 0
MSB						LSB	
Byte address 0		Byte address 1		Byte address 2		Byte address 3	

**Table 3-3. Default Bit and Byte Ordering in Singlewords and Single Precision Floats**

3-4 shows how bits and bytes are ordered in a doubleword.

Bit 63	Bit 56			
MSB				
Byte address 0	Byte address 1	Byte address 2	Byte address 3	
			Bit 7	Bit 0
			LSB	
Byte address 4	Byte address 5	Byte address 6	Byte address 7	

**Table 3-4. Default Bit and Byte Ordering in Doublewords, Double Precision Floats and all Vector Types**

### 3.2.2 Aligned and Misaligned Accesses

A memory operand is naturally aligned if its address is an integral multiple of the operand length. Implementations might support accessing unaligned memory operands, but the default behavior is that accesses to unaligned operands result in an alignment exception. See chapter Exception Model on page 272 for information on alignment exception.

Current OR32 implementations (OR1200) do not implement 8 byte alignment, but do require 4 byte alignment. Therefore the Application Binary Interface (chapter 17) uses 4 byte alignment for 8 byte types. Future extensions such as ORVFX64 may require natural alignment.

Operand	Length	addr[3:0] if aligned
Byte	8 bits	Xxxx
Halfword (or half)	2 bytes	Xxx0
Singleword (or word)	4 bytes	Xx00
Doubleword (or double)	8 bytes	X000
Single precision float	4 bytes	Xx00
Double precision float	8 bytes	X000
Vector of bytes	8 bytes	X000
Vector of halfwords	8 bytes	X000
Vector of singlewords	8 bytes	X000
Vector of single precision floats	8 bytes	X000

**Table 3-5. Memory Operand Alignment**

OR32 instructions are four bytes long and word-aligned.

## 4 Register Set

### 4.1 Features

The OpenRISC 1000 register set includes the following principal features:

- ✓ Thirty-two or sixteen 32/64-bit general-purpose registers – OpenRISC 1000 implementations optimized for use in FPGAs and ASICs in embedded and similar environments may implement only the first sixteen of the possible thirty-two registers.
- ✓ All other registers are special-purpose registers defined for each unit separately and accessible through the `l.mtspr`/`l.mfspr` instructions.

### 4.2 Overview

An OpenRISC 1000 processor includes several types of registers: user level general-purpose and special-purpose registers, supervisor level special-purpose registers and unit-dependent registers.

User level general-purpose and special-purpose registers are accessible both in user mode and supervisor mode of operation. Supervisor level special-purpose registers are accessible only in supervisor mode of operation (`SR[SM]=1`).

Unit dependent registers are usually only accessible in supervisor mode but there can be exceptions to this rule. Accessibility for architecture-defined units is defined in this manual. Accessibility for custom units not covered by this manual will be defined in the appropriate implementation-specific manuals.

### 4.3 Special-Purpose Registers

The special-purpose registers of all units are grouped into thirty-two groups. Each group can have different register address decoding depending on the maximum theoretical number of registers in that particular group. A group can contain registers from several different units or processes. The `SR[SM]` bit is also used in register address decoding, as some registers are accessible only in supervisor mode. The `l.mtspr` and `l.mfspr` instructions are used for reading and writing registers.

Unimplemented SPRs should read as zero. Writing to unimplemented SPRs will have no effect, and the `l.mtspr` instruction will effectively be a no-operation.

GROUP #	UNIT DESCRIPTION
0	System Control and Status registers
1	Data MMU (in the case of a single unified MMU, groups 1 and 2 decode into a single set of registers)
2	Instruction MMU (in the case of a single unified MMU, groups 1 and 2 decode into a single set of registers)
3	Data Cache (in the case of a single unified cache, groups 3 and 4 decode into a single set of registers)
4	Instruction Cache (in the case of a single unified cache, groups 3 and 4 decode into a single set of registers)
5	MAC unit
6	Debug unit
7	Performance counters unit
8	Power Management
9	Programmable Interrupt Controller
10	Tick Timer
11	Floating Point unit
12-23	Reserved for future use
24-31	Custom units

**Table 4-1. Groups of SPRs**

An OpenRISC 1000 processor implementation is required to implement at least the special purpose registers from group 0. All other groups are optional, and registers from these groups are implemented only if the implementation has the corresponding unit. Which units are actually implemented may be determined by reading the UPR register from group 0.

A 16-bit SPR address is made of 5-bit group index (bits 15-11) and 11-bit register index (bits 10-0).

Grp #	Reg #	Reg Name	USER MODE	SUPV MODE	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



Grp #	Reg #	Reg Name	USER MODE	SUPV MODE	Description
0	8	PCCFGR	—	R	Performance Counters Configuration register
0	9	VR2	—	R	Version register 2
0	10	AVR	—	R	Architecture version register
0	11	EVBAR	—	R/W	Exception vector base address register
0	12	AECR	—	R/W	Arithmetic Exception Control Register
0	13	AESR	—	R/W	Arithmetic Exception Status Register
0	16	NPC	—	R/W	PC mapped to SPR space (next PC)
0	17	SR	—	R/W	Supervision register
0	18	PPC	—	R	PC mapped to SPR space (previous PC)
0	20	FPCSR	R*	R/W	FP Control Status register
0	21-28	ISR0-ISR7		R	Implementation-specific registers
0	32-47	EPCR0-EPCR15	—	R/W	Exception PC registers
0	48-63	EEAR0-EEAR15	—	R/W	Exception EA registers
0	64-79	ESR0-ESR15	—	R/W	Exception SR registers
0	128	COREID	—	R	Core Identifier Register
0	129	NUMCORES	—	R	Number of Cores Register
0	1024-1535	GPR0-GPR511	—	R/W	GPRs mapped to SPR space
1	0	DMMUCR	—	R/W	Data MMU Control register
1	1	DMMUPR	—	R/W	Data MMU Protection Register
1	2	DTLBEIR	—	W	Data TLB Entry Invalidate register
1	4-7	DATBMR0-DATBMR3	—	R/W	Data ATB Match registers
1	8-11	DATBTR0-DATBTR3	—	R/W	Data ATB Translate registers
1	512-639	DTLBW0MR0-DTLBW0MR127	—	R/W	Data TLB Match registers Way 0
1	640-767	DTLBW0TR0-DTLBW0TR127	—	R/W	Data TLB Translate registers Way 0
1	768-895	DTLBW1MR0-DTLBW1MR127	—	R/W	Data TLB Match registers Way 1
1	896-1023	DTLBW1TR0-DTLBW1TR127	—	R/W	Data TLB Translate registers Way 1
1	1024-1151	DTLBW2MR0-DTLBW2MR127	—	R/W	Data TLB Match registers Way 2

Grp #	Reg #	Reg Name	USER MODE	SUPV MODE	Description
1	1152-1279	DTLBW2TR0-DTLBW2TR127	–	R/W	Data TLB Translate registers Way 2
1	1280-1407	DTLBW3MR0-DTLBW3MR127	–	R/W	Data TLB Match registers Way 3
1	1408-1535	DTLBW3TR0-DTLBW3TR127	–	R/W	Data TLB Translate registers Way 3
2	0	IMMUCR	–	R/W	Instruction MMU Control register
2	1	IMMUPR	–	R/W	Instruction MMU Protection Register
2	2	ITLBEIR	–	W	Instruction TLB Entry Invalidate register
2	4-7	IATBMR0-IATBMR3	–	R/W	Instruction ATB Match registers
2	8-11	IATBTR0-IATBTR3	–	R/W	Instruction ATB Translate registers
2	512-639	ITLBW0MR0-ITLBW0MR127	–	R/W	Instruction TLB Match registers Way 0
2	640-767	ITLBW0TR0-ITLBW0TR127	–	R/W	Instruction TLB Translate registers Way 0
2	768-895	ITLBW1MR0-ITLBW1MR127	–	R/W	Instruction TLB Match registers Way 1
2	896-1023	ITLBW1TR0-ITLBW1TR127	–	R/W	Instruction TLB Translate registers Way 1
2	1024-1151	ITLBW2MR0-ITLBW2MR127	–	R/W	Instruction TLB Match registers Way 2
2	1152-1279	ITLBW2TR0-ITLBW2TR127	–	R/W	Instruction TLB Translate registers Way 2
2	1280-1407	ITLBW3MR0-ITLBW3MR127	–	R/W	Instruction TLB Match registers Way 3
2	1408-1535	ITLBW3TR0-ITLBW3TR127	–	R/W	Instruction TLB Translate registers Way 3
3	0	DCCR	–	R/W	DC Control register
3	1	DCBPR	W	W	DC Block Prefetch register
3	2	DCBFR	W	W	DC Block Flush register
3	3	DCBIR	–	W	DC Block Invalidate register
3	4	DCBWR	W	W	DC Block Write-back register
3	5	DCBLR	W	W	DC Block Lock register
4	0	ICCR	–	R/W	IC Control register
4	1	ICBPR	W	W	IC Block Prefetch register
4	2	ICBIR	–	W	IC Block Invalidate register
4	3	ICBLR	W	W	IC Block Lock register
5	1	MACLO	R/W*	R/W*	MAC Low

Grp #	Reg #	Reg Name	USER MODE	SUPV MODE	Description
5	2	MACHI	R/W*	R/W*	MAC High
5	3	FPMADDLO	R/W*	R/W*	Floating Point MAC Low
5	4	FPMADDHI	R/W*	R/W*	Floating Point MAC High
5	5	VMACLO	R/W*	R/W*	Vector MAC Low
5	6	VMACHI	R/W*	R/W*	Vector MAC High
6	0-7	DVR0-DVR7	–	R/W	Debug Value registers
6	8-15	DCR0-DCR7	–	R/W	Debug Control registers
6	16	DMR1	–	R/W	Debug Mode register 1
6	17	DMR2	–	R/W	Debug Mode register 2
6	18-19	DCWR0-DCWR1	–	R/W	Debug Watchpoint Counter registers
6	20	DSR	–	R/W	Debug Stop register
6	21	DRR	–	R/W	Debug Reason register
7	0-7	PCCR0-PCCR7	R*	R/W	Performance Counters Count registers
7	8-15	PCMR0-PCMR7	–	R/W	Performance Counters Mode registers
8	0	PMR	–	R/W	Power Management register
9	0	PICMR	–	R/W	PIC Mask register
9	2	PICSR	–	R/W	PIC Status register
10	0	TTMR	–	R/W	Tick Timer Mode register
10	1	TTCR	R*	R/W	Tick Timer Count register

Table 4-2. List of All Special-Purpose Registers

SPRs with R\* for user mode access are readable in user mode if SR[SUMRA] is set.

The MACLO and MACHI registers are synchronized, such that any ongoing MAC operation finishes before they are read or written.

## 4.4 General-Purpose Registers (GPRs)

The thirty-two general-purpose registers are labeled R0-R31 and are 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit implementations. They hold scalar integer data, floating-point data, vectors or memory pointers. 4-3 contains a list of general-purpose registers. The GPRs may be accessed as both source and destination registers by ORBIS, ORVDX and ORFPX instructions.

See chapter Application Binary Interface on page 353 for information on floating-point data types. See also Register Usage on page 356, where r9 is defined as the Link Register.

Register					r31	r30
Register	r29	r28	r27	r26	r25	r24
Register	r23	r22	r21	r20	r19	r18
Register	r17	r16	r15	r14	r13	r12
Register	r11	r10	r9 LR	r8	r7	r6
Register	r5	r4	r3	r2	r1	r0

Table 4-3. General-Purpose Registers

R0 should always hold a zero value. It is the responsibility of software to initialize it. (This differs from architecture version 0 which commented on implementation and that it should never be used as a destination register – this is no longer specified.) Functions of other registers are explained in chapter Application Binary Interface on page 353.

An implementation may have several sets of GPRs and use them as shadow registers, switching between them whenever a new exception occurs. The current set is identified by the SR[CID] value.

An implementation is not required to initialize GPRs to zero during the reset procedure. The reset exception handler is responsible for initializing GPRs to zero if that is necessary.

## 4.5 Support for Custom Number of GPRs

Programs may be compiled with less than thirty-two registers. Unused registers are disabled (set as *fixed* registers) when compiling code. Such code is also executable on normal implementations with thirty-two registers but not vice versa. This feature is quite useful since users are expected to move from less powerful OpenRISC implementations with less than thirty-two registers to more powerful thirty-two register OpenRISC implementations.

If configuration registers are implemented, CPUCFGR[CGF] indicates whether implementation has complete thirty-two general-purpose registers or less than thirty-two registers. OR1200 has been implemented with 16 or 32 registers.

## 4.6 Supervision Register (SR)

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

The SR value defines the state of the processor.

<b>Bit</b>	31-28	27-17	16
<b>Identifier</b>	CID	Reserved	SUMRA
<b>Reset</b>	0	0	0
<b>R/W</b>	R/W	Read Only	R/W

<b>Bit</b>	15	14	13	12	11	10	9	8
<b>Identifier</b>	FO	EPH	DSX	OVE	OV	CY	F	CE
<b>Reset</b>	1	0	0	0	0	0	0	0
<b>R/W</b>	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W

<b>Bit</b>	7	6	5	4	3	2	1	0
<b>Identifier</b>	LEE	IME	DME	ICE	DCE	IEE	TEE	SM
<b>Reset</b>	0	0	0	0	0	0	0	1
<b>R/W</b>	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

SM	Supervisor Mode 0 Processor is in User Mode 1 Processor is in Supervisor Mode
TEE	Tick Timer Exception Enabled 0 Tick Timer Exceptions are not recognized 1 Tick Timer Exceptions are recognized
IEE	Interrupt Exception Enabled 0 Interrupts are not recognized 1 Interrupts are recognized
DCE	Data Cache Enable 0 Data Cache is not enabled 1 Data Cache is enabled
ICE	Instruction Cache Enable 0 Instruction Cache is not enabled 1 Instruction Cache is enabled
DME	Data MMU Enable 0 Data MMU is not enabled 1 Data MMU is enabled
IME	Instruction MMU Enable 0 Instruction MMU is not enabled 1 Instruction MMU is enabled
LEE	Little Endian Enable 0 Little Endian (LSB) byte ordering is not enabled 1 Little Endian (LSB) byte ordering is enabled
CE	CID Enable 0 CID disabled and shadow registers disabled 1 CID automatic increment and shadow registers enabled

F	<p>Flag</p> <p>0 Conditional branch flag was cleared by sfXX instructions</p> <p>1 Conditional branch flag was set by sfXX instructions</p>
CY	<p>Carry flag</p> <p>0 No carry out produced by last arithmetic operation</p> <p>1 Carry out was produced by last arithmetic operation</p>
OV	<p>Overflow flag</p> <p>0 No overflow occurred during last arithmetic operation</p> <p>1 Overflow occurred during last arithmetic operation</p>
OVE	<p>Overflow flag Exception</p> <p>0 Overflow flag does not cause an exception</p> <p>1 Overflow flag causes range exception</p>
DSX	<p>Delay Slot Exception</p> <p>0 EPCR points to instruction not in the delay slot</p> <p>1 EPCR points to instruction in delay slot</p>
EPH	<p>Exception Prefix High</p> <p>0 Exceptions vectors are located in memory area starting at 0x0</p> <p>1 Exception vectors are located in memory area starting at 0xF0000000</p>
FO	<p>Fixed One</p> <p>This bit is always set</p>
SUMRA	<p>SPRs User Mode Read Access</p> <p>0 All SPRs are inaccessible in user mode</p> <p>1 Certain SPRs can be read in user mode</p>
CID	<p>Context ID (<i>Fast Context Switching (Optional)</i>, page 275)</p> <p>0-15 Current Processor Context</p>

Table 4-4. SR Field Descriptions

## 4.7 Exception Program Counter Registers (EPCR0 - EPCR15)

The Exception Program Counter registers are special-purpose supervisor-level registers accessible with the l.mtspr/l.mfspr instructions in supervisor mode. Read access in user mode is possible if it is enabled in PCMRx[SUMRA]. They are 32-bit wide registers in 32-bit implementations and can be wider than 32 bits in 64-bit implementations.

After an exception, the EPCR is set to the program counter address (PC) of the instruction that was interrupted by the exception. If only one EPCR is present in the implementation (*Fast Context Switching (Optional)* disabled), it must be saved by the exception handler routine before exception recognition is re-enabled in the SR.

Bit	31-0
Identifier	EPC
Reset	0
R/W	R/W

EPC	Exception Program Counter Address
-----	-----------------------------------

Table 4-5. EPCR Field Descriptions

## 4.8 Exception Effective Address Registers (EEAR0-EEAR15)

The Exception Effective Address registers are special-purpose supervisor-level registers accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode. Read access in user mode is possible if it is enabled in `SR[SUMRA]`. The EEARs are 32-bit wide registers in 32-bit implementations and can be wider than 32 bits in 64-bit implementations.

After an exception, the EEAR is set to the effective address (EA) generated by the faulting instruction. If only one EEAR is present in the implementation, it must be saved by the exception handler routine before exception recognition is re-enabled in the SR.

Bit	31-0
Identifier	EEA
Reset	0
R/W	R/W

EEA	Exception Effective Address
-----	-----------------------------

Table 4-6. EEAR Field Descriptions

## 4.9 Exception Supervision Registers (ESR0-ESR15)

The Exception Supervision registers are special-purpose supervisor-level registers accessible with `l.mtspr/l.mfspr` instructions in supervisor mode. They are 32 bits wide registers in 32-bit implementations and can be wider than 32 bits in 64-bit implementations.

After an exception, the Supervision register (SR) is copied into the ESR. If only one ESR is present in the implementation, it must be saved by the exception handler routine before exception recognition is re-enabled in the SR.

Bit	31-0
Identifier	ESR
Reset	0
R/W	R/W

ESR	Exception SR
-----	--------------

Table 4-7. ESR Field Descriptions

## 4.10 Core Identification Registers (COREID and NUMCORES)

The Core Identification registers are special-purpose registers used in multicore platform configurations. They are 32 bit wide registers in 32-bit implementations and can be wider than 32 bits in 64-bit implementations.

The first core is indexed with 0.

## 4.11 Next and Previous Program Counter (NPC and PPC)

The Program Counter registers represent the address just executed and the address instruction just to be executed.

These and the GPR registers mapped into SPR space should only be used for debugging purposes by an external debugger. Applications should use the `l.jal` instruction to obtain the current program counter and arithmetic instructions to obtain GPR register values.

## 4.12 Floating Point Control Status Register (FPCSR)

Floating point control status register is a 32-bit special-purpose register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode and as read-only register in user mode if enabled in `SR[SUMRA]`.

The FPCSR value controls floating point rounding modes, optional generation of floating point exception and provides floating point status flags. Status flags are updated after every floating point instruction is completed and can serve to determine what caused the floating point exception.

If floating point exception is enabled then FPCSR status flags have to be cleared in floating point exception handler. Status flags are cleared by writing 0 to all status bits.

Bit	31-12	11	10	9	8
Identifier	Reserved	DZF	INF	IVF	IXF
Reset	0	0	0	0	0
R/W	Read Only	R/W	R/W	R/W	R/W

Bit	7	6	5	4	3	2-1	0
Identifier	ZF	QNF	SNF	UNF	OVF	RM	FPEE
Reset	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W



FPEE	Floating Point Exception Enabled 0 FP Exception is disabled 1 FP Exception is enabled
RM	Rounding Mode 0 Round to nearest 1 Round to zero 2 Round to infinity+ 3 Round to infinity-
OVF	Overflow Flag 0 No overflow 1 Result overflowed
UNF	Underflow Flag 0 No underflow 1 Result underflowed
SNF	SNAN Flag 0 Result not SNAN 1 Result SNAN
QNF	QNaN Flag 0 Result not QNaN 1 Result QNaN
ZF	Zero Flag 0 Result not zero 1 Result zero
IXF	IneXact Flag 0 Result precise 1 Result inexact
IVF	InValid Flag 0 Result valid 1 Result invalid
INF	INfinity Flag 0 Result finite 1 Result infinite
DZF	Divide by Zero Flag 0 Proper divide 1 Divide by zero

Table 4-8. FPCSR Field Descriptions

## 5 Instruction Set

This chapter describes the OpenRISC 1000 instruction set.

### 5.1 Features

The OpenRISC 1000 instruction set includes the following principal features:

- ✓ Simple and uniform-length instruction formats featuring five Instruction Subsets
- ✓ OpenRISC Basic Instruction Set (ORBIS32/64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 32-bit and 64-bit data
- ✓ OpenRISC Vector/DSP eXtension (ORVDX64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 8-, 16-, 32- and 64-bit data
- ✓ OpenRISC Floating-Point eXtension (ORFPX32/64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 32-bit and 64-bit data
- ✓ Reserved opcodes for custom instructions

Note: Instructions are divided into instruction classes. Only the basic classes are required to be implemented in an OpenRISC 1000 implementation.

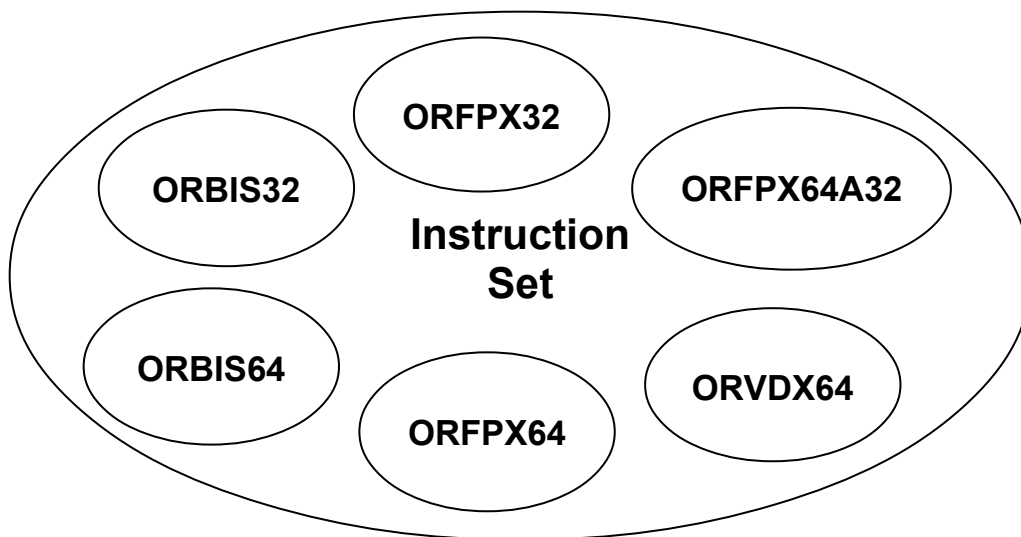


Figure 5-1. Instruction Set

### 5.2 Overview

OpenRISC 1000 instructions belong to one of the following instruction subsets:

- ✓ ORBIS32:
  - 32-bit integer instructions
  - Basic DSP instructions
  - 32-bit load and store instructions

- Program flow instructions
- Special instructions
- ✓ ORBIS64:
  - 64-bit integer instructions
  - 64-bit load and store instructions
- ✓ ORFPX32:
  - Single-precision floating-point instructions
- ✓ ORFPX64:
  - Double-precision floating-point instructions
  - 64-bit load and store instructions
- ✓ ORFPX64A32:
  - Double-precision floating-point instructions
  - Uses 32-bit general purpose register pairs for operations
- ✓ ORVDX64:
  - Vector instructions
  - DSP instructions

Instructions in each subset are also split into two instruction classes according to implementation importance:

- ✓ Class I
- ✓ Class II

Class	Description
I	Instructions in class I must always be implemented.
II	Instructions from class II are optional and an implementation may choose to use some or all instructions from this class based on requirements of the target application.

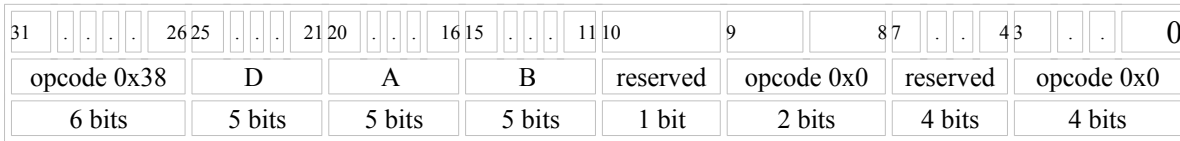
**Table 5-1. OpenRISC 1000 Instruction Classes**

## 5.3 ORBIS32/64

**l.add**

**Add**

**l.add**



### 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.

The instruction will set the carry flag on unsigned overflow, and the overflow flag on signed overflow.

### 32-bit Implementation:

```

rD[31:0] ← rA[31:0] + rB[31:0]
SR[CY]   ← carry (unsigned overflow)
SR[OV]   ← signed overflow

```

### 64-bit Implementation:

```

rD[63:0] ← rA[63:0] + rB[63:0]
SR[CY]   ← carry (unsigned overflow)
SR[OV]   ← signed overflow

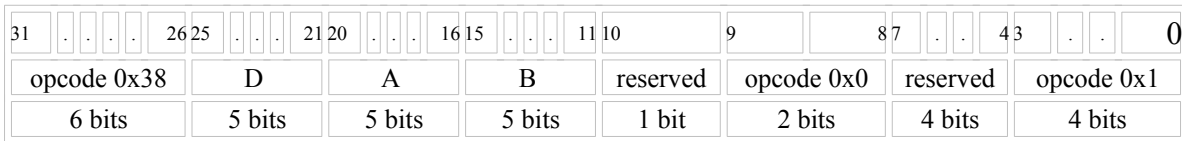
```

### Exceptions:

Range Exception on overflow if SR[OVE] and AECR[OVADDE] are set.

Range Exception on carry if SR[OVE] and AECR[CYADDE] are set.

Instruction Class  
ORBIS32 I

**l.addc****Add and Carry****l.addc****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.

The instruction will set the carry flag on unsigned overflow, and the overflow flag on signed overflow.

**32-bit Implementation:**

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

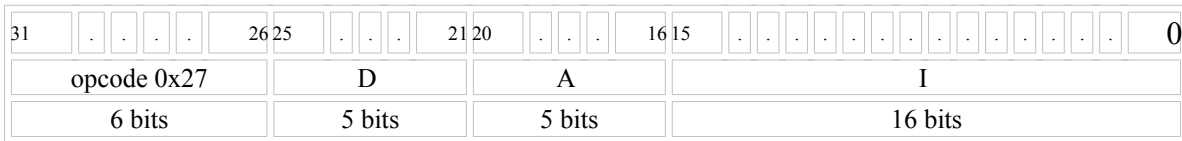
**64-bit Implementation:**

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

**Exceptions:**

Range Exception on overflow if SR[OVE] and AECR[OVADDE] are set.

Range Exception on carry if SR[OVE] and AECR[CYADDE] are set.

**l.addi****Add Immediate****l.addi****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.

The instruction will set the carry flag on unsigned overflow, and the overflow flag on signed overflow.

**32-bit Implementation:**

```

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

```

**64-bit Implementation:**

```

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

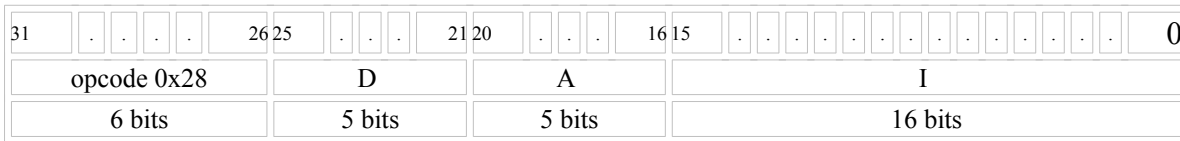
```

**Exceptions:**

Range Exception on overflow if SR[OVE] and AECR[OVADDE] are set.

Range Exception on carry if SR[OVE] and AECR[CYADDE] are set.

Instruction Class  
ORBIS32 I

**l.addic****Add Immediate and Carry****l.addic****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.

The instruction will set the carry flag on unsigned overflow, and the overflow flag on signed overflow.

**32-bit Implementation:**

```

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

```

**64-bit Implementation:**

```

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

```

**Exceptions:**

Range Exception on overflow if SR[OVE] and AECR[OVADDE] are set.

Range Exception on carry if SR[OVE] and AECR[CYADDE] are set.

**l.adrp      Compute PC-Relative Page Address      l.adrp**



### Format:

$$1.\text{adrp} \quad rD, I$$

**Description:**

The immediate value is shifted left 13 bits and sign extended to form a page offset. The page offset is added to the page address of the instruction to form the result. The result is placed into general-purpose register rD.

This can be used with a 13-bit page offset, computable at link time to create position independent code.

On 32-bit implementations the immediate is limited to 19 bits as the 13-bit shift left will truncate upper bits of the 21-bit immediate.

### 32-bit Implementation:

```

rD[31:0] ← exts(Immediate[18:0] << 13) + (InstAddr & -8192)

```

### 64-bit Implementation:

```

rD[63:0] ← exts(Immediate[20:0] << 13) + (InstAddr & -8192)

```

### Exceptions:

None



**l.and****And****l.and**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10			9			8	7	.	.	.	.	4	3	.	.			0
opcode 0x38					D					A					B					reserved					opcode 0x0					reserved					opcode 0x3							
6 bits					5 bits					5 bits					5 bits					1 bit					2 bits					4 bits					4 bits							

**Format:**

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

**Description:**

The contents of general-purpose register rA are combined with the contents of general-purpose register rB in a bit-wise logical AND operation. The result is placed into general-purpose register rD.

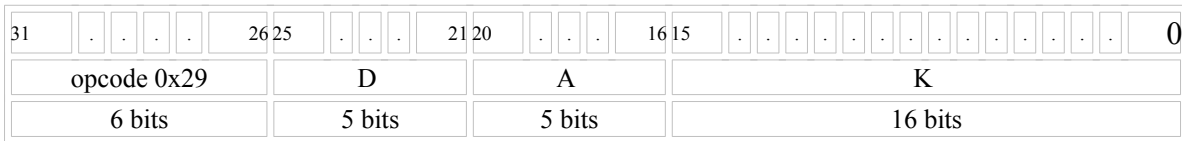
**32-bit Implementation:**

$$rD[31:0] \leftarrow rA[31:0] \text{ AND } rB[31:0]$$
**64-bit Implementation:**

$$rD[63:0] \leftarrow rA[63:0] \text{ AND } rB[63:0]$$
**Exceptions:**

None

# l.andi      And with Immediate Half Word      l.andi



## Format:

```
l.andi 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 AND operation. The result is placed into general-purpose register rD.

## 32-bit Implementation:

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

## 64-bit Implementation:

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

## Exceptions:

None

**l.bf**

## Branch if Flag

**l.bf**

**Format:**

1.bf N

### Description:

The immediate value is shifted left two bits, sign-extended to program counter width, and then added to the address of the branch instruction. The result is the effective address of the branch. If the flag is set, the program branches to EA. If CPUFCGR[ND] is not set, the branch occurs with a delay of one instruction.

### 32-bit Implementation:

```
EA ← exts(Immediate << 2) + BranchInsnAddr
PC ← EA if SR[F] set
```

### 64-bit Implementation:

```
EA ← exts(Immediate << 2) + BranchInsnAddr
PC ← EA if SR[F] set
```

## Exceptions:

None

**l.bnf**

## Branch if No Flag

**l.bnf**

**Format:**

1.bnf N

### Description:

The immediate value is shifted left two bits, sign-extended to program counter width, and then added to the address of the branch instruction. The result is the effective address of the branch. If the flag is cleared, the program branches to EA. If CPUCFGR[ND] is not set, the branch occurs with a delay of one instruction.

### 32-bit Implementation:

```
EA ← exts(Immediate << 2) + BranchInsnAddr
```

PC  $\leftarrow$  EA if SR[F] cleared

### 64-bit Implementation:

$$EA \leftarrow \text{exts}(\text{Immediate} \ll 2) + \text{BranchInsnAddr}$$

PC  $\leftarrow$  EA if SR[F] cleared

## Exceptions:

None

**l.cmov****Conditional Move****l.cmov**

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

**Format:**

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

**Description:**

If SR[F] is set, general-purpose register rA is placed in general-purpose register rD. If SR[F] is cleared, general-purpose register rB is placed in general-purpose register rD.

**32-bit Implementation:**

$$rD[31:0] \leftarrow SR[F] ? rA[31:0] : rB[31:0]$$
**64-bit Implementation:**

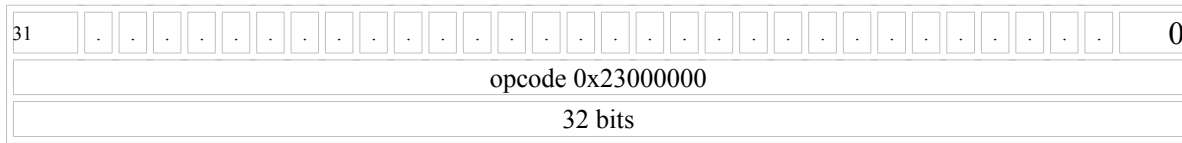
$$rD[63:0] \leftarrow SR[F] ? rA[63:0] : rB[63:0]$$
**Exceptions:**

None

## l.sync

## Context Synchronization

## l.sync



**Format:**

```
l.csync
```

### Description:

Execution of context synchronization instruction results in completion of all operations inside the processor and a flush of the instruction pipelines. When all operations are complete, the RISC core resumes with an empty instruction pipeline and fresh context in all units (MMU for example).

### 32-bit Implementation:

context-synchronization

### 64-bit Implementation:

context-synchronization

## Exceptions:

None

**l.cust1      Reserved for ORBIS32/64 Custom      l.cust1**  
**Instructions**



### Format:

1.cust1

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but rather by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

N/A

## Exceptions:

N/A

## l.cust2

## Reserved for ORBIS32/64 Custom Instructions

## l.cust2

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but rather by the implementation itself.

## N/A

## N/A

## N/A



## Reserved for ORBIS32/64 Custom Instructions

**l.cust3** **l.cust3**



### Format:

l.cust3

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but rather by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

N/A

### Exceptions:

N/A

**l.cust4      Reserved for ORBIS32/64 Custom      l.cust4**  
**Instructions**



**Format:**

1.cust4

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but rather by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

N/A

## Exceptions:

N/A

## Reserved for ORBIS32/64 Custom Instructions

**l.cust5** **l.cust5**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	5	4	.	.	.	0
opcode 0x3c						D					A					B					L						K				
6 bits						5 bits					5 bits					5 bits					6 bits						5 bits				

### Format:

`l.cust5 rD, rA, rB, L, K`

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but rather by the implementation itself.

### 32-bit Implementation:

N/A

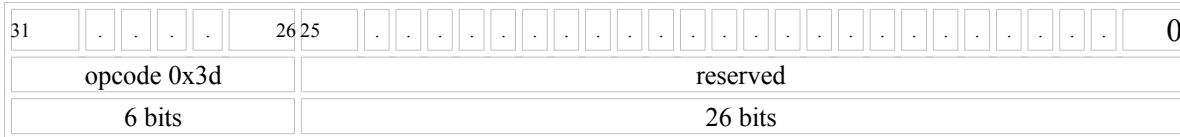
### 64-bit Implementation:

N/A

### Exceptions:

N/A

**l.cust6      Reserved for ORBIS32/64 Custom      l.cust6**  
**Instructions**



### Format:

1.cust6

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but rather by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

N/A

## Exceptions:

N/A

**l.cust7**[illegible]

### Format:

1.cust7

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but rather by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

N/A

## Exceptions:

N/A

**l.cust8      Reserved for ORBIS32/64 Custom      l.cust8**  
**Instructions**



### Format:

1.cust8

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but rather by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

N/A

## Exceptions:

N/A

**l.div****Divide Signed****l.div**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10			9				8	7	.	.	.	.	4	3	.	.	.	.	0
opcode 0x38					D					A					B					reserved					opcode 0x3					reserved					opcode 0x9								
6 bits					5 bits					5 bits					5 bits					1 bit					2 bits					4 bits					4 bits								

**Format:**

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

**Description:**

The content of general-purpose register rA are divided by the content of general-purpose register rB, and the result is placed into general-purpose register rD. Both operands are treated as signed integers.

On divide-by zero, rD will be undefined, and the overflow flag will be set. Note that prior revisions of the manual (pre-2011) stored the divide by zero flag in SR[CY].

**32-bit Implementation:**

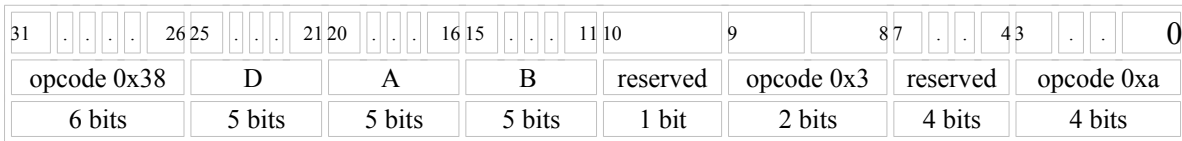
```
rD[31:0] ← rA[31:0] / rB[31:0]
SR[OV]   ← rB[31:0] == 0
```

**64-bit Implementation:**

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

**Exceptions:**

Range Exception when divisor is zero if SR[OVE] and AECR[DBZE] are set.

**l.divu****Divide Unsigned****l.divu****Format:**

```
l.divu rD, rA, rB
```

**Description:**

The content of general-purpose register rA are divided by the content of general-purpose register rB, and the result is placed into general-purpose register rD. Both operands are treated as unsigned integers.

On divide-by zero, rD will be undefined, and the overflow flag will be set.

**32-bit Implementation:**

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

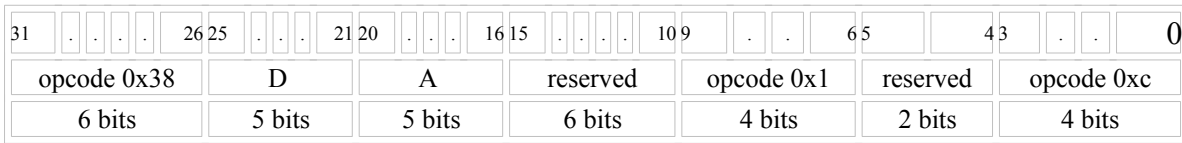
**64-bit Implementation:**

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

**Exceptions:**

Range Exception when divisor is zero if SR[OVE] and AECR[DBZE] are set.



**l.extbs****Extend Byte with Sign****l.extbs****Format:**

```
l.extbs rD, rA
```

**Description:**

Bit 7 of general-purpose register rA is placed in high-order bits of general-purpose register rD. The low-order eight bits of general-purpose register rA are copied into the low-order eight bits of general-purpose register rD.

**32-bit Implementation:**

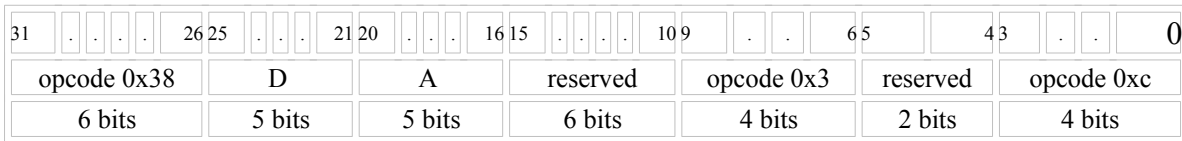
```
rD[31:8] ← rA[7]
rD[7:0]  ← rA[7:0]
```

**64-bit Implementation:**

```
rD[63:8] ← rA[7]
rD[7:0]  ← rA[7:0]
```

**Exceptions:**

None

**l.extbz****Extend Byte with Zero****l.extbz****Format:**

```
l.extbz rD, rA
```

**Description:**

Zero is placed in high-order bits of general-purpose register rD. The low-order eight bits of general-purpose register rA are copied into the low-order eight bits of general-purpose register rD.

**32-bit Implementation:**

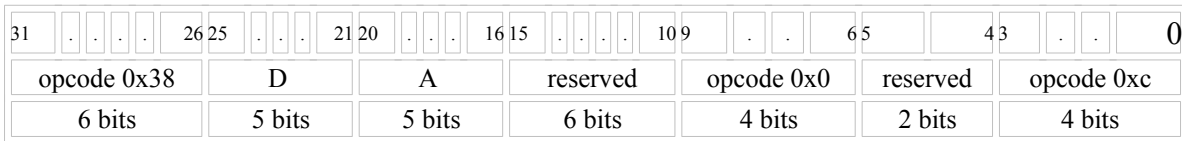
```
rD[31:8] ← 0
rD[7:0]  ← rA[7:0]
```

**64-bit Implementation:**

```
rD[63:8] ← 0
rD[7:0]  ← rA[7:0]
```

**Exceptions:**

None

**l.exths****Extend Half Word with Sign****l.exths****Format:**

```
l.exths rD, rA
```

**Description:**

Bit 15 of general-purpose register rA is placed in high-order bits of general-purpose register rD. The low-order 16 bits of general-purpose register rA are copied into the low-order 16 bits of general-purpose register rD.

**32-bit Implementation:**

```
rD[31:16] ← rA[15]
rD[15:0]  ← rA[15:0]
```

**64-bit Implementation:**

```
rD[63:16] ← rA[15]
rD[15:0]  ← rA[15:0]
```

**Exceptions:**

None

**l.exthz****Extend Half Word with Zero****l.exthz**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	.	10	9	.	.	6	5	.	.	4	3	.	.	0
opcode 0x38					D					A					reserved					opcode 0x2				reserved			opcode 0xc						
6 bits					5 bits					5 bits					6 bits					4 bits				2 bits			4 bits						

**Format:**

```
l.exthz rD, rA
```

**Description:**

Zero is placed in high-order bits of general-purpose register rD. The low-order 16 bits of general-purpose register rA are copied into the low-order 16 bits of general-purpose register rD.

**32-bit Implementation:**

```
rD[31:16] ← 0
rD[15:0]  ← rA[15:0]
```

**64-bit Implementation:**

```
rD[63:16] ← 0
rD[15:0]  ← rA[15:0]
```

**Exceptions:**

None

**l.extws****Extend Word with Sign****l.extws**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	10	9	.	.	6	5	.	.	4	3	.	.	0
opcode 0x38						D					A					reserved						opcode 0x0				reserved			opcode 0xd			
6 bits						5 bits					5 bits					6 bits						4 bits				2 bits			4 bits			

**Format:**

```
l.extws rD, rA
```

**Description:**

Bit 31 of general-purpose register rA is placed in high-order bits of general-purpose register rD. The low-order 32 bits of general-purpose register rA are copied from low-order 32 bits of general-purpose register rD.

**32-bit Implementation:**

```
rD[31:0] ← rA[31:0]
```

**64-bit Implementation:**

```
rD[63:32] ← rA[31]
rD[31:0] ← rA[31:0]
```

**Exceptions:**

None

**l.extwz****Extend Word with Zero****l.extwz**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	10	9	.	.	6	5	.	.	4	3	.	.	0
opcode 0x38					D					A					reserved					opcode 0x1				reserved				opcode 0xd				
6 bits					5 bits					5 bits					6 bits					4 bits				2 bits				4 bits				

**Format:**

```
l.extwz rD, rA
```

**Description:**

Zero is placed in high-order bits of general-purpose register rD. The low-order 32 bits of general-purpose register rA are copied into the low-order 32 bits of general-purpose register rD.

**32-bit Implementation:**

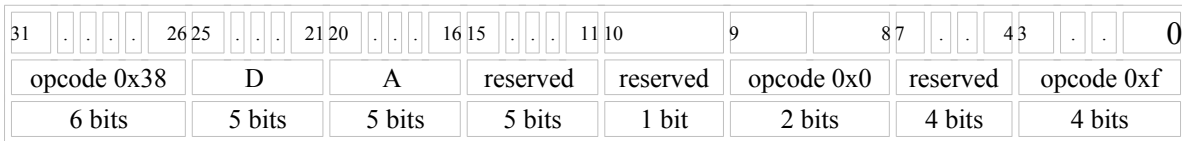
```
rD[31:0] ← rA[31:0]
```

**64-bit Implementation:**

```
rD[63:32] ← 0
rD[31:0] ← rA[31:0]
```

**Exceptions:**

None

**l.ffl****Find First 1****l.ffl****Format:**

```
l.ffl rD, rA
```

**Description:**

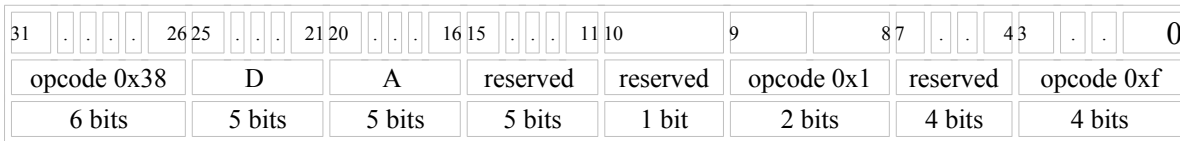
Position of the lowest order '1' bit is written into general-purpose register rD. Checking for bit '1' starts with bit 0 (LSB), and counting is incremented for every zero bit. If first '1' bit is discovered in LSB, one is written into rD, if first '1' bit is discovered in MSB, 32 (64) is written into rD. If there is no '1' bit, zero is written in rD.

**32-bit Implementation:**

$$rD[31:0] \leftarrow rA[0] ? 1 : rA[1] ? 2 \dots rA[31] ? 32 : 0$$
**64-bit Implementation:**

$$rD[63:0] \leftarrow rA[0] ? 1 : rA[1] ? 2 \dots rA[63] ? 64 : 0$$
**Exceptions:**

None

**l.fl1****Find Last 1****l.fl1****Format:**

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

**Description:**

Position of the highest order '1' bit is written into general-purpose register rD. Checking for bit '1' starts with bit 31/63 (MSB), and counting is decremented for every zero bit until the last '1' bit is found nearing the LSB. If highest order '1' bit is discovered in MSB, 32 (64) is written into rD, if highest order '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:**

$$rD[31:0] \leftarrow rA[31] ? 32 : rA[30] ? 31 \dots rA[0] ? 1 : 0$$
**64-bit Implementation:**

$$rD[63:0] \leftarrow rA[63] ? 64 : rA[62] ? 63 \dots rA[0] ? 1 : 0$$
**Exceptions:**

None



## 1.j



1. j N

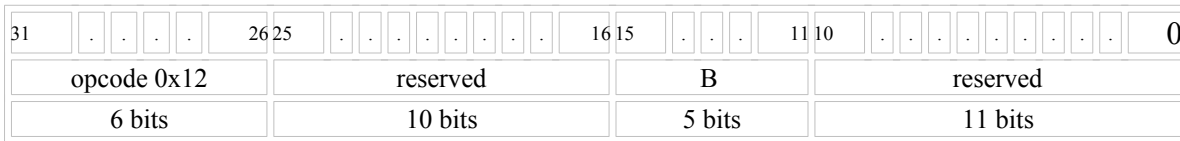
Note that l.sys should not be placed in the delay slot after a jump.

$$PC \leftarrow \text{exts}(\text{Immediate} \ll 2) + \text{JumpInsnAddr}$$
$$PC \leftarrow \text{exts}(\text{Immediate} \ll 2) + \text{JumpInsnAddr}$$

- TLB miss
- Page fault
- Bus error

**l.jal**



**l.jalr****Jump and Link Register****l.jalr****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. If CPUCFGR[ND] is not set, the jump occurs 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 link register r9 (see Register Usage on page 356) as rB. This is because an exception in the delay slot (including external interrupts) may cause l.jalr to be reexecuted.

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.

Note that l.sys should not be placed in the delay slot after a jump.

**32-bit Implementation:**

$$PC \leftarrow rB$$

$$LR \leftarrow CPUCFGR[ND] ? \text{JumpInsnAddr} + 4 : \text{DelayInsnAddr} + 4$$
**64-bit Implementation:**

$$PC \leftarrow rB$$

$$LR \leftarrow CPUCFGR[ND] ? \text{JumpInsnAddr} + 4 : \text{DelayInsnAddr} + 4$$
**Exceptions:**

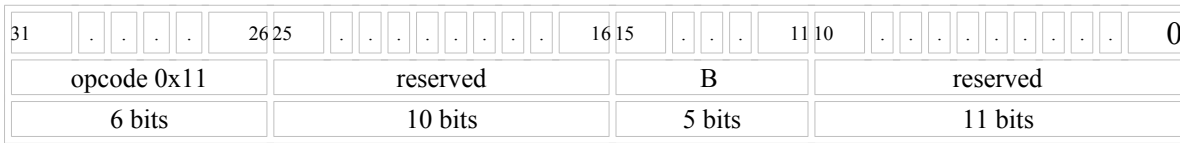
Alignment

TLB miss

Page fault

Bus error

Instruction Class  
ORBIS32 I

**l.jr****Jump Register****l.jr****Format:**

l.jr rB

**Description:**

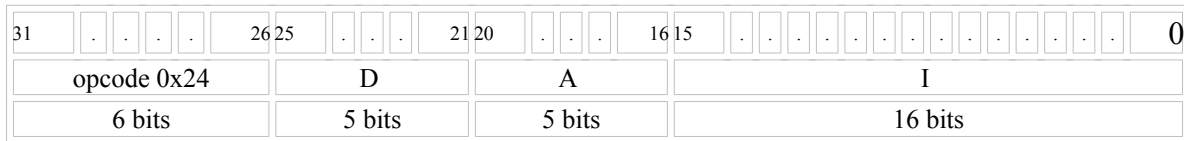
The contents of general-purpose register rB is the effective address of the jump. The program unconditionally jumps to EA. If CPUCFGR[ND] is not set, the jump occurs with a delay of one instruction.

Note that l.sys should not be placed in the delay slot after a jump.

**32-bit Implementation:**PC  $\leftarrow$  rB**64-bit Implementation:**PC  $\leftarrow$  rB**Exceptions:**

Alignment  
TLB miss  
Page fault  
Bus error

<b>1.lbs</b>	<b>Load Byte and Extend with Sign</b>	<b>1.lbs</b>
--------------	---------------------------------------	--------------



### Format:

1.lbs rD, I (rA)

### Description:

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The byte in memory addressed by EA is loaded into the low-order eight bits of general-purpose register rD. High-order bits of general-purpose register rD are replaced with bit 7 of the loaded value.

### 32-bit Implementation:

```
EA      ← exts(Immediate) + rA[31:0]
rD[7:0] ← (EA)[7:0]
rD[31:8] ← (EA)[7]
```

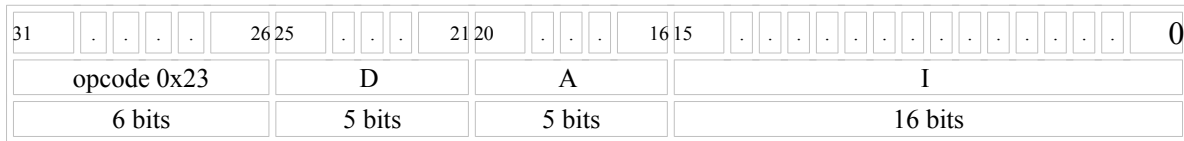
### 64-bit Implementation:

```
EA      ← exts(Immediate) + rA[63:0]
rD[7:0] ← (EA)[7:0]
rD[63:8] ← (EA)[7]
```

## Exceptions:

- TLB miss
- Page fault
- Bus error

<b>l.lbz</b>	<b>Load Byte and Extend with Zero</b>	<b>l.lbz</b>
--------------	---------------------------------------	--------------



**Format:**

1.lbz rD, I (rA)

### Description:

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The byte in memory addressed by EA is loaded into the low-order eight bits of general-purpose register rD. High-order bits of general-purpose register rD are replaced with zero.

### 32-bit Implementation:

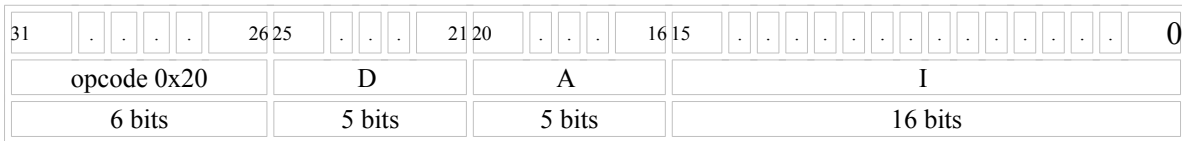
```
EA      ← exts(Immediate) + rA[31:0]
rD[7:0] ← (EA)[7:0]
rD[31:8] ← 0
```

### 64-bit Implementation:

```
EA      ← exts(Immediate) + rA[63:0]
rD[7:0] ← (EA)[7:0]
rD[63:8] ← 0
```

## Exceptions:

- TLB miss
- Page fault
- Bus error

**l.ld****Load Double Word****l.ld****Format:**

```
l.ld rD,I(rA)
```

**Description:**

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The double word in memory addressed by EA is loaded into general-purpose register rD.

**32-bit Implementation:**

N/A

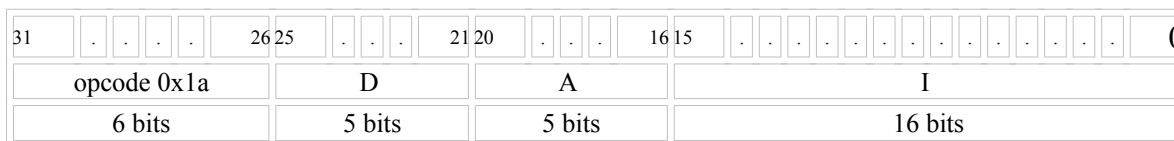
**64-bit Implementation:**

```
EA      ← exts(Immediate) + rA[63:0]
rD[63:0] ← (EA)[63:0]
```

**Exceptions:**

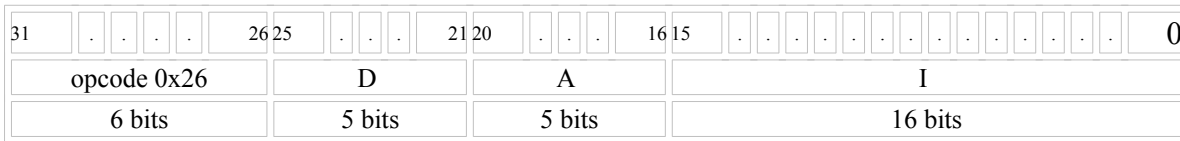
TLB miss  
 Page fault  
 Bus error  
 Alignment

**1.1f**





# l.lhs      Load Half Word and Extend with Sign      l.lhs



## Format:

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

## Description:

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The half word in memory addressed by EA is loaded into the low-order 16 bits of general-purpose register rD. High-order bits of general-purpose register rD are replaced with bit 15 of the loaded value.

## 32-bit Implementation:

```
EA      ← exts(Immediate) + rA[31:0]
rD[15:0] ← (EA)[15:0]
rD[31:16] ← (EA)[15]
```

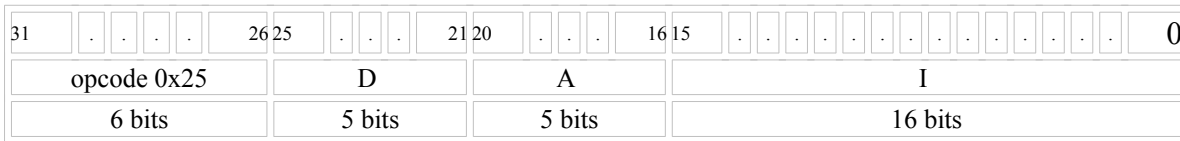
## 64-bit Implementation:

```
EA      ← exts(Immediate) + rA[63:0]
rD[15:0] ← (EA)[15:0]
rD[63:16] ← (EA)[15]
```

## Exceptions:

```
TLB miss
Page fault
Bus error
Alignment
```

## l.lhz      Load Half Word and Extend with Zero      l.lhz



### Format:

`l.lhz rD, I(rA)`

### Description:

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The half word in memory addressed by EA is loaded into the low-order 16 bits of general-purpose register rD. High-order bits of general-purpose register rD are replaced with zero.

### 32-bit Implementation:

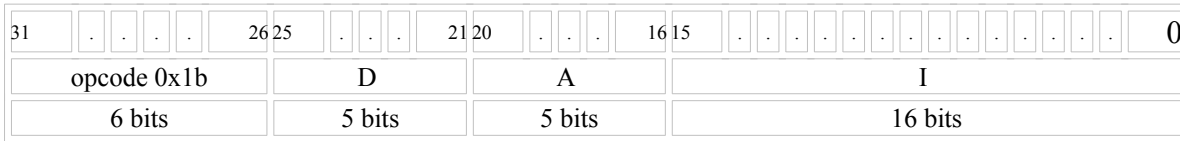
```
EA      ← exts(Immediate) + rA[31:0]
rD[15:0] ← (EA)[15:0]
rD[31:16] ← 0
```

### 64-bit Implementation:

```
EA      ← exts(Immediate) + rA[63:0]
rD[15:0] ← (EA)[15:0]
rD[63:16] ← 0
```

### Exceptions:

TLB miss  
Page fault  
Bus error  
Alignment

**l.lwa****Load Single Word Atomic****l.lwa****Format:**

```
l.lwa rD, I(rA)
```

**Description:**

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The single word in memory addressed by EA is loaded into the low-order 32 bits of general-purpose register rD. High-order bits of general-purpose register rD are replaced with zero.

An atomic reservation is placed on the address formed from EA. In case an MMU is enabled, the physical translation of EA is used.

**32-bit Implementation:**

```
EA          ← exts(Immediate) + rA[31:0]
rD[31:0]    ← (EA)[31:0]
atomic_reserve[to_phys(EA)] ← 1
```

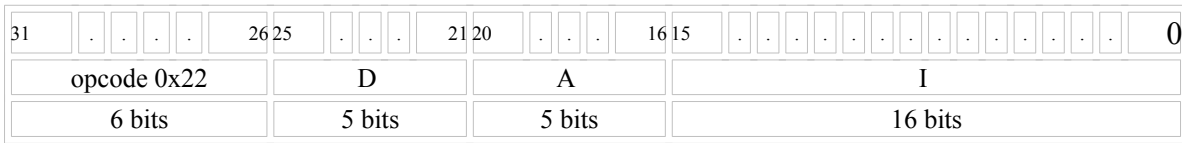
**64-bit Implementation:**

```
EA          ← exts(Immediate) + rA[63:0]
rD[31:0]    ← (EA)[31:0]
rD[63:32]   ← 0
atomic_reserve[to_phys(EA)] ← 1
```

**Exceptions:**

```
TLB miss
Page fault
Bus error
Alignment
```

## l.lws    Load Single Word and Extend with Sign    l.lws



### Format:

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

### Description:

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The single word in memory addressed by EA is loaded into the low-order 32 bits of general-purpose register rD. High-order bits of general-purpose register rD are replaced with bit 31 of the loaded value.

### 32-bit Implementation:

```
EA      ← exts(Immediate) + rA[31:0]
rD[31:0] ← (EA) [31:0]
```

### 64-bit Implementation:

```
EA      ← exts(Immediate) + rA[63:0]
rD[31:0] ← (EA) [31:0]
rD[63:32] ← (EA) [31]
```

### Exceptions:

```
TLB miss
Page fault
Bus error
Alignment
```

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0
opcode 0x21					D					A					I																
6 bits					5 bits					5 bits					16 bits																

1.lwz rD, I (rA)

1.lwz rD, I (rA)

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The single word in memory addressed by EA is loaded into the low-order 32 bits of general-purpose register rD. High-order bits of general-purpose register rD are replaced with zero.

```
EA      ← exts(Immediate) + rA[31:0]
rD[31:0] ← (EA)[31:0]
```

```
EA      ← exts(Immediate) + rA[63:0]
rD[31:0] ← (EA)[31:0]
rD[63:32] ← 0
```

- TLB miss
- Page fault
- Bus error
- Alignment

# l.mac Multiply and Accumulate Signed l.mac

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	4	3	.	.	0
opcode 0x31					reserved					A					B					reserved					opcode 0x1					
6 bits					5 bits					5 bits					5 bits					7 bits					4 bits					

## Format:

l.mac rA, rB

## Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are multiplied, and the 64 bit result is added to the special-purpose registers MACHI and MACLO. All operands are treated as signed integers.

The instruction will set the overflow flag if signed overflow is detecting during the addition stage.

## 32-bit Implementation:

$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{MACHI}[31:0]\text{MACLO}[31:0] + \text{rA}[31:0] * \text{rB}[31:0]$$

$$\text{SR}[\text{OV}] \leftarrow \text{signed overflow during addition stage}$$

## 64-bit Implementation:

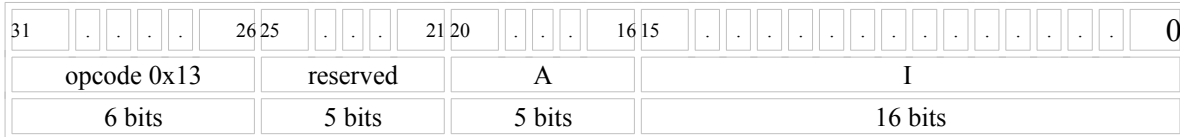
$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{MACHI}[31:0]\text{MACLO}[31:0] + \text{rA}[63:0] * \text{rB}[63:0]$$

$$\text{SR}[\text{OV}] \leftarrow \text{signed overflow during addition stage}$$

## Exceptions:

Range Exception on signed overflow if SR[OVE] and AECR[OVMACADDE] are set.

# l.maci **Multiply Immediate and Accumulate Signed** l.maci



## Format:

l.maci rA, I

## Description:

The immediate value and the contents of general-purpose register rA are multiplied, and the 64 bit result is added to the special-purpose registers MACHI and MACLO. All operands are treated as signed integers.

The instruction will set the overflow flag if signed overflow is detecting during the addition stage.

## 32-bit Implementation:

$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{MACHI}[31:0]\text{MACLO}[31:0] + \text{rA}[31:0] * \text{exts}(\text{Immediate})$$

$$\text{SR}[\text{OV}] \leftarrow \text{signed overflow during addition stage}$$

## 64-bit Implementation:

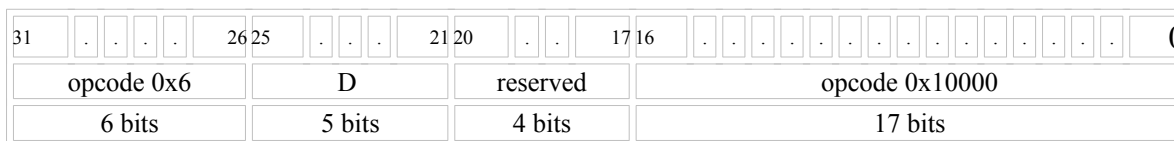
$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{MACHI}[31:0]\text{MACLO}[31:0] + \text{rA}[63:0] * \text{exts}(\text{Immediate})$$

$$\text{SR}[\text{OV}] \leftarrow \text{signed overflow during addition stage}$$

## Exceptions:

Range Exception on signed overflow if SR[OVE] and AECR[OVMACADDE] are set.

## l.macrc



```
l.macrc rD
```

The MAC pipeline also synchronizes with the instruction pipeline on any access to MACLO or MACHI SPRs, so that `l.mfspr` can be used to read MACHI before executing `l.macrc`.

```
synchronize-mac
rD[31:0] ← MACLO[31:0]
MACLO[31:0], MACHI[31:0] ← 0
```

```
synchronize-mac
rD[63:0] ← MACHI[31:0]MACLO[31:0]
MACLO[31:0], MACHI[31:0] ← 0
```

None



# l.macu Multiply and Accumulate Unsigned l.macu

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	4	3	.	.	0
opcode 0x31					reserved					A					B					reserved					opcode 0x3					
6 bits					5 bits					5 bits					5 bits					7 bits					4 bits					

## Format:

l.macu rA, rB

## Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are multiplied, and the 64 bit result is added to the special-purpose registers MACHI and MACLO. All operands are treated as unsigned integers.

The instruction will set the overflow flag if unsigned overflow is detecting during the addition stage.

## 32-bit Implementation:

$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{MACHI}[31:0]\text{MACLO}[31:0] + \text{rA}[31:0] * \text{rB}[31:0]$$

$$\text{SR}[\text{CY}] \leftarrow \text{unsigned overflow during addition stage}$$

## 64-bit Implementation:

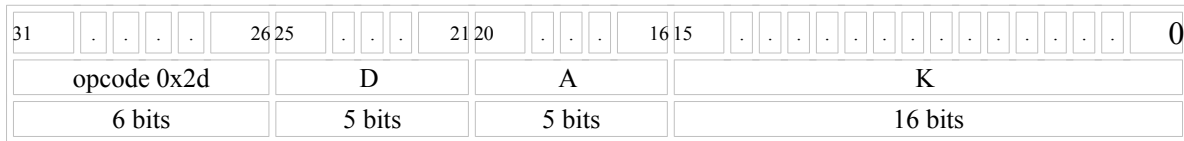
$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{MACHI}[31:0]\text{MACLO}[31:0] + \text{rA}[63:0] * \text{rB}[63:0]$$

$$\text{SR}[\text{CY}] \leftarrow \text{unsigned overflow during addition stage}$$

## Exceptions:

Range Exception on unsigned overflow if SR[OVE] and AECR[CYMACADDE] are set.

## l.mfspr Move From Special-Purpose Register l.mfspr



**Format:**

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

### Description:

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

### 32-bit Implementation:

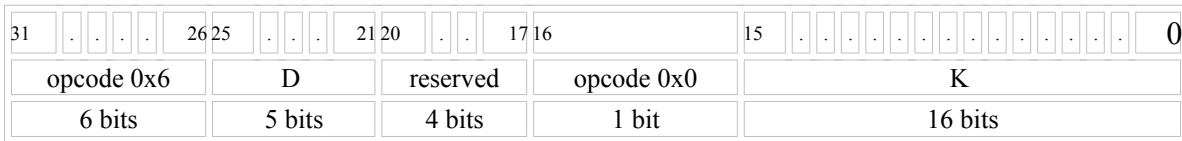
```
rD[31:0] ← spr(rA OR Immediate)
```

### 64-bit Implementation:

```
rD[63:0] ← spr(rA OR Immediate)
```

## Exceptions:

None

**l.movhi****Move Immediate High****l.movhi****Format:**

```
l.movhi rD, K
```

**Description:**

The 16-bit immediate value is zero-extended, shifted left by 16 bits, and placed into general-purpose register rD.

**32-bit Implementation:**

$$rD[31:0] \leftarrow \text{extz}(\text{Immediate}) \ll 16$$
**64-bit Implementation:**

$$rD[63:0] \leftarrow \text{extz}(\text{Immediate}) \ll 16$$
**Exceptions:**

None

# l.msb      Multiply and Subtract Signed      l.msb

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	4	3	.	.	0
opcode 0x31					reserved					A					B					reserved					opcode 0x2					
6 bits					5 bits					5 bits					5 bits					7 bits					4 bits					

## Format:

l.msb rA, rB

## Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are multiplied, and the 64 bit result is subtracted from the special-purpose registers MACHI and MACLO. Result of the subtraction is placed into MACHI and MACLO registers. All operands are treated as signed integers.

The instruction will set the overflow flag if signed overflow is detecting during the subtraction stage.

## 32-bit Implementation:

$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{MACHI}[31:0]\text{MACLO}[31:0] - \text{rA}[31:0] * \text{rB}[31:0]$$

$$\text{SR}[\text{OV}] \leftarrow \text{signed overflow during subtraction stage}$$

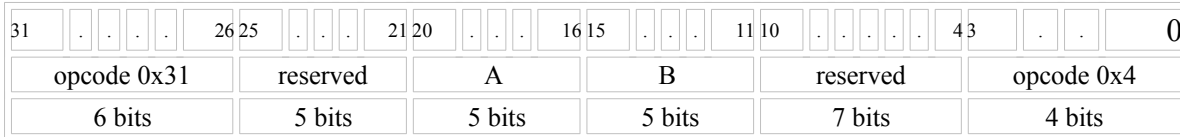
## 64-bit Implementation:

$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{MACHI}[31:0]\text{MACLO}[31:0] - \text{rA}[63:0] * \text{rB}[63:0]$$

$$\text{SR}[\text{OV}] \leftarrow \text{signed overflow during subtraction stage}$$

## Exceptions:

Range Exception on signed overflow if SR[OVE] and AECR[OVMACADDE] are set.

**l.msbu****Multiply and Subtract Unsigned****l.msb  
u****Format:**

l.msbu rA, rB

**Description:**

The contents of general-purpose register rA and the contents of general-purpose register rB are multiplied, and the 64 bit result is subtracted from the special-purpose registers MACHI and MACLO. Result of the subtraction is placed into MACHI and MACLO registers. All operands are treated as unsigned integers.

The instruction will set the overflow flag if unsigned overflow is detecting during the subtraction stage.

**32-bit Implementation:**

$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{MACHI}[31:0]\text{MACLO}[31:0] - \text{rA}[31:0] * \text{rB}[31:0]$$

$$\text{SR}[\text{CY}] \leftarrow \text{unsigned overflow during subtraction stage}$$
**64-bit Implementation:**

$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{MACHI}[31:0]\text{MACLO}[31:0] - \text{rA}[63:0] * \text{rB}[63:0]$$

$$\text{SR}[\text{CY}] \leftarrow \text{unsigned overflow during subtraction stage}$$
**Exceptions:**

Range Exception on signed overflow if SR[OVE] and AECR[CYMACADDE] are set.



# l.mtspr      Move To Special-Purpose Register      l.mtspr

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	.	.	.	.	0
opcode 0x30					K					A					B					K										
6 bits					5 bits					5 bits					5 bits					11 bits										

## Format:

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

### Description:

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

## 32-bit Implementation:

```
spr(rA OR Immediate) ← rB[31:0]
```

## 64-bit Implementation:

```
spr(rA OR Immediate) ← rB[31:0]
```

## Exceptions:

None

**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					reserved					opcode 0x3					reserved					opcode 0x6							
6 bits					5 bits					5 bits					5 bits					1 bit					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.

The instruction will set the overflow flag on signed overflow.

**32-bit Implementation:**

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

**64-bit Implementation:**

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

**Exceptions:**

Range Exception on signed overflow if SR[OVE] and AECR[OVMULE] are set.



**l.muld****Multiply Signed to Double****l.muld**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10			9				8	7	.	.	.	.	4	3	.	.			0
opcode 0x38					reserved					A					B					reserved					opcode 0x3					reserved					opcode 0x7								
6 bits					5 bits					5 bits					5 bits					1 bit					2 bits					4 bits					4 bits								

**Format:**

```
l.muld rA, rB
```

**Description:**

The contents of general-purpose register rA and the contents of general-purpose register rB are multiplied, and the result is stored in the MACHI and MACLO registers. Both operands are treated as signed integers.

The instruction will set the overflow flag on signed overflow.

**32-bit Implementation:**

$$\text{MACHI}[31:0] \text{MACLO}[31:0] \leftarrow \text{rA}[31:0] * \text{rB}[31:0]$$
**64-bit Implementation:**

$$\begin{aligned} \text{MACHI}[31:0] \text{MACLO}[31:0] &\leftarrow \text{rA}[63:0] * \text{rB}[63:0] \\ \text{SR}[\text{OV}] &\leftarrow \text{signed overflow} \end{aligned}$$
**Exceptions:**

Range Exception on signed overflow if SR[OVE] and AECR[OVMLE] are set.

# l.muldu      Multiply Unsigned to Double      l.muldu

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

## Format:

```
l.muldu rA, rB
```

## Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are multiplied, and the result is stored in the MACHI and MACLO registers. Both operands are treated as unsigned integers.

The instruction will set the overflow flag on unsigned overflow.

## 32-bit Implementation:

$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{rA}[31:0] * \text{rB}[31:0]$$

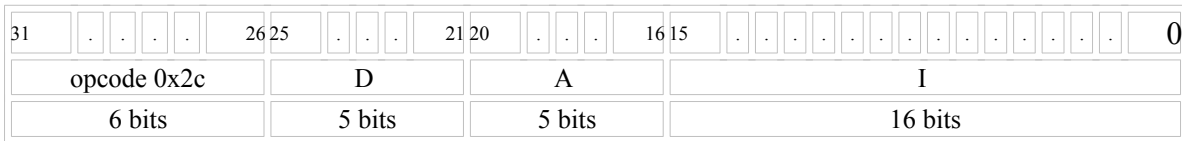
## 64-bit Implementation:

$$\text{MACHI}[31:0]\text{MACLO}[31:0] \leftarrow \text{rA}[63:0] * \text{rB}[63:0]$$

$$\text{SR}[\text{CY}] \leftarrow \text{unsigned overflow}$$

## Exceptions:

Range Exception on signed overflow if SR[OVE] and AECR[CYMULE] are set.

**l.muli****Multiply Immediate Signed****l.muli****Format:**

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

**Description:**

The immediate value and the contents of general-purpose register rA are multiplied, and the result is truncated to destination register width and placed into general-purpose register rD.

The instruction will set the overflow flag on signed overflow.

**32-bit Implementation:**

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

**64-bit Implementation:**

```
rD[63:0] ← rA[63:0] * exts(Immediate)
SR[OV]   ← signed overflow
```

**Exceptions:**

Range Exception on signed overflow if SR[OVE] and AECR[OVMLE] are set.

**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					reserved					opcode 0x3					reserved					opcode 0xb								
6 bits					5 bits					5 bits					5 bits					1 bit					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.

The instruction will set the carry flag on unsigned overflow.

**32-bit Implementation:**

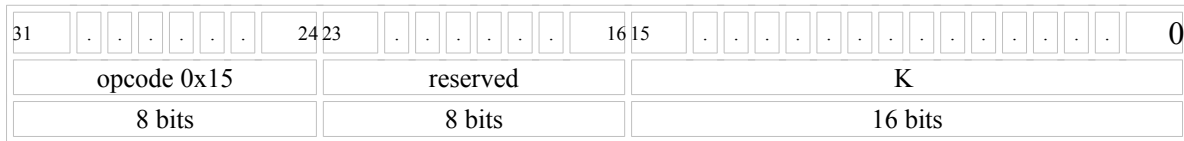
```
rD[31:0] ← rA[31:0] * rB[31:0]
SR[CY]   ← carry (unsigned overflow)
```

**64-bit Implementation:**

```
rD[63:0] ← rA[63:0] * rB[63:0]
SR[CY]   ← carry (unsigned overflow)
```

**Exceptions:**

Range Exception on unsigned overflow if SR[OVE] and AECR[CYMULE] are set.

**l.nop****No Operation****l.nop****Format:**

l.nop K

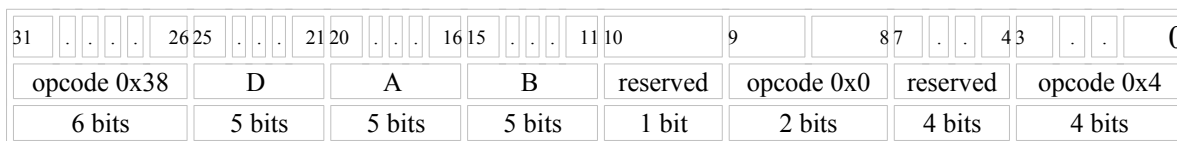
**Description:**

This instruction does not do anything except that it takes at least one clock cycle to complete. It is often used to fill delay slot gaps. Immediate value can be used for simulation purposes.

**32-bit Implementation:****64-bit Implementation:****Exceptions:**

None

Instruction Class  
ORBIS32 I

**l.or****Or****l.or****Format:**

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

**Description:**

The contents of general-purpose register rA are combined with the contents of general-purpose register rB in a bit-wise logical OR operation. The result is placed into general-purpose register rD.

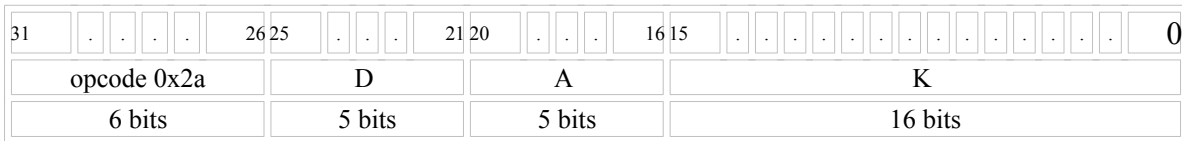
**32-bit Implementation:**

$$rD[31:0] \leftarrow rA[31:0] \text{ OR } rB[31:0]$$
**64-bit Implementation:**

$$rD[63:0] \leftarrow rA[63:0] \text{ OR } rB[63:0]$$
**Exceptions:**

None

# l.ori                      Or with Immediate Half Word                      l.ori



Format:

`l.ori 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 OR operation. The result is placed into general-purpose register rD.

## 32-bit Implementation:

$rD[31:0] \leftarrow rA[31:0] \text{ OR } \text{extz}(\text{Immediate})$

## 64-bit Implementation:

$rD[63:0] \leftarrow rA[63:0] \text{ OR } \text{extz}(\text{Immediate})$

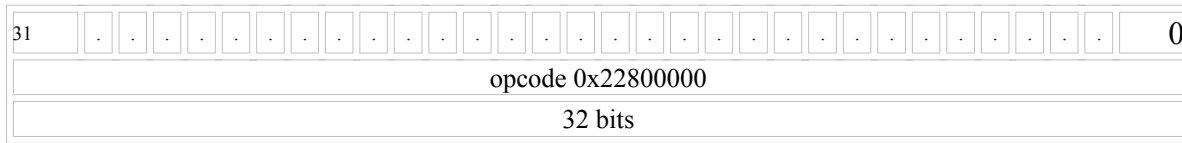
## Exceptions:

None

## l.psync

## Pipeline Synchronization

## l.psync



**Format:**

```
l.psync
```

### Description:

Execution of pipeline synchronization instruction results in completion of all instructions that were fetched before `l.psync` instruction. Once all instructions are completed, instructions fetched after `l.psync` are flushed from the pipeline and fetched again.

### 32-bit Implementation:

pipeline-synchronization

### 64-bit Implementation:

pipeline-synchronization

## Exceptions:

None



**l.rfe**

## Return From Exception

**l.rfe**

**Format:**

```
1.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.

### 32-bit Implementation:

$$\text{PC} \leftarrow \text{EPCR}$$
$$\text{SR} \leftarrow \text{ESR}$$

### 64-bit Implementation:

$$\text{PC} \leftarrow \text{EPCR}$$
$$\text{SR} \leftarrow \text{ESR}$$

## Exceptions:

None

**l.ror****Rotate Right****l.ror**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10				9	.	.	.	.	6	5				4	3	.	.	.	.	0
opcode 0x38					D					A					B					reserved					opcode 0x3					reserved					opcode 0x8									
6 bits					5 bits					5 bits					5 bits					1 bit					4 bits					2 bits					4 bits									

**Format:**

```
l.ror rD, rA, rB
```

**Description:**

General-purpose register rB specifies the number of bit positions; the contents of general-purpose register rA are rotated right. The result is written into general-purpose register rD.

**32-bit Implementation:**

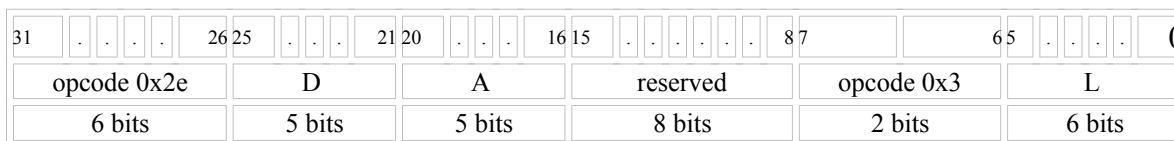
```
rD[31-rB[4:0]:0] ← rA[31:rB[4:0]]
rD[31:32-rB[4:0]] ← rA[rB[4:0]-1:0]
```

**64-bit Implementation:**

```
rD[63-rB[5:0]:0] ← rA[63:rB[5:0]]
rD[63:64-rB[5:0]] ← rA[rB[5:0]-1:0]
```

**Exceptions:**

None

**l.rori****Rotate Right with Immediate****l.rori****Format:**

```
l.rori rD, rA, L
```

**Description:**

The 6-bit immediate value specifies the number of bit positions; the contents of general-purpose register rA are rotated right. The result is written into general-purpose register rD. In 32-bit implementations bit 5 of immediate is ignored.

**32-bit Implementation:**

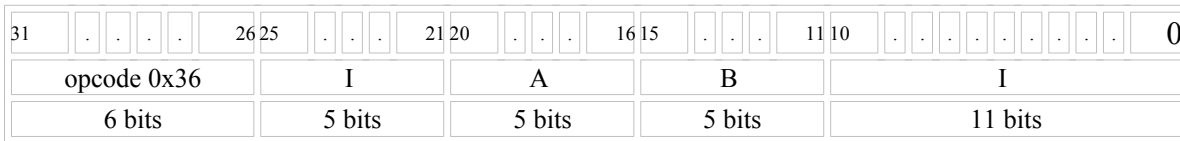
```
rD[31-L:0] ← rA[31:L]
rD[31:32-L] ← rA[L-1:0]
```

**64-bit Implementation:**

```
rD[63-L:0] ← rA[63:L]
rD[63:64-L] ← rA[L-1:0]
```

**Exceptions:**

None

**l.sb****Store Byte****l.sb****Format:**

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

**Description:**

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The low-order 8 bits of general-purpose register rB are stored to memory location addressed by EA.

**32-bit Implementation:**

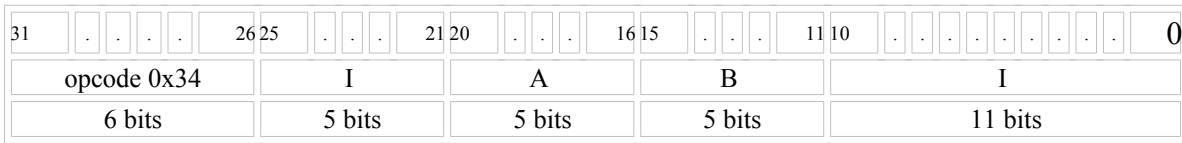
```
EA          ← exts(Immediate) + rA[31:0]
(EA)[7:0]   ← rB[7:0]
```

**64-bit Implementation:**

```
EA          ← exts(Immediate) + rA[63:0]
(EA)[7:0]   ← rB[7:0]
```

**Exceptions:**

```
TLB miss
Page fault
Bus error
```

**l.sd****Store Double Word****l.sd****Format:**

```
l.sd I(rA), rB
```

**Description:**

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The double word in general-purpose register rB is stored to memory location addressed by EA.

**32-bit Implementation:**

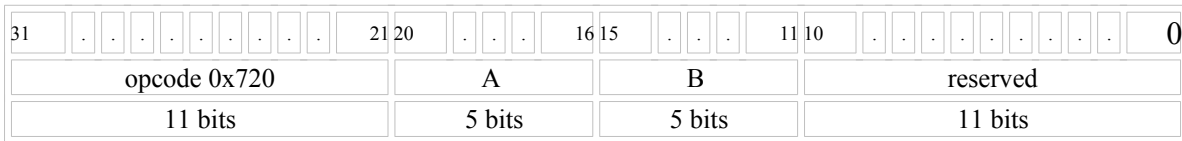
N/A

**64-bit Implementation:**

```
EA ← exts(Immediate) + rA[63:0]
(EA)[63:0] ← rB[63:0]
```

**Exceptions:**

TLB miss  
 Page fault  
 Bus error  
 Alignment

**l.sfeq****Set Flag if Equal****l.sfeq****Format:**

```
l.sfeq rA, rB
```

**Description:**

The contents of general-purpose registers rA and rB are compared. If the contents are equal, the compare flag is set; otherwise the compare flag is cleared.

**32-bit Implementation:**

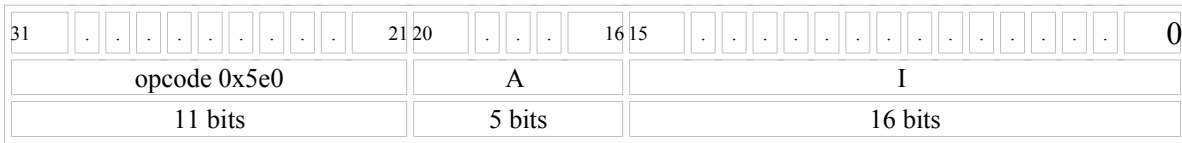
$$SR[F] \leftarrow rA[31:0] == rB[31:0]$$
**64-bit Implementation:**

$$SR[F] \leftarrow rA[63:0] == rB[63:0]$$
**Exceptions:**

None

**l.sfeqi**

## Set Flag if Equal Immediate

**l.sfeqi**

**Format:**

$$1.\text{sfeqi} \quad r_{A,I}$$

### Description:

The contents of general-purpose register rA and the sign-extended immediate value are compared. If the two values are equal, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

```
SR[F] ← rA[31:0] == exts(Immediate)
```

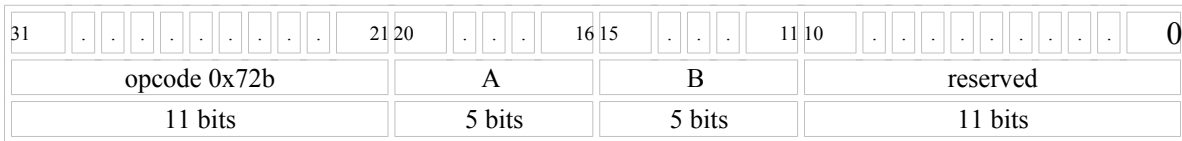
### 64-bit Implementation:

```
SR[F] ← rA[63:0] == exts(Immediate)
```

## Exceptions:

None

## l.sfges Set Flag if Greater or Equal Than Signed l.sfges



### Format:

```
l.sfges rA, rB
```

### Description:

The contents of general-purpose registers rA and rB are compared as signed integers. If the contents of the first register are greater than or equal to the contents of the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

```
SR[F] ← rA[31:0] >= rB[31:0]
```

### 64-bit Implementation:

```
SR[F] ← rA[63:0] >= rB[63:0]
```

### Exceptions:

None

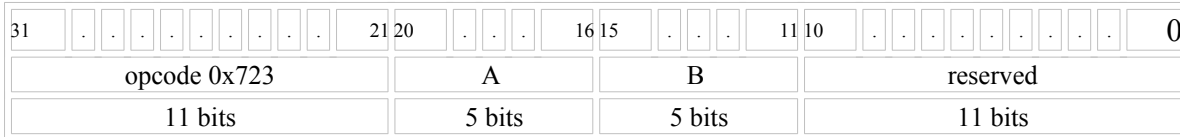


## l.sfgesi

## Set Flag if Greater or Equal Than Immediate Signed

# l.sfgeu      Set Flag if Greater or Equal Than      l.sfgeu

## Unsigned



### Format:

l.sfgeu rA, rB

### Description:

The contents of general-purpose registers rA and rB are compared as unsigned integers. If the contents of the first register are greater than or equal to the contents of the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$SR[F] \leftarrow rA[31:0] \geq rB[31:0]$

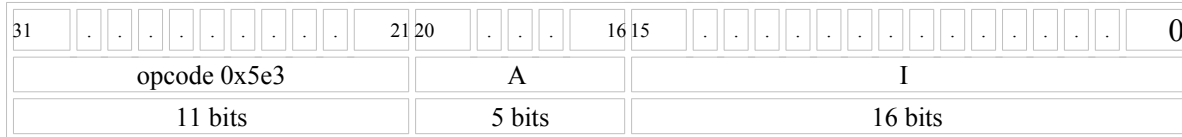
### 64-bit Implementation:

$SR[F] \leftarrow rA[63:0] \geq rB[63:0]$

### Exceptions:

None

# l.sfgeui Set Flag if Greater or Equal Than Immediate Unsigned l.sfgeui



### Format:

1.sfgeui rA,I

### Description:

The contents of general-purpose register rA and the sign-extended immediate value are compared as unsigned integers. If the contents of the first register are greater than or equal to the immediate value the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

```
SR[F] ← rA[31:0] >= exts(Immediate)
```

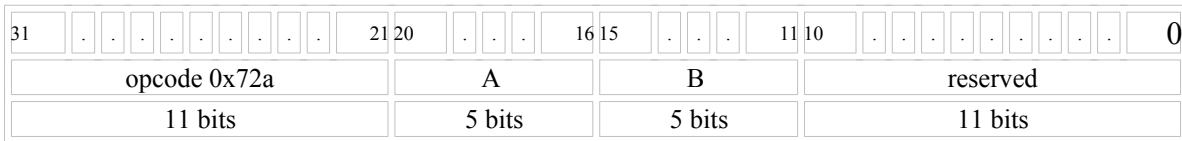
### 64-bit Implementation:

```
SR[F] ← rA[63:0] >= exts(Immediate)
```

### Exceptions:

None

# l.sfgts      Set Flag if Greater Than Signed      l.sfgts



## Format:

```
l.sfgts rA, rB
```

## Description:

The contents of general-purpose registers rA and rB are compared as signed integers. If the contents of the first register are greater than the contents of the second register, the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

$$SR[F] \leftarrow rA[31:0] > rB[31:0]$$

## 64-bit Implementation:

$$SR[F] \leftarrow rA[63:0] > rB[63:0]$$

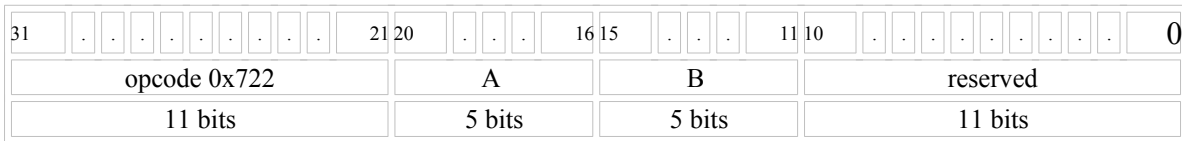
## Exceptions:

None

**l.sfgtsi**

## Set Flag if Greater Than Immediate Signed

# l.sfgtu      Set Flag if Greater Than Unsigned      l.sfgtu



## Format:

`l.sfgtu rA, rB`

## Description:

The contents of general-purpose registers rA and rB are compared as unsigned integers. If the contents of the first register are greater than the contents of the second register, the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

$SR[F] \leftarrow rA[31:0] > rB[31:0]$

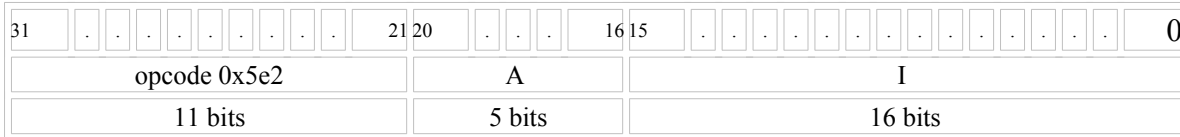
## 64-bit Implementation:

$SR[F] \leftarrow rA[63:0] > rB[63:0]$

## Exceptions:

None

# **l.sfgtui      Set Flag if Greater Than Immediate      l.sfgtui** **Unsigned**



### Format:

1.sfgtui rA,I

### Description:

The contents of general-purpose register rA and the sign-extended immediate value are compared as unsigned integers. If the contents of the first register are greater than the immediate value the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

```
SR[F] ← rA[31:0] > exts(Immediate)
```

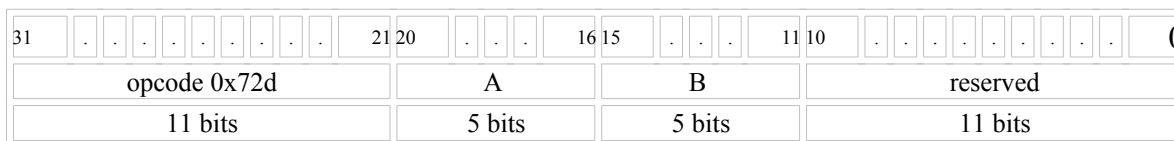
### 64-bit Implementation:

```
SR[F] ← rA[63:0] > exts(Immediate)
```

## Exceptions:

None

## l.sfles      Set Flag if Less or Equal Than Signed      l.sfles



### Format:

```
l.sfles rA, rB
```

### Description:

The contents of general-purpose registers rA and rB are compared as signed integers. If the contents of the first register are less than or equal to the contents of the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$$SR[F] \leftarrow rA[31:0] \leq rB[31:0]$$

### 64-bit Implementation:

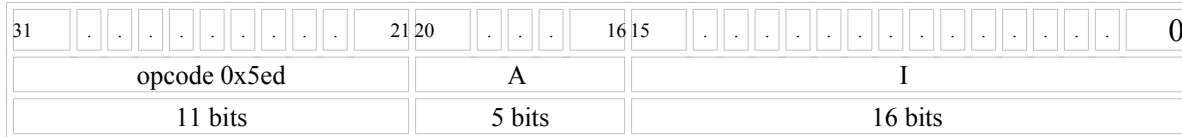
$$SR[F] \leftarrow rA[63:0] \leq rB[63:0]$$

### Exceptions:

None



**l.sflesi    Set Flag if Less or Equal Than Immediate    l.sflesi**  
**Signed**



### Format:

1.sflesi rA,I

### Description:

The contents of general-purpose register rA and the sign-extended immediate value are compared as signed integers. If the contents of the first register are less than or equal to the immediate value the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

```
SR[F] ← rA[31:0] <= exts(Immediate)
```

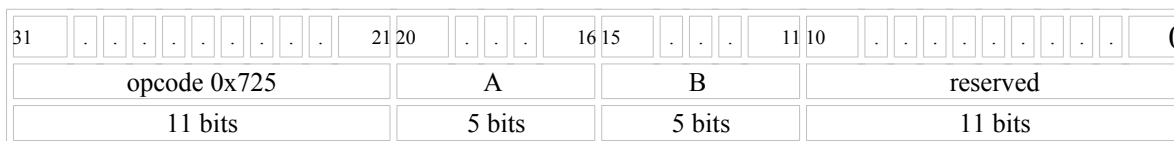
### 64-bit Implementation:

```
SR[F] ← rA[63:0] <= exts(Immediate)
```

## Exceptions:

None

## l.sfleu Set Flag if Less or Equal Than Unsigned l.sfleu



### Format:

```
l.sfleu rA, rB
```

### Description:

The contents of general-purpose registers rA and rB are compared as unsigned integers. If the contents of the first register are less than or equal to the contents of the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$$SR[F] \leftarrow rA[31:0] \leq rB[31:0]$$

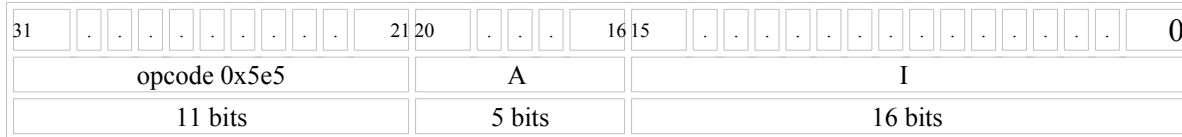
### 64-bit Implementation:

$$SR[F] \leftarrow rA[63:0] \leq rB[63:0]$$

### Exceptions:

None

## l.sfleui Set Flag if Less or Equal Than Immediate Unsigned l.sfleui



### Format:

1.sfleui rA,I

### Description:

The contents of general-purpose register rA and the sign-extended immediate value are compared as unsigned integers. If the contents of the first register are less than or equal to the immediate value the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

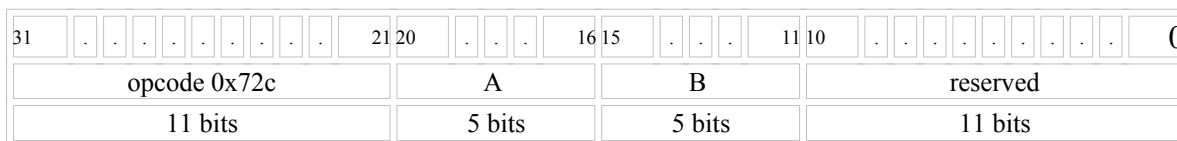
```
SR[F] ← rA[31:0] <= exts(Immediate)
```

### 64-bit Implementation:

```
SR[F] ← rA[63:0] <= exts(Immediate)
```

## Exceptions:

None

**l.sflts****Set Flag if Less Than Signed****l.sflts****Format:**

```
l.sflts rA, rB
```

**Description:**

The contents of general-purpose registers rA and rB are compared as signed integers. If the contents of the first register are less than the contents of the second register, the compare flag is set; otherwise the compare flag is cleared.

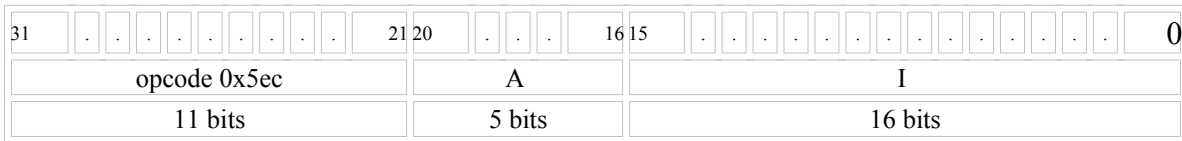
**32-bit Implementation:**

$$SR[F] \leftarrow rA[31:0] < rB[31:0]$$
**64-bit Implementation:**

$$SR[F] \leftarrow rA[63:0] < rB[63:0]$$
**Exceptions:**

None

## l.sfltsi Set Flag if Less Than Immediate Signed l.sfltsi



**Format:**

1.sfltsi rA,I

### Description:

The contents of general-purpose register rA and the sign-extended immediate value are compared as signed integers. If the contents of the first register are less than the immediate value the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

```
SR[F] ← rA[31:0] < exts(Immediate)
```

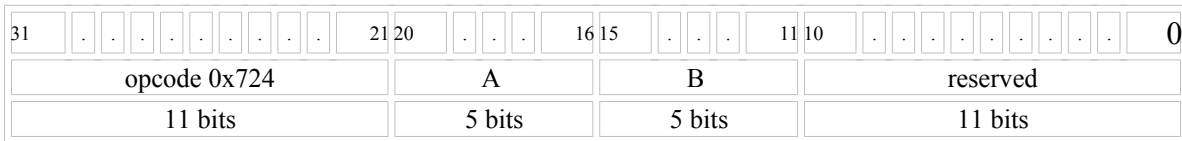
### 64-bit Implementation:

```
SR[F] ← rA[63:0] < exts(Immediate)
```

## Exceptions:

None

# l.sftu      Set Flag if Less Than Unsigned      l.sftu



## Format:

```
l.sftu rA, rB
```

## Description:

The contents of general-purpose registers rA and rB are compared as unsigned integers. If the contents of the first register are less than the contents of the second register, the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

$$SR[F] \leftarrow rA[31:0] < rB[31:0]$$

## 64-bit Implementation:

$$SR[F] \leftarrow rA[63:0] < rB[63:0]$$

## Exceptions:

None

31	.	.	.	.	.	.	.	.	.	21	20	.	.	.	16	15	.	.	.	.	.	.	.	.	.	.	.	.	.	0
<b>opcode 0x5e4</b>											<b>A</b>			<b>I</b>																
<b>11 bits</b>											<b>5 bits</b>			<b>16 bits</b>																

1.sfltui rA,I

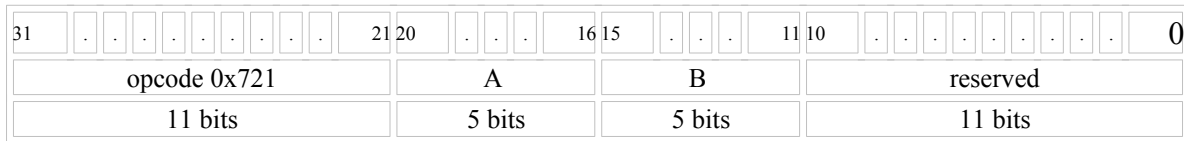
1.sfltui rA,I

The contents of general-purpose register rA and the sign-extended immediate value are compared as unsigned integers. If the contents of the first register are less than the immediate value the compare flag is set; otherwise the compare flag is cleared.

```
SR[F] ← rA[31:0] < exts(Immediate)
```

```
SR[F] ← rA[63:0] < exts(Immediate)
```

## None

**l.sfne****Set Flag if Not Equal****l.sfne****Format:**

```
l.sfne rA, rB
```

**Description:**

The contents of general-purpose registers rA and rB are compared. If the contents are not equal, the compare flag is set; otherwise the compare flag is cleared.

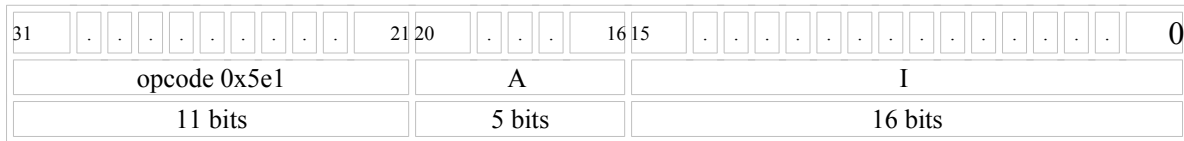
**32-bit Implementation:**

$$SR[F] \leftarrow rA[31:0] \neq rB[31:0]$$
**64-bit Implementation:**

$$SR[F] \leftarrow rA[63:0] \neq rB[63:0]$$
**Exceptions:**

None



**l.sfnei**

**Format:**

1.sfnei rA,I

### Description:

The contents of general-purpose register rA and the sign-extended immediate value are compared. If the two values are not equal, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

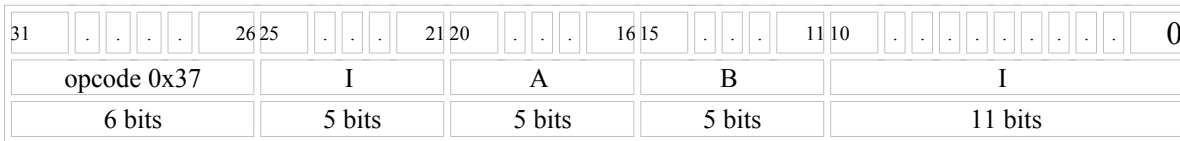
```
SR[F] ← rA[31:0] != exts(Immediate)
```

### 64-bit Implementation:

```
SR[F] ← rA[63:0] != exts(Immediate)
```

## Exceptions:

None

**l.sh****Store Half Word****l.sh****Format:**

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

**Description:**

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The low-order 16 bits of general-purpose register rB are stored to memory location addressed by EA.

**32-bit Implementation:**

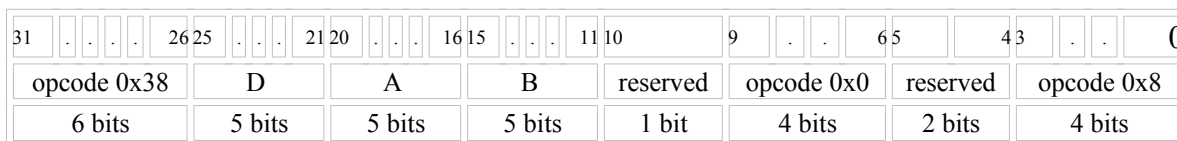
```
EA      ← exts(Immediate) + rA[31:0]
(EA)[15:0] ← rB[15:0]
```

**64-bit Implementation:**

```
EA      ← exts(Immediate) + rA[63:0]
(EA)[15:0] ← rB[15:0]
```

**Exceptions:**

```
TLB miss
Page fault
Bus error
Alignment
```

**l.sll****Shift Left Logical****l.sll****Format:**

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

**Description:**

General-purpose register rB specifies the number of bit positions; the contents of general-purpose register rA are shifted left, inserting zeros into the low-order bits. The result is written into general-purpose rD. In 32-bit implementations bit 5 of rB is ignored.

**32-bit Implementation:**

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

**64-bit Implementation:**

```
rD[63:rB[5:0]] ← rA[63-rB[5:0]:0]
rD[rB[5:0]-1:0] ← 0
```

**Exceptions:**

None

# l.slli      Shift Left Logical with Immediate      l.slli

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	.	.	8	7					6	5	.	.	.	.	0
opcode 0x2e					D					A					reserved								opcode 0x0				L							
6 bits					5 bits					5 bits					8 bits								2 bits				6 bits							

## Format:

```
l.slli rD, rA, L
```

## Description:

The immediate value specifies the number of bit positions; the contents of general-purpose register rA are shifted left, inserting zeros into the low-order bits. The result is written into general-purpose register rD. In 32-bit implementations bit 5 of immediate is ignored.

## 32-bit Implementation:

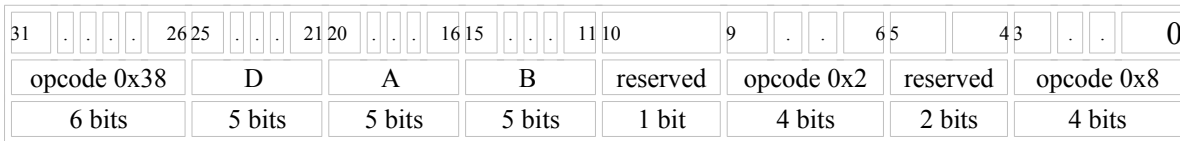
```
rD[31:L] ← rA[31-L:0]
rD[L-1:0] ← 0
```

## 64-bit Implementation:

```
rD[63:L] ← rA[63-L:0]
rD[L-1:0] ← 0
```

## Exceptions:

None

**l.sra****Shift Right Arithmetic****l.sra****Format:**

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

**Description:**

General-purpose register rB specifies the number of bit positions; the contents of general-purpose register rA are shifted right, sign-extending the high-order bits. The result is written into general-purpose register rD. In 32-bit implementations bit 5 of rB is ignored.

**32-bit Implementation:**

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

**64-bit Implementation:**

```
rD[63-rB[5:0]:0] ← rA[63:rB[5:0]]
rD[63:64-rB[5:0]] ← rA[63]
```

**Exceptions:**

None

# l.srai      Shift Right Arithmetic with Immediate      l.srai

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	.	.	8	7					6	5	.	.	.	.	0
opcode 0x2e					D					A					reserved								opcode 0x2				L							
6 bits					5 bits					5 bits					8 bits								2 bits				6 bits							

## Format:

```
l.srai rD, rA, L
```

## Description:

The 6-bit immediate value specifies the number of bit positions; the contents of general-purpose register rA are shifted right, sign-extending the high-order bits. The result is written into general-purpose register rD. In 32-bit implementations bit 5 of immediate is ignored.

## 32-bit Implementation:

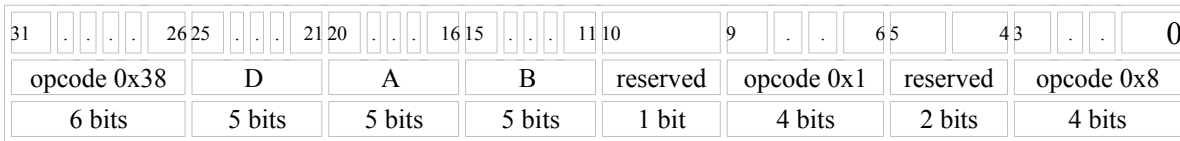
```
rD[31-L:0] ← rA[31:L]
rD[31:32-L] ← rA[31]
```

## 64-bit Implementation:

```
rD[63-L:0] ← rA[63:L]
rD[63:64-L] ← rA[63]
```

## Exceptions:

None

**l.srl****Shift Right Logical****l.srl****Format:**

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

**Description:**

General-purpose register rB specifies the number of bit positions; the contents of general-purpose register rA are shifted right, inserting zeros into the high-order bits. The result is written into general-purpose register rD. In 32-bit implementations bit 5 of rB is ignored.

**32-bit Implementation:**

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

**64-bit Implementation:**

```
rD[63-rB[5:0]:0] ← rA[63:rB[5:0]]
rD[63:64-rB[5:0]] ← 0
```

**Exceptions:**

None

# l.srli      Shift Right Logical with Immediate      l.srli

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	.	.	8	7					6	5	.	.	.	.	0
opcode 0x2e					D					A					reserved								opcode 0x1				L							
6 bits					5 bits					5 bits					8 bits								2 bits				6 bits							

## Format:

```
l.srli rD, rA, L
```

## Description:

The 6-bit immediate value specifies the number of bit positions; the contents of general-purpose register rA are shifted right, inserting zeros into the high-order bits. The result is written into general-purpose register rD. In 32-bit implementations bit 5 of immediate is ignored.

## 32-bit Implementation:

```
rD[31-L:0] ← rA[31:L]
rD[31:32-L] ← 0
```

## 64-bit Implementation:

```
rD[63-L:0] ← rA[63:L]
rD[63:64-L] ← 0
```

## Exceptions:

None



**l.sub****Subtract****l.sub**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10			9				8	7	.	.	.	.	4	3	.	.			0
opcode 0x38					D					A					B					reserved					opcode 0x0					reserved					opcode 0x2								
6 bits					5 bits					5 bits					5 bits					1 bit					2 bits					4 bits					4 bits								

**Format:**

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

**Description:**

The contents of general-purpose register rB are subtracted from the contents of general-purpose register rA to form the result. The result is placed into general-purpose register rD.

The instruction will set the carry flag on unsigned overflow, and the overflow flag on signed overflow.

**32-bit Implementation:**

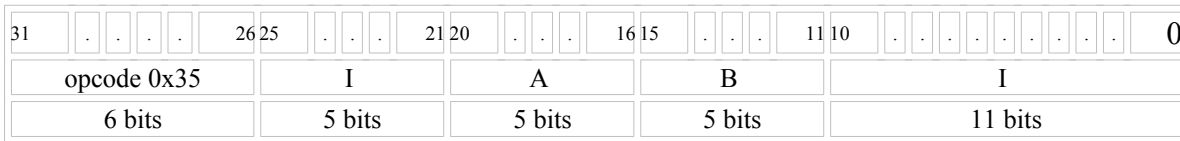
```
rD[31:0] ← rA[31:0] - rB[31:0]
SR[CY]   ← carry (unsigned overflow)
SR[OV]   ← signed overflow
```

**64-bit Implementation:**

```
rD[63:0] ← rA[63:0] - rB[63:0]
SR[CY]   ← carry (unsigned overflow)
SR[OV]   ← signed overflow
```

**Exceptions:**

Range Exception on overflow if SR[OVE] and AECR[OVADDE] are set.  
 Range Exception on carry if SR[OVE] and AECR[CYADDE] are set.

**l.sw****Store Single Word****l.sw****Format:**

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

**Description:**

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The low-order 32 bits of general-purpose register rB are stored to memory location addressed by EA.

**32-bit Implementation:**

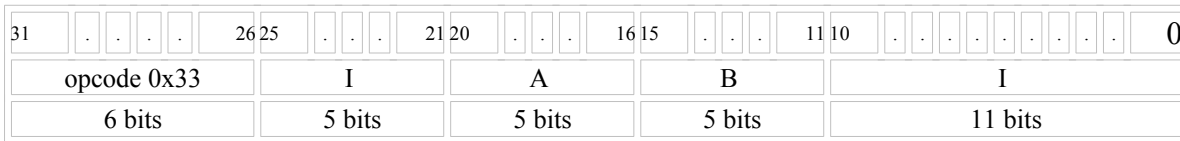
```
EA          ← exts(Immediate) + rA[31:0]
(EA)[31:0] ← rB[31:0]
```

**64-bit Implementation:**

```
EA          ← exts(Immediate) + rA[63:0]
(EA)[31:0] ← rB[31:0]
```

**Exceptions:**

```
TLB miss
Page fault
Bus error
Alignment
```

**l.swa****Store Single Word Atomic****l.swa****Format:**

```
l.swa I(rA), rB
```

**Description:**

The offset is sign-extended and added to the contents of general-purpose register rA. The sum represents an effective address. The low-order 32 bits of general-purpose register rB are conditionally stored to memory location addressed by EA. The 'atomic' condition relies on that an atomic reserve to EA is still intact. When the MMU is enabled, the physical translation of EA is used to do the address comparison.

**32-bit Implementation:**

```
EA          ← exts(Immediate) + rA[31:0]
if (atomic) (EA)[31:0] ← rB[31:0]
SR[F]       ← atomic
```

**64-bit Implementation:**

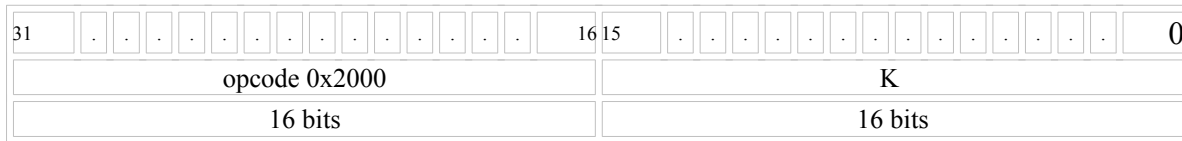
```
EA          ← exts(Immediate) + rA[63:0]
if (atomic) (EA)[31:0] ← rB[31:0]
SR[F]       ← atomic
```

**Exceptions:**

```
TLB miss
Page fault
Bus error
Alignment
```

# l.sys

## System Call

**l.sys**

### Format:

1.sys K

### Description:

Execution of the system call 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.

Because an l.sys causes an intentional exception, rather than an interruption of normal processing, the matching l.rfe returns to the next instruction. As this is considered to be the jump itself for exceptions occurring in a delay slot, l.sys should not be placed in a delay slot.

### 32-bit Implementation:

system-call-exception (K)

### 64-bit Implementation:

system-call-exception (K)

## Exceptions:

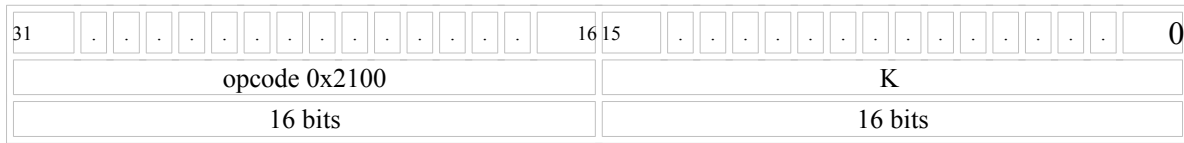
## System Call

Instruction Class  
ORBIS32 I

## l.trap

# Trap

## l.trap



**Format:**

```
1.trap K
```

### Description:

Trap exception is a request to the operating system or to the debug facility to execute certain debug services. Immediate value is used to select which SR bit is tested by trap instruction.

### 32-bit Implementation:

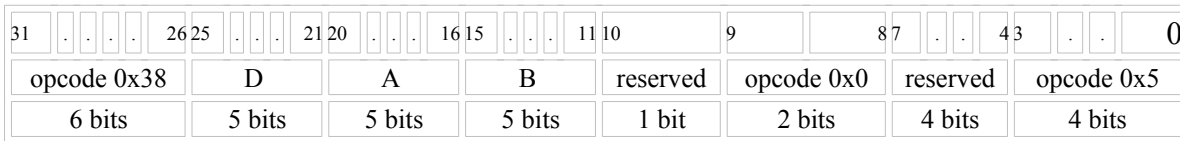
```
trap-exception()
```

### 64-bit Implementation:

```
trap-exception()
```

## Exceptions:

## Trap exception

**l.xor****Exclusive Or****l.xor****Format:**

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

**Description:**

The contents of general-purpose register rA are combined with the contents of general-purpose register rB in a bit-wise logical XOR operation. The result is placed into general-purpose register rD.

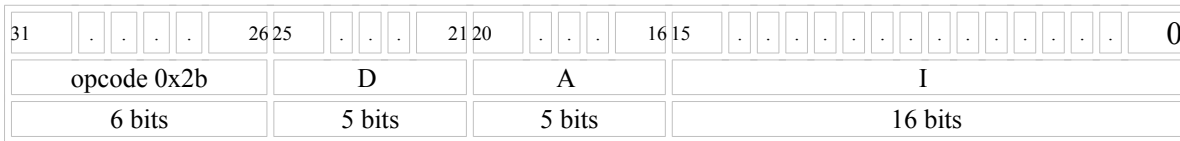
**32-bit Implementation:**

$$rD[31:0] \leftarrow rA[31:0] \text{ XOR } rB[31:0]$$
**64-bit Implementation:**

$$rD[63:0] \leftarrow rA[63:0] \text{ XOR } rB[63:0]$$
**Exceptions:**

None

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



### Format:

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

### Description:

The immediate value is sign-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 exts(Immediate)
```

### 64-bit Implementation:

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

### Exceptions:

None

## 5.4 ORFPX32/64

### lf.add.d Add Floating-Point Double-Precision lf.add.d

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	0
opcode 0x32					D / D1					A / A1					B / B1					reserved					opcode 0x10					
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits					

#### Format (32/64-bit):

```
lf.add.d rD1, rD2, rA1, rA2, rB1, rB2
lf.add.d rD, rA, rB
```

#### Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} are added to the contents of general-purpose registers pair {rB1,rB2} to form the result. The result is placed into general-purpose registers pair {rD1,rD2}. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine 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:

$$\{rD1[31:0], rD2[31:0]\} \leftarrow \{rA1[31:0], rA2[31:0]\} + \{rB1[31:0], rB2[31:0]\}$$

#### 64-bit Implementation:

$$rD[63:0] \leftarrow rA[63:0] + rB[63:0]$$

#### Exceptions:

Floating Point



## lf.add.s    Add Floating-Point Single-Precision    lf.add.s

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						D					A					B					reserved					opcode 0x0							
6 bits						5 bits					5 bits					5 bits					3 bits					8 bits							

### Format:

```
lf.add.s 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. On 64-bit machine the result should be NaN-boxed.

### 32-bit Implementation:

$$rD[31:0] \leftarrow rA[31:0] + rB[31:0]$$

### 64-bit Implementation:

$$rD[31:0] \leftarrow rA[31:0] + rB[31:0]$$

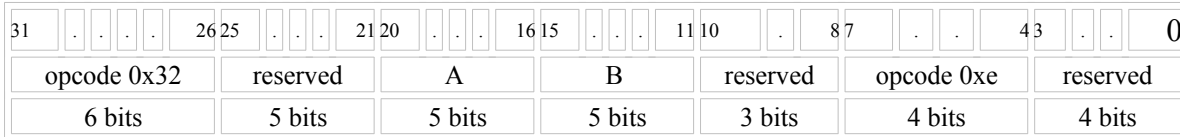
$$rD[63:32] \leftarrow 0xFFFFFFFF$$

### Exceptions:

Floating Point

## Reserved for ORFPX64 Custom Instructions

**lf.cust1.d** **Reserved for ORFPX64 Custom Instructions** **lf.cust1.d**



### Format:

lf.cust1.d rA, rB

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but instead by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

N/A

### Exceptions:

N/A

## Reserved for ORFPX32 Custom Instructions

**lf.cust1.s** **Reserved for ORFPX32 Custom Instructions** **lf.cust1.s**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	4	3	.	.	0
opcode 0x32					reserved					A					B					reserved			opcode 0xd				reserved					
6 bits					5 bits					5 bits					5 bits					3 bits			4 bits				4 bits					

### Format:

lf.cust1.s rA, rB

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but instead by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

N/A

### Exceptions:

N/A

## lf.div.d Divide Floating-Point Double-Precision lf.div.d

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						D / D1					A / A1					B / B1					reserved			opcode 0x13										
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits										

### Format (32/64-bit):

```
lf.div.d rD1, rD2, rA1, rA2, rB1, rB2
```

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

### Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} are divided by the contents of general-purpose registers pair {rB1,rB2} to form the result. The result is placed into general-purpose registers pair {rD1,rD2}. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA are divided by the contents of general-purpose register rB to form the result. The result is placed into general-purpose register rD.

### 32-bit Implementation:

$$\{rD1[31:0], rD2[31:0]\} \leftarrow \{rA1[31:0], rA2[31:0]\} / \{rB1[31:0], rB2[31:0]\}$$

### 64-bit Implementation:

$$rD[63:0] \leftarrow rA[63:0] / rB[63:0]$$

### Exceptions:

Floating Point

## lf.div.s Divide Floating-Point Single-Precision lf.div.s

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10		.	8	7	.	.	.	.	.	.	0
opcode 0x32						D					A					B					reserved			opcode 0x3								
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits								

### Format:

```
lf.div.s 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. The result is placed into general-purpose register rD. On 64-bit machine the result should be NaN-boxed.

### 32-bit Implementation:

$$rD[31:0] \leftarrow rA[31:0] / rB[31:0]$$

### 64-bit Implementation:

$$rD[31:0] \leftarrow rA[31:0] / rB[31:0]$$

$$rD[63:32] \leftarrow 0xFFFFFFFF$$

### Exceptions:

Floating Point

# **lf.dtos.d      Convert Double-precision Floating-Point Number to Single-precision      lf.dtos.d**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						D						A / A1						reserved						reserved						opcode 0x35							
6 bits						5 bits						5 bits						5 bits						3 bits						8 bits							

## Format (32/64-bit):

```
lf.dtos.d rD, rA1, rA2
lf.dtos.d rD, rA
```

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} are converted from Double-precision to Single-precision. The result is placed into general-purpose register rD. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA are converted from Double-precision to Single-precision. The result is NaN-boxed and stored in register rD.

## 32-bit Implementation:

```
rD ← float({rA1[31:0], rA2[31:0]})
```

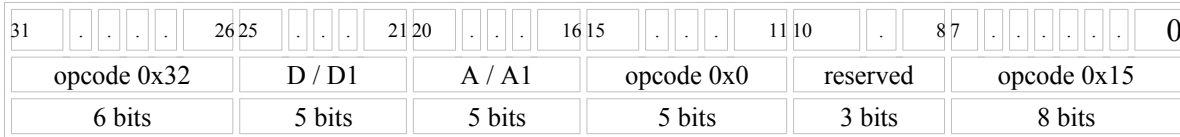
## 64-bit Implementation:

```
rD[31:0] ← float(rA[63:0])
rD[63:32] ← 0xFFFFFFFF
```

## Exceptions:

Floating Point

# **lf.ftoi.d      Floating-Point Double-Precision To      lf.ftoi.d** **Integer**



## Format (32/64-bit):

```
lf.ftoi.d rD1, rD2, rA1, rA2
lf.ftoi.d rD, rA
```

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} are converted to a 64-bit integer and stored in general-purpose registers pair {rD1,rD2}. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA are converted to a 64-bit integer and stored in general-purpose register rD.

The rounding mode for conversion shall be truncate towards zero.

## 32-bit Implementation:

```
{rD1[31:0], rD2[31:0]} ← ftoi({rA1[31:0], rA2[31:0]})
```

## 64-bit Implementation:

```
rD[63:0] ← ftoi(rA[63:0])
```

## Exceptions:

Floating Point

# **lf.ftoi.s      Floating-Point Single-Precision To Integer      lf.ftoi.s**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	0
opcode 0x32					D					A					opcode 0x0					reserved			opcode 0x5							
6 bits					5 bits					5 bits					5 bits					3 bits			8 bits							

## Format:

`lf.ftoi.s rD, rA`

## Description:

The contents of general-purpose register `rA` are converted to an integer and stored into general-purpose register `rD`.

The rounding mode for conversion shall be truncate towards zero.

## 32-bit Implementation:

$rD[31:0] \leftarrow ftoi(rA[31:0])$

## 64-bit Implementation:

$rD[31:0] \leftarrow ftoi(rA[31:0])$

$rD[63:32] \leftarrow rD[31]$

## Exceptions:

Floating Point



# **lf.itof.d      Integer To Floating-Point Double-Precision      lf.itof.d**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	0
opcode 0x32					D / D1					A / A1					opcode 0x0					reserved			opcode 0x14							
6 bits					5 bits					5 bits					5 bits					3 bits			8 bits							

## Format (32/64-bit):

```
lf.itof.d rD1, rD2, rA1, rA2
lf.itof.d rD, rA
```

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} are converted to a double-precision floating-point number and stored in general-purpose registers pair {rD1,rD2}. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA are converted to a double-precision floating-point number and stored in general-purpose register rD.

## 32-bit Implementation:

```
{rD1[31:0], rD2[31:0]} ← itof({rA1[31:0], rA2[31:0]})
```

## 64-bit Implementation:

```
rD[63:0] ← itof(rA[63:0])
```

## Exceptions:

Floating Point

# **lf.itof.s      Integer To Floating-Point Single-Precision      lf.itof.s**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	0
opcode 0x32					D					A					opcode 0x0					reserved					opcode 0x4					
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits					

## Format:

`lf.itof.s rD, rA`

## Description:

The contents of general-purpose register `rA` are converted to a single-precision floating-point number and stored into general-purpose register `rD`. On 64-bit machine the result should be NaN-boxed.

## 32-bit Implementation:

`rD[31:0] ← itof(rA[31:0])`

## 64-bit Implementation:

`rD[31:0] ← itof(rA[31:0])`  
`rD[63:32] ← 0xFFFFFFFF`

## Exceptions:

Floating Point

# **lf.madd.d      Multiply and Add Floating-Point      lf.madd.d** **Double-Precision**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	0
opcode 0x32					reserved					A / A1					B / B1					reserved					opcode 0x17					
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits					

## Format (32/64-bit):

```
lf.madd.d rA1, rA2, rB1, rB2
lf.madd.d rA, rB
```

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} are multiplied by the contents of general-purpose registers pair {rB1,rB2}, and added to special-purpose register FPMADDLO/FPMADDHI. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA are multiplied by the contents of general-purpose register rB, and added to special-purpose register FPMADDLO/FPMADDHI.

No intermediate rounding is performed.

## 32-bit Implementation:

$$\text{FPMADDHI}[31:0] \text{FPMADDLO}[31:0] \leftarrow \text{rA1}[31:0] \text{rA2}[31:0] * \text{rB1}[31:0] \text{rB2}[31:0] + \text{FPMADDHI}[31:0] \text{FPMADDLO}[31:0]$$

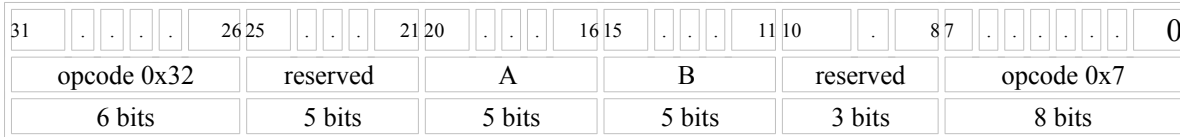
## 64-bit Implementation:

$$\text{FPMADDHI}[31:0] \text{FPMADDLO}[31:0] \leftarrow \text{rA}[63:0] * \text{rB}[63:0] + \text{FPMADDHI}[31:0] \text{FPMADDLO}[31:0]$$

## Exceptions:

Floating Point

# **lf.madd.s      Multiply and Add Floating-Point      lf.madd.s** **Single-Precision**



## Format:

lf.madd.s rA, rB

## Description:

The contents of general-purpose register rA are multiplied by the contents of general-purpose register rB, and added to special-purpose register FPMADDLO/FPMADDHI. No intermediate rounding is performed.

On 64-bit machine the result should be NaN-boxed.

## 32-bit Implementation:

$$\text{FPMADDHI}[31:0] \text{FPMADDLO}[31:0] \leftarrow \text{rA}[31:0] * \text{rB}[31:0] + \text{FPMADDHI}[31:0] \text{FPMADDLO}[31:0]$$

## 64-bit Implementation:

$$\text{FPMADDHI}[31:0] \text{FPMADDLO}[31:0] \leftarrow \text{rA}[31:0] * \text{rB}[31:0] + \text{FPMADDHI}[31:0] \text{FPMADDLO}[31:0]$$

$$\text{FPMADDHI}[63:32] \leftarrow 0xFFFFFFFF$$

$$\text{FPMADDLO}[63:32] \leftarrow 0xFFFFFFFF$$

## Exceptions:

Floating Point

# **lf.mul.d      Multiply Floating-Point Double-Precision      lf.mul.d**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	0
opcode 0x32						D / D1					A / A1					B / B1					reserved			opcode 0x12						
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits						

## Format (32/64-bit):

```
lf.mul.d rD1, rD2, rA1, rA2, rB1, rB2
lf.mul.d rD, rA, rB
```

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} are multiplied by the contents of general-purpose registers pair {rB1,rB2} to form the result. The result is placed into general-purpose registers pair {rD1,rD2}. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA are the contents of general-purpose register rB to form the result. The result is placed into general-purpose register rD.

## 32-bit Implementation:

$$\{rD1[31:0], rD2[31:0]\} \leftarrow \{rA1[31:0], rA2[31:0]\} * \{rB1[31:0], rB2[31:0]\}$$

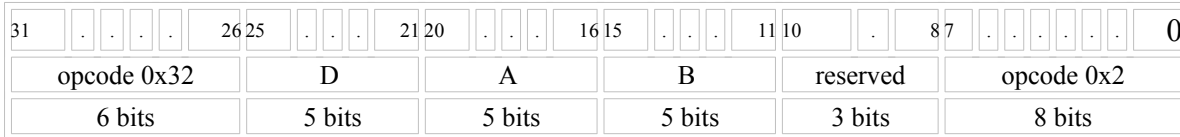
## 64-bit Implementation:

$$rD[63:0] \leftarrow rA[63:0] * rB[63:0]$$

## Exceptions:

Floating Point

# **lf.mul.s      Multiply Floating-Point Single-Precision      lf.mul.s**



## Format:

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

## Description:

The contents of general-purpose register rA are multiplied by the contents of general-purpose register rB to form the result. The result is placed into general-purpose register rD. On 64-bit machine the result should be NaN-boxed.

## 32-bit Implementation:

$rD[31:0] \leftarrow rA[31:0] * rB[31:0]$

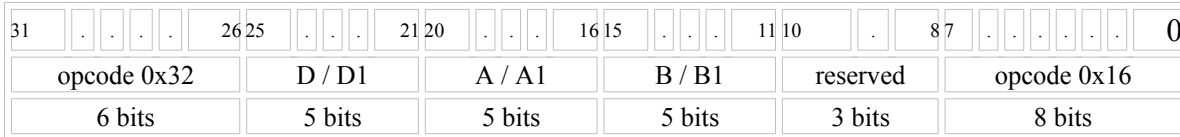
## 64-bit Implementation:

$rD[31:0] \leftarrow rA[31:0] * rB[31:0]$   
 $rD[63:32] \leftarrow 0xFFFFFFFF$

## Exceptions:

Floating Point

# **lf.rem.d      Remainder Floating-Point Double-Precision      lf.rem.d**



## Format (32/64-bit):

```
lf.rem.d rD1, rD2, rA1, rA2, rB1, rB2
lf.rem.d rD, rA, rB
```

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} are divided by the contents of general-purpose registers pair {rB1,rB2}, and remainder is used as the result. The result is placed into general-purpose registers pair {rD1,rD2}. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA are divided by the contents of general-purpose register rB, and remainder is used as the result. The result is placed into general-purpose register rD.

## 32-bit Implementation:

$$\{rD1[31:0], rD2[31:0]\} \leftarrow \{rA1[31:0], rA2[31:0]\} \% \{rB1[31:0], rB2[31:0]\}$$

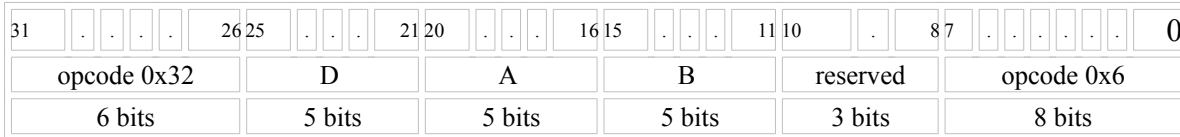
## 64-bit Implementation:

$$rD[63:0] \leftarrow rA[63:0] \% rB[63:0]$$

## Exceptions:

Floating Point

# **lf.rem.s      Remainder Floating-Point Single-Precision      lf.rem.s**



## Format:

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

## Description:

The contents of general-purpose register `rA` are divided by the contents of general-purpose register `rB`, and remainder is used as the result. The result is placed into general-purpose register `rD`. On 64-bit machine the result should be NaN-boxed.

## 32-bit Implementation:

$rD[31:0] \leftarrow rA[31:0] \% rB[31:0]$

## 64-bit Implementation:

$rD[31:0] \leftarrow rA[31:0] \% rB[31:0]$   
 $rD[63:32] \leftarrow 0xFFFFFFFF$

## Exceptions:

Floating Point



## lf.sfeq.d      Set Flag if Equal Floating-Point Double-Precision      lf.sfeq.d

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32					reserved					A / A1					B / B1					reserved					opcode 0x18													
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits													

### Format (32/64-bit):

```
lf.sfeq.d rA1, rA2, rB1, rB2
lf.sfeq.d rA, rB
```

### Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If the two registers pairs are equal, the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the two registers are equal, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$$SR[F] \leftarrow \{rA1[31:0], rA2[31:0]\} == \{rB1[31:0], rB2[31:0]\}$$

### 64-bit Implementation:

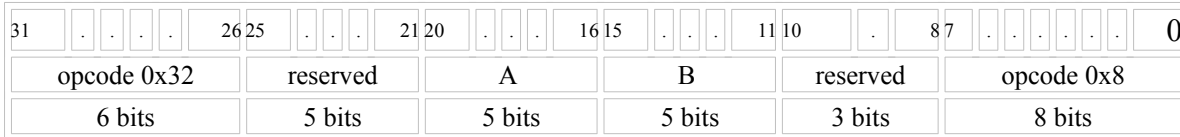
$$SR[F] \leftarrow rA[63:0] == rB[63:0]$$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is a signaling NaN, sets FPCSR[IVF].

## lf.sfeq.s Set Flag if Equal Floating-Point Single-Precision



### Format:

lf.sfeq.s rA, rB

### Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the two registers are equal, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$SR[F] \leftarrow rA[31:0] == rB[31:0]$

### 64-bit Implementation:

$SR[F] \leftarrow rA[31:0] == rB[31:0]$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is a signaling NaN, sets FPCSR[IVF].

# **lf.sfge.d      Set Flag if Greater or Equal Than      lf.sfge.d** **Floating-Point Double-Precision**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32					reserved					A / A1					B / B1					reserved					opcode 0x1b													
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits													

## Format (32/64-bit):

```
lf.sfge.d rA1, rA2, rB1, rB2
lf.sfge.d rA, rB
```

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If the first registers pair is greater than or equal to the second registers pair, the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the first register is greater than or equal to the second register, the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

$$SR[F] \leftarrow \{rA1[31:0], rA2[31:0]\} \geq \{rB1[31:0], rB2[31:0]\}$$

## 64-bit Implementation:

$$SR[F] \leftarrow rA[63:0] \geq rB[63:0]$$

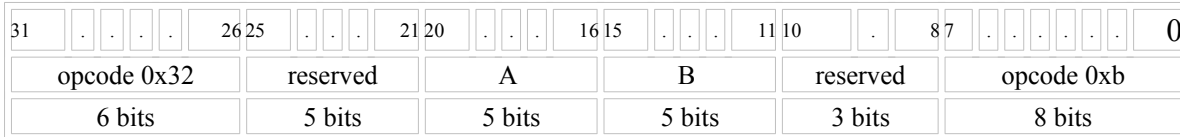
## Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is a NaN, sets FPCSR[IVF].

# lf.sfge.s      Set Flag if Greater or Equal Than      lf.sfge.s

## Floating-Point Single-Precision



### Format:

lf.sfge.s rA,rB

### Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the first register is greater than or equal to the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$SR[F] \leftarrow rA[31:0] \geq rB[31:0]$

### 64-bit Implementation:

$SR[F] \leftarrow rA[31:0] \geq rB[31:0]$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is a NaN, sets FPCSR[IVF].

## lf.sfgt.d Set Flag if Greater Than Floating-Point Double-Precision lf.sfgt.d

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32					reserved					A / A1					B / B1					reserved					opcode 0x1a													
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits													

### Format (32/64-bit):

```
lf.sfgt.d rA1, rA2, rB1, rB2
lf.sfgt.d rA, rB
```

### Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If the first registers pair is greater than the second registers pair, the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the first register is greater than the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$$SR[F] \leftarrow \{rA1[31:0], rA2[31:0]\} > \{rB1[31:0], rB2[31:0]\}$$

### 64-bit Implementation:

$$SR[F] \leftarrow rA[63:0] > rB[63:0]$$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is a NaN, sets FPCSR[IVF].

## lf.sfgt.s Set Flag if Greater Than Floating-Point Single-Precision lf.sfgt.s

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

### Format:

lf.sfgt.s rA, rB

### Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the first register is greater than the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$SR[F] \leftarrow rA[31:0] > rB[31:0]$

### 64-bit Implementation:

$SR[F] \leftarrow rA[31:0] > rB[31:0]$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is a NaN, sets FPCSR[IVF].

## lf.sfle.d Set Flag if Less or Equal Than Floating-Point Double-Precision

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						reserved					A / A1					B / B1					reserved					opcode 0x1d												
6 bits						5 bits					5 bits					5 bits					3 bits					8 bits												

### Format (32/64-bit):

```
lf.sfle.d rA1, rA2, rB1, rB2
lf.sfle.d rA, rB
```

### Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If the first registers pair is less than or equal to the second registers pair, the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the first register is less than or equal to the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$$SR[F] \leftarrow \{rA1[31:0], rA2[31:0]\} \leq \{rB1[31:0], rB2[31:0]\}$$

### 64-bit Implementation:

$$SR[F] \leftarrow rA[63:0] \leq rB[63:0]$$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is a NaN, sets FPCSR[IVF].

## lf.sfle.s Set Flag if Less or Equal Than Floating-Point Single-Precision lf.sfle.s

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

### Format:

lf.sfle.s rA,rB

### Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the first register is less than or equal to the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$SR[F] \leftarrow rA[31:0] \leq rB[31:0]$

### 64-bit Implementation:

$SR[F] \leftarrow rA[31:0] \leq rB[31:0]$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is a NaN, sets FPCSR[IVF].



## lf.sflt.d      Set Flag if Less Than Floating-Point      lf.sflt.d

### Double-Precision

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32					reserved					A / A1					B / B1					reserved					opcode 0x1c													
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits													

### Format (32/64-bit):

```
lf.sflt.d rA1, rA2, rB1, rB2
lf.sflt.d rA, rB
```

### Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If the first registers pair is less than the second registers pair, the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the first register is less than the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$$SR[F] \leftarrow \{rA1[31:0], rA2[31:0]\} < \{rB1[31:0], rB2[31:0]\}$$

### 64-bit Implementation:

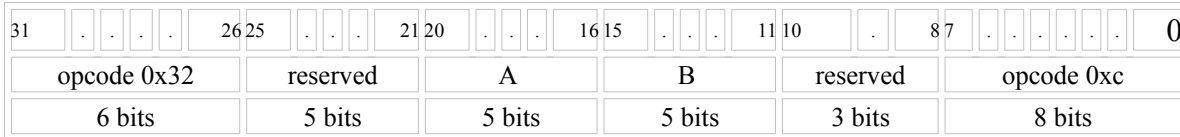
$$SR[F] \leftarrow rA[63:0] < rB[63:0]$$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is a NaN, sets FPCSR[IVF].

## If.sflt.s      Set Flag if Less Than Floating-Point Single-Precision      If.sflt.s



### Format:

`lf.sflt.s rA, rB`

### Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the first register is less than the second register, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$SR[F] \leftarrow rA[31:0] < rB[31:0]$

### 64-bit Implementation:

$SR[F] \leftarrow rA[31:0] < rB[31:0]$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is a NaN, sets FPCSR[IVF].

## lf.sfne.d      Set Flag if Not Equal Floating-Point      lf.sfne.d

### Double-Precision

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32					reserved					A / A1					B / B1					reserved					opcode 0x19													
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits													

### Format (32/64-bit):

```
lf.sfne.d rA1, rA2, rB1, rB2
lf.sfne.d rA, rB
```

### Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If the two registers pairs are not equal, the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the two registers are not equal, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$$SR[F] \leftarrow \{rA1[31:0], rA2[31:0]\} \neq \{rB1[31:0], rB2[31:0]\}$$

### 64-bit Implementation:

$$SR[F] \leftarrow rA[63:0] \neq rB[63:0]$$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is signaling NaN, sets FPCSR[IVF].

## lf.sfne.s      Set Flag if Not Equal Floating-Point Single-Precision      lf.sfne.s

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						reserved					A					B					reserved					opcode 0x9									
6 bits						5 bits					5 bits					5 bits					3 bits					8 bits									

### Format:

lf.sfne.s rA, rB

### Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If the two registers are not equal, the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$SR[F] \leftarrow rA[31:0] \neq rB[31:0]$

### 64-bit Implementation:

$SR[F] \leftarrow rA[31:0] \neq rB[31:0]$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

Floating Point if either input is signaling NaN, sets FPCSR[IVF].

# **lf.sfueq.d      Set Flag if Unordered or Equal      lf.sfueq.d** **Floating-Point Double-precision**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						reserved						A / A1						B / B1						reserved						opcode 0x38							
6 bits						5 bits						5 bits						5 bits						3 bits						8 bits							

## Format (32/64-bit):

```
lf.sfueq.d rA1, rA2, rB1, rB2
lf.sfueq.d rA, rB
```

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If either of the two registers pairs in NaN the compare flag is set; or if the two registers are equal the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; or if the two registers are equal the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

```
SR[F] ← isNaN({rA1[31:0], rA2[31:0]})
        OR isNaN({rB1[31:0], rB2[31:0]})
        OR {rA1[31:0], rA2[31:0]} == {rB1[31:0], rB2[31:0]}
```

## 64-bit Implementation:

```
SR[F] ← isNaN(rA[63:0]) OR isNaN(rB[63:0])
        OR rA[63:0] == rB[63:0]
```

## Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

# If.sfueq.s      Set Flag if Unordered or Equal Floating-Point Single-precision      If.sfueq.s

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						reserved						A					B					reserved					opcode 0x28										
6 bits						5 bits						5 bits					5 bits					3 bits					8 bits										

## Format:

lf.sfueq.s rA, rB

## Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; or if the two registers are equal the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

$$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0]) \\ \text{OR } rA[31:0] == rB[31:0]$$

## 64-bit Implementation:

$$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0]) \\ \text{OR } rA[31:0] == rB[31:0]$$

## Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

# Set Flag if Unordered or Greater If.sfuge.d Than or Equal Floating-Point If.sfuge.d Double-precision

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0x32						reserved						A / A1						B / B1						reserved						opcode 0x3b						
6 bits						5 bits						5 bits						5 bits						3 bits						8 bits						

## Format (32/64-bit):

```
lf.sfuge.d rA1, rA2, rB1, rB2
lf.sfuge.d rA, rB
```

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If either of the two registers pairs in NaN the compare flag is set; or if registers pair {rA1,rA2} is greater than or equal to registers pair {rB1,rB2} the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; or if register rA is greater than or equal to register rB the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

```
SR[F] ← isNaN({rA1[31:0], rA2[31:0]})
        OR isNaN({rB1[31:0], rB2[31:0]})
        OR {rA1[31:0], rA2[31:0]} >= {rB1[31:0], rB2[31:0]}
```

## 64-bit Implementation:

```
SR[F] ← isNaN(rA[63:0]) OR isNaN(rB[63:0])
        OR rA[63:0] >= rB[63:0]
```

## Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

# Set Flag if Unordered or Greater Than or Equal Floating-Point Single-precision

**lf.sfuge.s**      **lf.sfuge.s**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	8	7	.	.	.	.	.	0
opcode 0x32						reserved					A					B					reserved			opcode 0x2b								
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits								

## Format:

`lf.sfuge.s rA, rB`

## Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; or if register rA is greater than or equal to register rB the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

$$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0]) \\ \text{OR } rA[31:0] \geq rB[31:0]$$

## 64-bit Implementation:

$$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0]) \\ \text{OR } rA[31:0] \geq rB[31:0]$$

## Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].



# Set Flag if Unordered or Greater Than Floating-Point Double-precision

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						reserved						A / A1						B / B1						reserved						opcode 0x3a							
6 bits						5 bits						5 bits						5 bits						3 bits						8 bits							

## Format (32/64-bit):

```
lf.sfugt.d rA1, rA2, rB1, rB2
lf.sfugt.d rA, rB
```

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If either of the two registers pairs in NaN the compare flag is set; or if registers pair {rA1,rA2} is greater than registers pair {rB1,rB2} the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; or if register rA is greater than register rB the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

```
SR[F] ← isNaN({rA1[31:0], rA2[31:0]})
        OR isNaN({rB1[31:0], rB2[31:0]})
        OR {rA1[31:0], rA2[31:0]} > {rB1[31:0], rB2[31:0]}
```

## 64-bit Implementation:

```
SR[F] ← isNaN(rA[63:0]) OR isNaN(rB[63:0])
        OR rA[63:0] > rB[63:0]
```

## Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

# Set Flag if Unordered or Greater Than Floating-Point Single- precision

**lf.sfugt.s**      **lf.sfugt.s**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						reserved						A					B					reserved					opcode 0x2a										
6 bits						5 bits						5 bits					5 bits					3 bits					8 bits										

## Format:

lf.sfugt.s rA, rB

## Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; or if register rA is greater than register rB the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

$$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0]) \\ \text{OR } rA[31:0] > rB[31:0]$$

## 64-bit Implementation:

$$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0]) \\ \text{OR } rA[31:0] > rB[31:0]$$

## Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

## Set Flag if Unordered or Less Than If.sfule.d or Equal Floating-Point Double- If.sfule.d precision

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0	
opcode 0x32						reserved						A / A1						B / B1						reserved						opcode 0x3d							
6 bits						5 bits						5 bits						5 bits						3 bits						8 bits							

### Format (32/64-bit):

```
lf.sfule.d rA1, rA2, rB1, rB2
lf.sfule.d rA, rB
```

### Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If either of the two registers pairs in NaN the compare flag is set; or if registers pair {rA1,rA2} is less than or equal to registers pair {rB1,rB2} the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; or if register rA is less than or equal to register rB the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

```
SR[F] ← isNaN({rA1[31:0], rA2[31:0]})
        OR isNaN({rB1[31:0], rB2[31:0]})
        OR {rA1[31:0], rA2[31:0]} <= {rB1[31:0], rB2[31:0]}
```

### 64-bit Implementation:

```
SR[F] ← isNaN(rA[63:0]) OR isNaN(rB[63:0])
        OR rA[63:0] <= rB[63:0]
```

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

## Set Flag if Unordered or Less Than If.sfule.s or Equal Floating-Point Single- If.sfule.s precision

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	8	7	.	.	.	.	.	0
opcode 0x32						reserved					A					B					reserved			opcode 0x2d							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

### Format:

lf.sfuge.s rA, rB

### Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; or if register rA is less than or equal to register rB the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0]) \\ \text{OR } rA[31:0] \leq rB[31:0]$$

### 64-bit Implementation:

$$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0]) \\ \text{OR } rA[31:0] \leq rB[31:0]$$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

## lf.sfult.d    Set Flag if Unordered or Less Than    lf.sfult.d

### Floating-Point Double-precision

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						reserved						A / A1						B / B1						reserved						opcode 0x3c							
6 bits						5 bits						5 bits						5 bits						3 bits						8 bits							

### Format (32/64-bit):

```
lf.sfult.d rA1, rA2, rB1, rB2
lf.sfult.d rA, rB
```

### Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If either of the two registers pairs in NaN the compare flag is set; or if registers pair {rA1,rA2} is less than registers pair {rB1,rB2} the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; or if register rA is less than register rB the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

```
SR[F] ← isNaN({rA1[31:0], rA2[31:0]})
        OR isNaN({rB1[31:0], rB2[31:0]})
        OR {rA1[31:0], rA2[31:0]} < {rB1[31:0], rB2[31:0]}
```

### 64-bit Implementation:

```
SR[F] ← isNaN(rA[63:0]) OR isNaN(rB[63:0])
        OR rA[63:0] < rB[63:0]
```

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

# If.sfult.s      Set Flag if Unordered or Less Than      If.sfult.s

## Floating-Point Single-precision

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						reserved						A						B						reserved						opcode 0x2c							
6 bits						5 bits						5 bits						5 bits						3 bits						8 bits							

### Format:

lf.sfult.s rA, rB

### Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; or if register rA is less than register rB the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

$$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0]) \\ \text{OR } rA[31:0] < rB[31:0]$$

### 64-bit Implementation:

$$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0]) \\ \text{OR } rA[31:0] < rB[31:0]$$

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

## lf.sfun.d      Set Flag if Unordered Floating-Point Double-precision      lf.sfun.d

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0x32						reserved						A / A1						B / B1						reserved						opcode 0x3e							
6 bits						5 bits						5 bits						5 bits						3 bits						8 bits							

### Format (32/64-bit):

```
lf.sfun.d rA1, rA2, rB1, rB2
lf.sfun.d rA, rB
```

### Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} and the contents of general-purpose registers pair {rB1,rB2} are compared. If either of the two registers pairs in NaN the compare flag is set; otherwise the compare flag is cleared. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; otherwise the compare flag is cleared.

### 32-bit Implementation:

```
SR[F] ← isNaN({rA1[31:0], rA2[31:0]})
        OR isNaN({rB1[31:0], rB2[31:0]})
```

### 64-bit Implementation:

```
SR[F] ← isNaN(rA[63:0]) OR isNaN(rB[63:0])
```

### Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].

# **lf.sfun.s**      **Set Flag if Unordered Floating-Point Single-precision**      **lf.sfun.s**

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.
opcode 0x32						reserved					A					B					reserved					opcode 0x2b										
6 bits						5 bits					5 bits					5 bits					3 bits					8 bits										

## Format:

lf.sfun.s rA,rB

## Description:

The contents of general-purpose register rA and the contents of general-purpose register rB are compared. If either of the two registers in NaN the compare flag is set; otherwise the compare flag is cleared.

## 32-bit Implementation:

$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0])$

## 64-bit Implementation:

$SR[F] \leftarrow \text{isNaN}(rA[31:0]) \text{ OR } \text{isNaN}(rB[31:0])$

## Exceptions:

Floating Point if either input is infinity, sets FPCSR[INF].



# **lf.stod.d      Convert Single-precision Floating-Point Number To Double-precision      lf.stod.d**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	8	7	.	.	.	.	.	0
opcode 0x32						D / D1					A					reserved					reserved					opcode 0x34					
6 bits						5 bits					5 bits					5 bits					3 bits					8 bits					

## Format (32/64-bit):

```
lf.stod.d rD1, rD2, rA
lf.stod.d rD, rA
```

## Description:

On 32-bit machine the contents of general-purpose register rA are converted from Single-precision to Double-precision. The results are stored in general-purpose registers pair {rD1,rD2}. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rA are converted from Single-precision to Double-precision. The results are stored in general-purpose register rD.

## 32-bit Implementation:

```
{rD1[31:0], rD2[31:0]} ← double(rA[31:0])
```

## 64-bit Implementation:

```
rD[63:0] ← double(rA[31:0])
```

## Exceptions:

None

# **lf.sub.d      Subtract Floating-Point Double-Precision      lf.sub.d**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	0
opcode 0x32						D / D1					A / A1					B / B1					reserved			opcode 0x11						
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits						

## Format (32/64-bit):

```
lf.sub.d rD1, rD2, rA1, rA2, rB1, rB2
```

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

## Description:

On 32-bit machine the contents of general-purpose registers pair {rA1,rA2} are subtracted from the contents of general-purpose registers pair {rB1,rB2} to form the result. The result is placed into general-purpose registers pair {rD1,rD2}. See chapter ORFPX64A32 for registers pairing details.

On 64-bit machine the contents of general-purpose register rB are subtracted from the contents of general-purpose register rA to form the result. The result is placed into general-purpose register rD.

## 32-bit Implementation:

$$\{rD1[31:0], rD2[31:0]\} \leftarrow \{rA1[31:0], rA2[31:0]\} - \{rB1[31:0], rB2[31:0]\}$$

## 64-bit Implementation:

$$rD[63:0] \leftarrow rA[63:0] - rB[63:0]$$

## Exceptions:

Floating Point

## lf.sub.s Subtract Floating-Point Single-Precision lf.sub.s

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0x32					D					A					B					reserved					opcode 0x1									
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits									

### Format:

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

### Description:

The contents of general-purpose register rB are subtracted from the contents of general-purpose register rA to form the result. The result is placed into general-purpose register rD. On 64-bit machine the result should be NaN-boxed.

### 32-bit Implementation:

$$rD[31:0] \leftarrow rA[31:0] - rB[31:0]$$

### 64-bit Implementation:

$$rD[31:0] \leftarrow rA[31:0] - rB[31:0]$$

$$rD[63:32] \leftarrow 0xFFFFFFFF$$

### Exceptions:

Floating Point

## 5.5 ORFPX64A32

Support for Double-precision floating point operations on 32-bit hardware is provided by performing operations using 32-bit register pairs.

When expressed in assembler register pairs are explicitly written, for example an add instruction may be written as `lf.add.d rD1, rD2, rA1, rA2, rB1, rB2`. The first registers `rD1`, `rA1` and `rB1` are encoded in the instruction directly. The second registers `rD2`, `rA2` and `rB2` are encoded via the register offset bit mask stored in instruction bits 10,9 and 8. The reg offset bit mask indicates if the second register is offset from the first by 1 or 2 as per the following:

- `bit[10]` – if set indicates `rD2` is `rD1+2`, otherwise `rD2` is `rD1+1`
- `bit[9]` – if set indicates `rA2` is `rA1+2`, otherwise `rA2` is `rA1+1`
- `bit[8]` – if set indicates `rB2` is `rB1+2`, otherwise `rB2` is `rB1+1`

On 64-bit machines these shall be set to 0.

The resulting 64-bit register pair, expressed `{rA1[31:0], rA2[31:0]}`, represents a big-endian encoded register. That is, the most significant bits come first, in this case in `rA1`.

The following is a list of various assembler encodings and semantics:

### Comparison Operations:

```
ls.sf*.d rA1, rA2, rB1, rB2
SR[F] ← {rA1[31:0], rA2[31:0]} CMP {rB1[31:0], rB2[31:0]}
```

### Binary Operation:

```
ls.*.d rD1, rD2, rA1, rA2, rB1, rB2
{rD1[31:0], rD2[31:0]} ← {rA1[31:0], rA2[31:0]}
                        OP {rB1[31:0], rB2[31:0]}
```

### Unary Operation:

```
ls.*.d rD1, rD2, rA1, rA2
{rD1[31:0], rD2[31:0]} ← OP({rA1[31:0], rA2[31:0]})
```

### Single-Precision to Double-Precision:

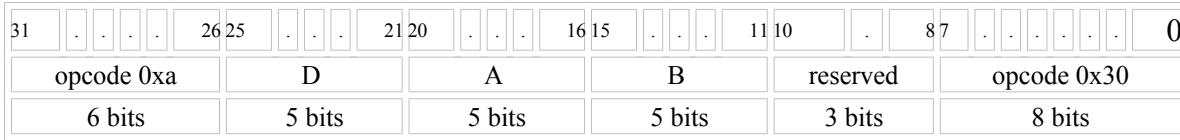
```
ls.stod.d rD1, rD2, rA
{rD1[31:0], rD2[31:0]} ← stod(rA[31:0])
```

### Double-Precision to Single-Precision:

```
ls.dtos.d rD, rA1, rA2
rD[31:0] ← stod({rA1[31:0], rA2[31:0]})
```

## 5.6 ORVDX64

### lv.add.b    Vector Byte Elements Add Signed    lv.add.b



#### Format:

lv.add.b rD, rA, rB

#### Description:

The byte elements of general-purpose register rA are added to the byte elements of general-purpose register rB to form the result elements. The result elements are placed into general-purpose register rD.

#### 32-bit Implementation:

N/A

#### 64-bit Implementation:

```

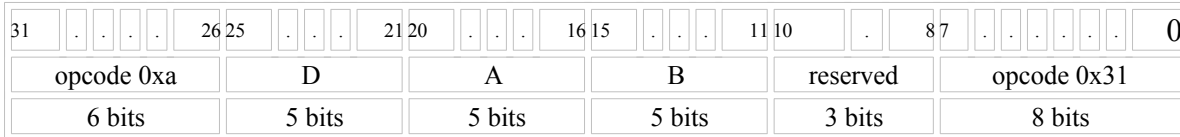
rD[7:0]   ← rA[7:0]   + rB[7:0]
rD[15:8]  ← rA[15:8]  + rB[15:8]
rD[23:16] ← rA[23:16] + rB[23:16]
rD[31:24] ← rA[31:24] + rB[31:24]
rD[39:32] ← rA[39:32] + rB[39:32]
rD[47:40] ← rA[47:40] + rB[47:40]
rD[55:48] ← rA[55:48] + rB[55:48]
rD[63:56] ← rA[63:56] + rB[63:56]

```

#### Exceptions:

None

# lv.add.h **Vector Half-Word Elements Add Signed** lv.add.h



## Format:

lv.add.h rD, rA, rB

## Description:

The half-word elements of general-purpose register rA are added to the half-word elements of general-purpose register rB to form the result elements. The result elements are placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

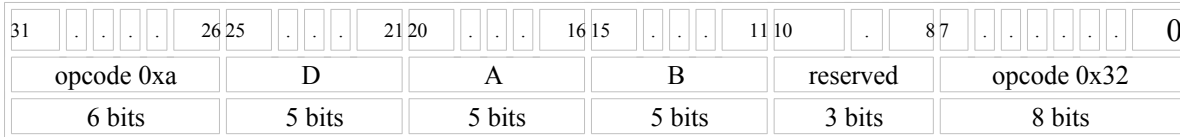
rD[15:0]  ← rA[15:0]  + rB[15:0]
rD[31:16] ← rA[31:16] + rB[31:16]
rD[47:32] ← rA[47:32] + rB[47:32]
rD[63:48] ← rA[63:48] + rB[63:48]

```

## Exceptions:

None

## lv.adds.b Vector Byte Elements Add Signed Saturated lv.adds.b



### Format:

lv.adds.b rD, rA, rB

### Description:

The byte elements of general-purpose register rA are added to the byte elements of general-purpose register rB to form the result elements. If the result exceeds the min/max value for the destination data type, it is saturated to the min/max value and placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← sat8s(rA[7:0]    + rB[7:0])
rD[15:8]   ← sat8s(rA[15:8]   + rB[15:8])
rD[23:16]  ← sat8s(rA[23:16]  + rB[23:16])
rD[31:24]  ← sat8s(rA[31:24]  + rB[31:24])
rD[39:32]  ← sat8s(rA[39:32]  + rB[39:32])
rD[47:40]  ← sat8s(rA[47:40]  + rB[47:40])
rD[55:48]  ← sat8s(rA[55:48]  + rB[55:48])
rD[63:56]  ← sat8s(rA[63:56]  + rB[63:56])

```

### Exceptions:

None

# lv.adds.h Vector Half-Word Elements Add Signed Saturated lv.adds.h

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	8	7	.	.	.	.	.	0
opcode 0xa						D					A					B					reserved			opcode 0x33							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

## Format:

lv.adds.h rD, rA, rB

## Description:

The half-word elements of general-purpose register rA are added to the half-word elements of general-purpose register rB to form the result elements. If the result exceeds the min/max value for the destination data type, it is saturated to the min/max value and placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[15:0]  ← sat16s(rA[15:0]  + rB[15:0])
rD[31:16] ← sat16s(rA[31:16] + rB[31:16])
rD[47:32] ← sat16s(rA[47:32] + rB[47:32])
rD[63:48] ← sat16s(rA[63:48] + rB[63:48])

```

## Exceptions:

None



# lv.addu.b      Vector Byte Elements Add      lv.addu.b

## Unsigned

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x34							
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits							

### Format:

lv.addu.b rD, rA, rB

### Description:

The unsigned byte elements of general-purpose register rA are added to the unsigned byte elements of general-purpose register rB to form the result elements. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

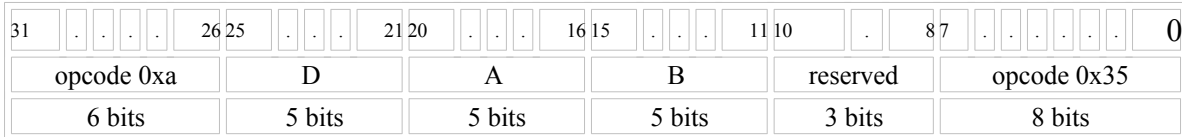
rD[7:0]    ← rA[7:0]    + rB[7:0]
rD[15:8]   ← rA[15:8]   + rB[15:8]
rD[23:16]  ← rA[23:16]  + rB[23:16]
rD[31:24]  ← rA[31:24]  + rB[31:24]
rD[39:32]  ← rA[39:32]  + rB[39:32]
rD[47:40]  ← rA[47:40]  + rB[47:40]
rD[55:48]  ← rA[55:48]  + rB[55:48]
rD[63:56]  ← rA[63:56]  + rB[63:56]

```

### Exceptions:

None

# lv.addu.h Vector Half-Word Elements Add Unsigned lv.addu.h



## Format:

lv.addu.h rD, rA, rB

## Description:

The unsigned half-word elements of general-purpose register rA are added to the unsigned half-word elements of general-purpose register rB to form the result elements. The result elements are placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[15:0]  ← rA[15:0]  + rB[15:0]
rD[31:16] ← rA[31:16] + rB[31:16]
rD[47:32] ← rA[47:32] + rB[47:32]
rD[63:48] ← rA[63:48] + rB[63:48]

```

## Exceptions:

None

# lv.addus.b      Vector Byte Elements Add      lv.addus.b

## Unsigned Saturated

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x36								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

### Format:

lv.addus.b rD, rA, rB

### Description:

The unsigned byte elements of general-purpose register rA are added to the unsigned byte elements of general-purpose register rB to form the result elements. If the result exceeds the min/max value for the destination data type, it is saturated to the min/max value and placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← sat8u(rA[7:0]    + rB[7:0])
rD[15:8]   ← sat8u(rA[15:8]   + rB[15:8])
rD[23:16]  ← sat8u(rA[23:16]  + rB[23:16])
rD[31:24]  ← sat8u(rA[31:24]  + rB[31:24])
rD[39:32]  ← sat8u(rA[39:32]  + rB[39:32])
rD[47:40]  ← sat8u(rA[47:40]  + rB[47:40])
rD[55:48]  ← sat8u(rA[55:48]  + rB[55:48])
rD[63:56]  ← sat8u(rA[63:56]  + rB[63:56])

```

### Exceptions:

None

# lv.addus.h Vector Half-Word Elements Add Unsigned Saturated lv.addus.h

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x37						
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits						

## Format:

lv.addus.h rD, rA, rB

## Description:

The unsigned half-word elements of general-purpose register rA are added to the unsigned half-word elements of general-purpose register rB to form the result elements. If the result exceeds the min/max value for the destination data type, it is saturated to the min/max value and placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

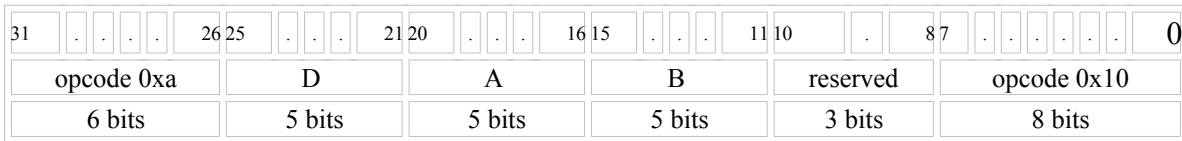
rD[15:0]  ← sat16s(rA[15:0]  + rB[15:0])
rD[31:16] ← sat16s(rA[31:16] + rB[31:16])
rD[47:32] ← sat16s(rA[47:32] + rB[47:32])
rD[63:48] ← sat16s(rA[63:48] + rB[63:48])

```

## Exceptions:

None

## lv.all\_eq.b Vector Byte Elements All Equal lv.all\_eq.b



### Format:

```
lv.all_eq.b rD, rA, rB
```

### Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if all corresponding elements are equal; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

### 32-bit Implementation:

N/A

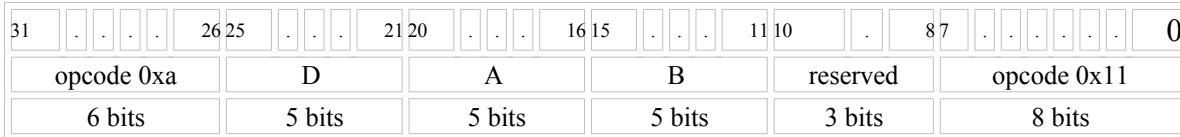
### 64-bit Implementation:

```
flag ← rA[7:0]    == rB[7:0]    &&
      rA[15:8]   == rB[15:8]    &&
      rA[23:16]  == rB[23:16]   &&
      rA[31:24]  == rB[31:24]   &&
      rA[39:32]  == rB[39:32]   &&
      rA[47:40]  == rB[47:40]   &&
      rA[55:48]  == rB[55:48]   &&
      rA[63:56]  == rB[63:56]
rD[63:0] ← repl(flag)
```

### Exceptions:

None

# lv.all\_eq.h Vector Half-Word Elements All Equal



## Format:

lv.all\_eq.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if all corresponding elements are equal; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```
flag      ← rA[15:0]  == rB[15:0]  &&
           rA[31:16] == rB[31:16] &&
           rA[47:32] == rB[47:32] &&
           rA[63:48] == rB[63:48]
rD[63:0] ← repl(flag)
```

## Exceptions:

None

# lv.all\_ge.b Vector Byte Elements All Greater Than or Equal To

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x12							
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits							

## Format:

lv.all\_ge.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if all elements of rA are greater than or equal to the elements of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[7:0]    >= rB[7:0]  &&
           rA[15:8]    >= rB[15:8]  &&
           rA[23:16]   >= rB[23:16] &&
           rA[31:24]   >= rB[31:24] &&
           rA[39:32]   >= rB[39:32] &&
           rA[47:40]   >= rB[47:40] &&
           rA[55:48]   >= rB[55:48] &&
           rA[63:56]   >= rB[63:56]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.all\_ge.h Vector Half-Word Elements All Greater Than or Equal To lv.all\_ge.h

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x13								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

## Format:

lv.all\_ge.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if all elements of rA are greater than or equal to the elements of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```
flag      ← rA[15:0]  >= rB[15:0]  &&
           rA[31:16] >= rB[31:16] &&
           rA[47:32] >= rB[47:32] &&
           rA[63:48] >= rB[63:48]
rD[63:0] ← repl(flag)
```

## Exceptions:

None

Instruction Class  
ORVDX64 I



# lv.all\_gt.b Vector Byte Elements All Greater Than lv.all\_gt.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0	
opcode 0xa					D					A					B					reserved					opcode 0x14									
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits									

## Format:

lv.all\_gt.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if all elements of rA are greater than the elements of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[7:0]    > rB[7:0]  &&
           rA[15:8]    > rB[15:8]  &&
           rA[23:16]   > rB[23:16]  &&
           rA[31:24]   > rB[31:24]  &&
           rA[39:32]   > rB[39:32]  &&
           rA[47:40]   > rB[47:40]  &&
           rA[55:48]   > rB[55:48]  &&
           rA[63:56]   > rB[63:56]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.all\_gt.h Vector Half-Word Elements All Greater Than lv.all\_gt.h

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x15							
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits							

## Format:

lv.all\_gt.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if all elements of rA are greater than the elements of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[15:0]  > rB[15:0]  &&
           rA[31:16] > rB[31:16] &&
           rA[47:32] > rB[47:32] &&
           rA[63:48] > rB[63:48]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

# lv.all\_le.b      Vector Byte Elements All Less Than or Equal To      lv.all\_le.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x16								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

## Format:

lv.all\_le.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if all elements of rA are less than or equal to the elements of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[7:0]    <= rB[7:0]  &&
           rA[15:8]    <= rB[15:8]  &&
           rA[23:16]   <= rB[23:16]  &&
           rA[31:24]   <= rB[31:24]  &&
           rA[39:32]   <= rB[39:32]  &&
           rA[47:40]   <= rB[47:40]  &&
           rA[55:48]   <= rB[55:48]  &&
           rA[63:56]   <= rB[63:56]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.all\_le.h Vector Half-Word Elements All Less Than or Equal To lv.all\_le.h

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

## Format:

lv.all\_le.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if all elements of rA are less than or equal to the elements of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

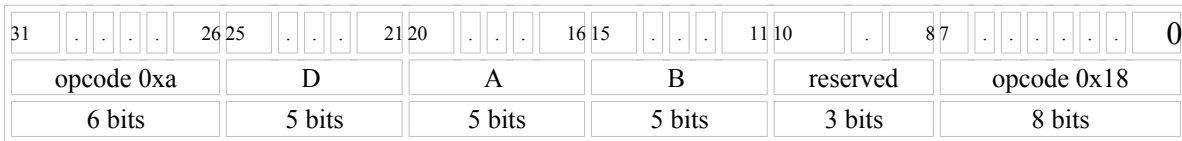
## 64-bit Implementation:

```
flag      ← rA[15:0]  <= rB[15:0]  &&
           rA[31:16] <= rB[31:16] &&
           rA[47:32] <= rB[47:32] &&
           rA[63:48] <= rB[63:48]
rD[63:0] ← repl(flag)
```

## Exceptions:

None

## lv.all\_lt.b Vector Byte Elements All Less Than lv.all\_lt.b



### Format:

```
lv.all_lt.b rD, rA, rB
```

### Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if all elements of rA are less than the elements of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

### 32-bit Implementation:

N/A

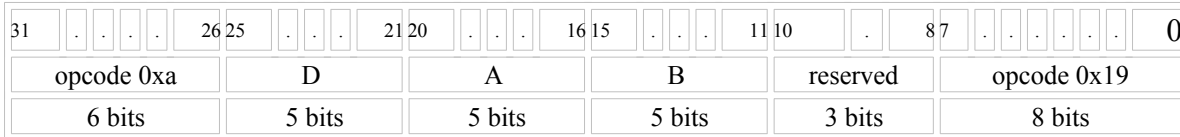
### 64-bit Implementation:

```
flag      ← rA[7:0]    < rB[7:0]  &&
           rA[15:8]   < rB[15:8]  &&
           rA[23:16]  < rB[23:16]  &&
           rA[31:24]  < rB[31:24]  &&
           rA[39:32]  < rB[39:32]  &&
           rA[47:40]  < rB[47:40]  &&
           rA[55:48]  < rB[55:48]  &&
           rA[63:56]  < rB[63:56]
rD[63:0] ← repl(flag)
```

### Exceptions:

None

# lv.all\_lt.h Vector Half-Word Elements All Less Than



## Format:

lv.all\_lt.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if all elements of rA are less than the elements of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[15:0]  < rB[15:0]  &&
           rA[31:16] < rB[31:16] &&
           rA[47:32] < rB[47:32] &&
           rA[63:48] < rB[63:48]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.all\_ne.b      Vector Byte Elements All Not Equal      lv.all\_ne.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x1a									
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits									

## Format:

lv.all\_ne.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if all corresponding elements are not equal; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[7:0]    != rB[7:0]  &&
           rA[15:8]    != rB[15:8]  &&
           rA[23:16]   != rB[23:16]  &&
           rA[31:24]   != rB[31:24]  &&
           rA[39:32]   != rB[39:32]  &&
           rA[47:40]   != rB[47:40]  &&
           rA[55:48]   != rB[55:48]  &&
           rA[63:56]   != rB[63:56]
rD[63:0] ← repl(flag)

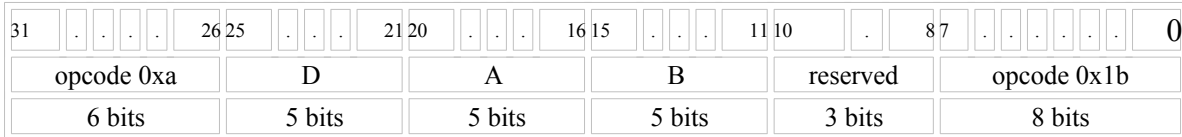
```

## Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.all\_ne.h Vector Half-Word Elements All Not Equal



## Format:

lv.all\_ne.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if all corresponding elements are not equal; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

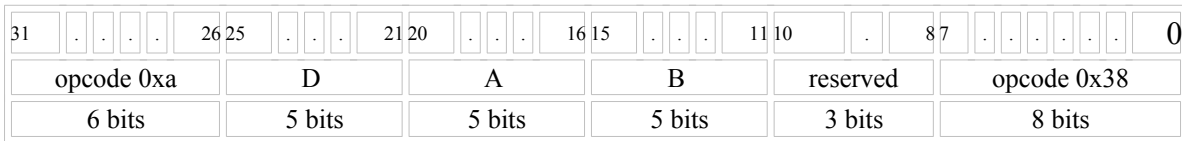
## 64-bit Implementation:

```
flag      ← rA[15:0]  != rB[15:0] &&
           rA[31:16] != rB[31:16] &&
           rA[47:32] != rB[47:32] &&
           rA[63:48] != rB[63:48]
rD[63:0] ← repl(flag)
```

## Exceptions:

None



**lv.and****Vector And****lv.and****Format:**

$$\text{lv.and } rD, rA, rB$$
**Description:**

The contents of general-purpose register rA are combined with the contents of general-purpose register rB in a bit-wise logical AND operation. The result is placed into general-purpose register rD.

**32-bit Implementation:**

N/A

**64-bit Implementation:**

$$rD[63:0] \leftarrow rA[63:0] \text{ AND } rB[63:0]$$
**Exceptions:**

None

# lv.any\_eq.b      Vector Byte Elements Any Equal      lv.any\_eq.b

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x20												
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits												

## Format:

lv.any\_eq.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if any two corresponding elements are equal; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

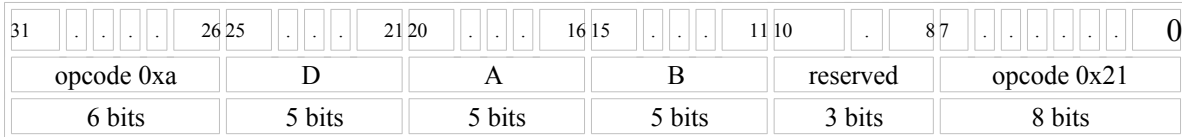
flag      ← rA[7:0]    == rB[7:0] ||
           rA[15:8]    == rB[15:8] ||
           rA[23:16]   == rB[23:16] ||
           rA[31:24]   == rB[31:24] ||
           rA[39:32]   == rB[39:32] ||
           rA[47:40]   == rB[47:40] ||
           rA[55:48]   == rB[55:48] ||
           rA[63:56]   == rB[63:56]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

# lv.any\_eq.h Vector Half-Word Elements Any Equal



## Format:

lv.any\_eq.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if any two corresponding elements are equal; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[15:0]  == rB[15:0] ||
           rA[31:16] == rB[31:16] ||
           rA[47:32] == rB[47:32] ||
           rA[63:48] == rB[63:48]
rD[63:0] ← repl(flag)

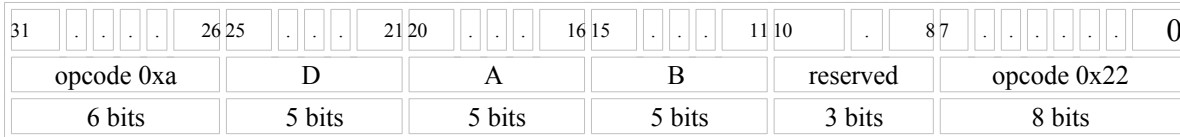
```

## Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.any\_ge.b      Vector Byte Elements Any Greater Than or Equal To      lv.any\_ge.b



## Format:

lv.any\_ge.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if any element of rA is greater than or equal to the corresponding element of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[7:0]    >= rB[7:0] ||
           rA[15:8]    >= rB[15:8] ||
           rA[23:16]   >= rB[23:16] ||
           rA[31:24]   >= rB[31:24] ||
           rA[39:32]   >= rB[39:32] ||
           rA[47:40]   >= rB[47:40] ||
           rA[55:48]   >= rB[55:48] ||
           rA[63:56]   >= rB[63:56]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

# lv.any\_ge.h Vector Half-Word Elements Any Greater Than or Equal To lv.any\_ge.h

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x23								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

## Format:

lv.any\_ge.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if any element of rA is greater than or equal to the corresponding element of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```
flag      ← rA[15:0]  >= rB[15:0] ||
           rA[31:16] >= rB[31:16] ||
           rA[47:32] >= rB[47:32] ||
           rA[63:48] >= rB[63:48]
rD[63:0] ← repl(flag)
```

## Exceptions:

None

# lv.any\_gt.b      Vector Byte Elements Any Greater Than      lv.any\_gt.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x24								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

## Format:

lv.any\_gt.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if any element of rA is greater than the corresponding element of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

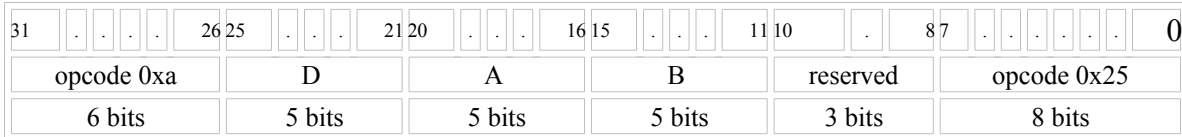
flag      ← rA[7:0]    > rB[7:0] ||
           rA[15:8]    > rB[15:8] ||
           rA[23:16]   > rB[23:16] ||
           rA[31:24]   > rB[31:24] ||
           rA[39:32]   > rB[39:32] ||
           rA[47:40]   > rB[47:40] ||
           rA[55:48]   > rB[55:48] ||
           rA[63:56]   > rB[63:56]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

# lv.any\_gt.h Vector Half-Word Elements Any Greater Than



## Format:

lv.any\_gt.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if any element of rA is greater than the corresponding element of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[15:0]  > rB[15:0] ||
           rA[31:16] > rB[31:16] ||
           rA[47:32] > rB[47:32] ||
           rA[63:48] > rB[63:48]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

# lv.any\_le.b Vector Byte Elements Any Less Than or Equal To lv.any\_le.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x26								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

## Format:

lv.any\_le.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if any element of rA is less than or equal to the corresponding element of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[7:0]    <= rB[7:0] ||
           rA[15:8]    <= rB[15:8] ||
           rA[23:16]   <= rB[23:16] ||
           rA[31:24]   <= rB[31:24] ||
           rA[39:32]   <= rB[39:32] ||
           rA[47:40]   <= rB[47:40] ||
           rA[55:48]   <= rB[55:48] ||
           rA[63:56]   <= rB[63:56]
rD[63:0] ← repl(flag)

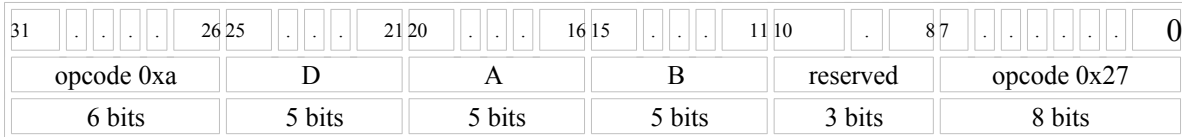
```

## Exceptions:

None



# lv.any\_le.h Vector Half-Word Elements Any Less Than or Equal To



## Format:

lv.any\_le.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if any element of rA is less than or equal to the corresponding element of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[15:0]  <= rB[15:0] ||
           rA[31:16] <= rB[31:16] ||
           rA[47:32] <= rB[47:32] ||
           rA[63:48] <= rB[63:48]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

# lv.any\_lt.b Vector Byte Elements Any Less Than lv.any\_lt.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x28								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

## Format:

lv.any\_lt.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if any element of rA is less than the corresponding element of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

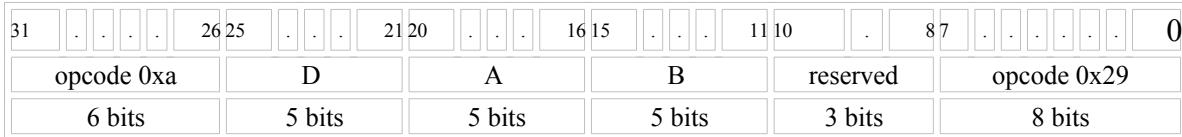
flag      ← rA[7:0]    < rB[7:0] ||
           rA[15:8]    < rB[15:8] ||
           rA[23:16]   < rB[23:16] ||
           rA[31:24]   < rB[31:24] ||
           rA[39:32]   < rB[39:32] ||
           rA[47:40]   < rB[47:40] ||
           rA[55:48]   < rB[55:48] ||
           rA[63:56]   < rB[63:56]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

## lv.any\_lt.h Vector Half-Word Elements Any Less Than



### Format:

lv.any\_lt.h rD, rA, rB

### Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if any element of rA is less than the corresponding element of rB; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

flag      ← rA[15:0]  < rB[15:0] ||
           rA[31:16] < rB[31:16] ||
           rA[47:32] < rB[47:32] ||
           rA[63:48] < rB[63:48]
rD[63:0] ← repl(flag)

```

### Exceptions:

None

# lv.any\_ne.b Vector Byte Elements Any Not Equal lv.any\_ne.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x2a									
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits									

## Format:

lv.any\_ne.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. The compare flag is set if any two corresponding elements are not equal; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

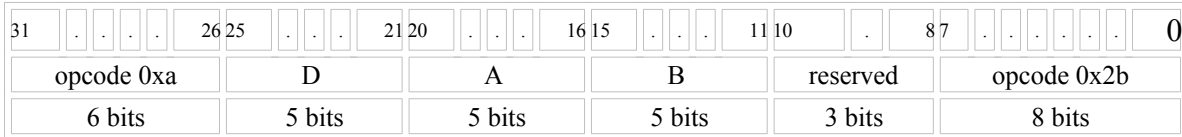
flag      ← rA[7:0]    != rB[7:0] ||
           rA[15:8]    != rB[15:8] ||
           rA[23:16]   != rB[23:16] ||
           rA[31:24]   != rB[31:24] ||
           rA[39:32]   != rB[39:32] ||
           rA[47:40]   != rB[47:40] ||
           rA[55:48]   != rB[55:48] ||
           rA[63:56]   != rB[63:56]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

# lv.any\_ne.h Vector Half-Word Elements Any Not Equal lv.any\_ne.h



## Format:

lv.any\_ne.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. The compare flag is set if any two corresponding elements are not equal; otherwise the compare flag is cleared. The compare flag is replicated into all bit positions of general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

flag      ← rA[15:0]  != rB[15:0] ||
           rA[31:16] != rB[31:16] ||
           rA[47:32] != rB[47:32] ||
           rA[63:48] != rB[63:48]
rD[63:0] ← repl(flag)

```

## Exceptions:

None

Instruction Class  
ORVDX64 I

## lv.avg.b      Vector Byte Elements Average      lv.avg.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x39								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

### Format:

lv.avg.b rD, rA, rB

### Description:

The byte elements of general-purpose register rA are added to the byte elements of general-purpose register rB, and the sum is shifted right by one to form the result elements. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]   ← (rA[7:0]   + rB[7:0])   >> 1
rD[15:8]  ← (rA[15:8]  + rB[15:8])  >> 1
rD[23:16] ← (rA[23:16] + rB[23:16]) >> 1
rD[31:24] ← (rA[31:24] + rB[31:24]) >> 1
rD[39:32] ← (rA[39:32] + rB[39:32]) >> 1
rD[47:40] ← (rA[47:40] + rB[47:40]) >> 1
rD[55:48] ← (rA[55:48] + rB[55:48]) >> 1
rD[63:56] ← (rA[63:56] + rB[63:56]) >> 1

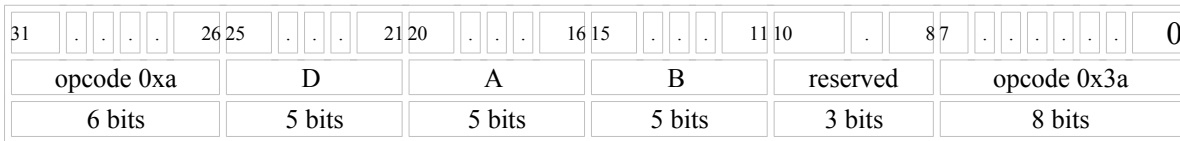
```

### Exceptions:

None

Instruction Class  
ORV DX64 I

## lv.avg.h Vector Half-Word Elements Average lv.avg.h



### Format:

lv.avg.h rD, rA, rB

### Description:

The half-word elements of general-purpose register rA are added to the half-word elements of general-purpose register rB, and the sum is shifted right by one to form the result elements. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[15:0]  ← (rA[15:0]  + rB[15:0])  >> 1
rD[31:16] ← (rA[31:16] + rB[31:16]) >> 1
rD[47:32] ← (rA[47:32] + rB[47:32]) >> 1
rD[63:48] ← (rA[63:48] + rB[63:48]) >> 1

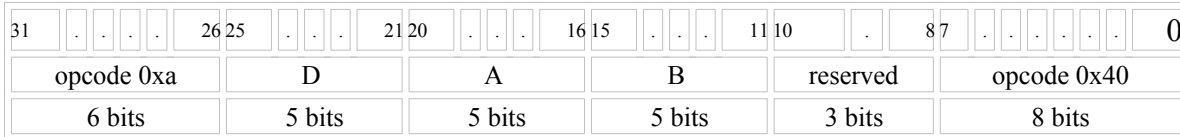
```

### Exceptions:

None

**lv.cmp\_eq.b**

## Vector Byte Elements Compare Equal

**lv.cmp\_eq.b**

### Format:

```
lv.cmp_eq.b rD, rA, rB
```

### Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the two corresponding compared elements are equal; otherwise the element bits are cleared.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← repl (rA[7:0]    == rB[7:0])
rD[15:8]   ← repl (rA[15:8]   == rB[15:8])
rD[23:16]  ← repl (rA[23:16]  == rB[23:16])
rD[31:24]  ← repl (rA[31:24]  == rB[31:24])
rD[39:32]  ← repl (rA[39:32]  == rB[39:32])
rD[47:40]  ← repl (rA[47:40]  == rB[47:40])
rD[55:48]  ← repl (rA[55:48]  == rB[55:48])
rD[63:56]  ← repl (rA[63:56]  == rB[63:56])

```

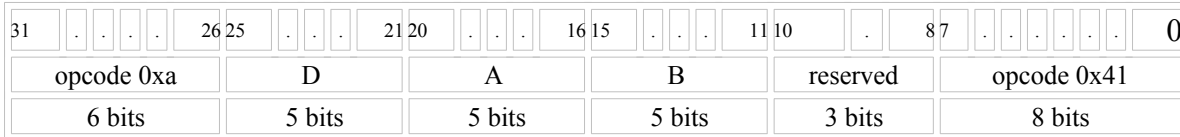
### Exceptions:

None

Instruction Class  
ORV DX64 I



# lv.cmp\_eq.h Vector Half-Word Elements Compare Equal



## Format:

lv.cmp\_eq.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the two corresponding compared elements are equal; otherwise the element bits are cleared.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[15:0]  ← repl(rA[15:0]  == rB[15:0])
rD[31:16] ← repl(rA[31:16] == rB[31:16])
rD[47:32] ← repl(rA[47:32] == rB[47:32])
rD[63:48] ← repl(rA[63:48] == rB[63:48])

```

## Exceptions:

None

## Vector Byte Elements

### lv.cmp\_ge.b    Compare Greater Than or    lv.cmp\_ge.b

### Equal To

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	7	.	.	.	.	.	0
opcode 0xa						D					A					B					reserved			opcode 0x42							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

#### Format:

lv.cmp\_ge.b rD, rA, rB

#### Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the element in rA is greater than or equal to the element in rB; otherwise the element bits are cleared.

#### 32-bit Implementation:

N/A

#### 64-bit Implementation:

```

rD[7:0]    ← repl (rA[7:0]    >= rB[7:0])
rD[15:8]   ← repl (rA[15:8]   >= rB[15:8])
rD[23:16]  ← repl (rA[23:16]  >= rB[23:16])
rD[31:24]  ← repl (rA[31:24]  >= rB[31:24])
rD[39:32]  ← repl (rA[39:32]  >= rB[39:32])
rD[47:40]  ← repl (rA[47:40]  >= rB[47:40])
rD[55:48]  ← repl (rA[55:48]  >= rB[55:48])
rD[63:56]  ← repl (rA[63:56]  >= rB[63:56])

```

#### Exceptions:

None

Instruction Class  
ORVDX64 I

## Vector Half-Word Elements

### lv.cmp\_ge.h    Compare Greater Than or    lv.cmp\_ge.h Equal To

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	7	.	.	.	.	.	0
opcode 0xa						D					A					B					reserved			opcode 0x43							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

#### Format:

lv.cmp\_ge.h rD, rA, rB

#### Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the element in rA is greater than or equal to the element in rB; otherwise the element bits are cleared.

#### 32-bit Implementation:

N/A

#### 64-bit Implementation:

```

rD[15:0]  ← repl(rA[15:0]  >= rB[15:0])
rD[31:16] ← repl(rA[31:16] >= rB[31:16])
rD[47:32] ← repl(rA[47:32] >= rB[47:32])
rD[63:48] ← repl(rA[63:48] >= rB[63:48])

```

#### Exceptions:

None

Instruction Class  
ORV DX64 I

# lv.cmp\_gt.b Vector Byte Elements Compare Greater Than lv.cmp\_gt.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x44								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

## Format:

lv.cmp\_gt.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the element in rA is greater than the element in rB; otherwise the element bits are cleared.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

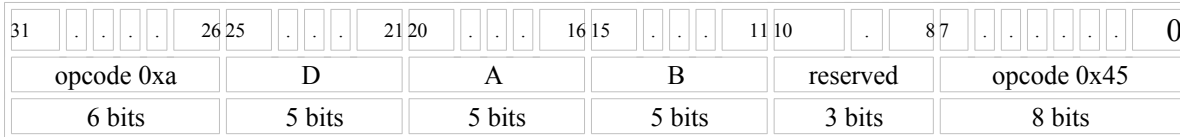
rD[7:0]    ← repl (rA[7:0]    > rB[7:0])
rD[15:8]   ← repl (rA[15:8]   > rB[15:8])
rD[23:16]  ← repl (rA[23:16]  > rB[23:16])
rD[31:24]  ← repl (rA[31:24]  > rB[31:24])
rD[39:32]  ← repl (rA[39:32]  > rB[39:32])
rD[47:40]  ← repl (rA[47:40]  > rB[47:40])
rD[55:48]  ← repl (rA[55:48]  > rB[55:48])
rD[63:56]  ← repl (rA[63:56]  > rB[63:56])

```

## Exceptions:

None

# lv.cmp\_gt.h Vector Half-Word Elements Compare Greater Than lv.cmp\_gt.h



## Format:

lv.cmp\_gt.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the element in rA is greater than the element in rB; otherwise the element bits are cleared.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[15:0] ← repl(rA[15:0] > rB[15:0])
rD[31:16] ← repl(rA[31:16] > rB[31:16])
rD[47:32] ← repl(rA[47:32] > rB[47:32])
rD[63:48] ← repl(rA[63:48] > rB[63:48])

```

## Exceptions:

None

# lv.cmp\_le.b Vector Byte Elements Compare Less Than or Equal To lv.cmp\_le.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x46								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

## Format:

lv.cmp\_le.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the element in rA is less than or equal to the element in rB; otherwise the element bits are cleared.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[7:0]    ← repl(rA[7:0]    <= rB[7:0])
rD[15:8]   ← repl(rA[15:8]   <= rB[15:8])
rD[23:16]  ← repl(rA[23:16]  <= rB[23:16])
rD[31:24]  ← repl(rA[31:24]  <= rB[31:24])
rD[39:32]  ← repl(rA[39:32]  <= rB[39:32])
rD[47:40]  ← repl(rA[47:40]  <= rB[47:40])
rD[55:48]  ← repl(rA[55:48]  <= rB[55:48])
rD[63:56]  ← repl(rA[63:56]  <= rB[63:56])

```

## Exceptions:

None

## Vector Half-Word Elements

### lv.cmp\_le.h   Compare Less Than or Equal   lv.cmp\_le.h To

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	7	.	.	.	.	.	0
opcode 0xa						D					A					B					reserved			opcode 0x47							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

#### Format:

lv.cmp\_le.h rD, rA, rB

#### Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the element in rA is less than or equal to the element in rB; otherwise the element bits are cleared.

#### 32-bit Implementation:

N/A

#### 64-bit Implementation:

```

rD[15:0]  ← repl (rA[15:0]  <= rB[15:0])
rD[31:16] ← repl (rA[31:16] <= rB[31:16])
rD[47:32] ← repl (rA[47:32] <= rB[47:32])
rD[63:48] ← repl (rA[63:48] <= rB[63:48])

```

#### Exceptions:

None

Instruction Class  
ORV DX64 I

# lv.cmp\_lt.b Vector Byte Elements Compare Less Than lv.cmp\_lt.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x48								
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits								

## Format:

lv.cmp\_lt.b rD, rA, rB

## Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the element in rA is less than the element in rB; otherwise the element bits are cleared.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[7:0]    ← repl (rA[7:0]    <= rB[7:0])
rD[15:8]   ← repl (rA[15:8]   <= rB[15:8])
rD[23:16]  ← repl (rA[23:16]  <= rB[23:16])
rD[31:24]  ← repl (rA[31:24]  <= rB[31:24])
rD[39:32]  ← repl (rA[39:32]  <= rB[39:32])
rD[47:40]  ← repl (rA[47:40]  <= rB[47:40])
rD[55:48]  ← repl (rA[55:48]  <= rB[55:48])
rD[63:56]  ← repl (rA[63:56]  <= rB[63:56])

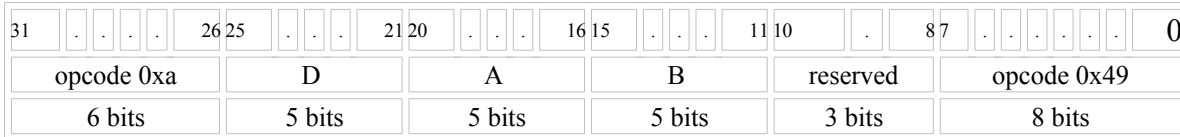
```

## Exceptions:

None



# lv.cmp\_lt.h Vector Half-Word Elements Compare Less Than lv.cmp\_lt.h



## Format:

lv.cmp\_lt.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the element in rA is less than the element in rB; otherwise the element bits are cleared.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[15:0] ← repl(rA[15:0] <= rB[15:0])
rD[31:16] ← repl(rA[31:16] <= rB[31:16])
rD[47:32] ← repl(rA[47:32] <= rB[47:32])
rD[63:48] ← repl(rA[63:48] <= rB[63:48])

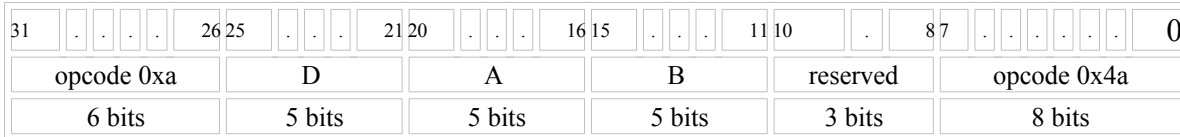
```

## Exceptions:

None

**lv.cmp\_ne.b**

## Vector Byte Elements Compare Not Equal

**lv.cmp\_ne.b**

### Format:

```
lv.cmp_ne.b rD, rA, rB
```

### Description:

All byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the two corresponding compared elements are not equal; otherwise the element bits are cleared.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← repl(rA[7:0]    != rB[7:0])
rD[15:8]   ← repl(rA[15:8]   != rB[15:8])
rD[23:16]  ← repl(rA[23:16]  != rB[23:16])
rD[31:24]  ← repl(rA[31:24]  != rB[31:24])
rD[39:32]  ← repl(rA[39:32]  != rB[39:32])
rD[47:40]  ← repl(rA[47:40]  != rB[47:40])
rD[55:48]  ← repl(rA[55:48]  != rB[55:48])
rD[63:56]  ← repl(rA[63:56]  != rB[63:56])

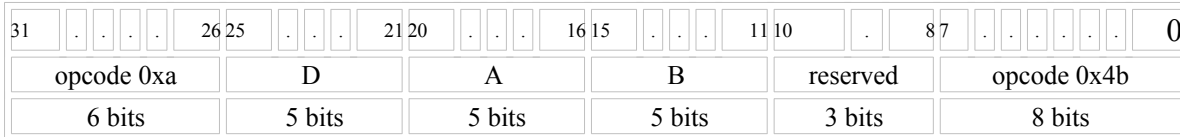
```

### Exceptions:

None

Instruction Class  
ORV DX64 I

# lv.cmp\_ne.h Vector Half-Word Elements Compare Not Equal lv.cmp\_ne.h



## Format:

lv.cmp\_ne.h rD, rA, rB

## Description:

All half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB. Bits of the element in general-purpose register rD are set if the two corresponding compared elements are not equal; otherwise the element bits are cleared.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[15:0]  ← repl(rA[15:0]  != rB[15:0])
rD[31:16] ← repl(rA[31:16] != rB[31:16])
rD[47:32] ← repl(rA[47:32] != rB[47:32])
rD[63:48] ← repl(rA[63:48] != rB[63:48])

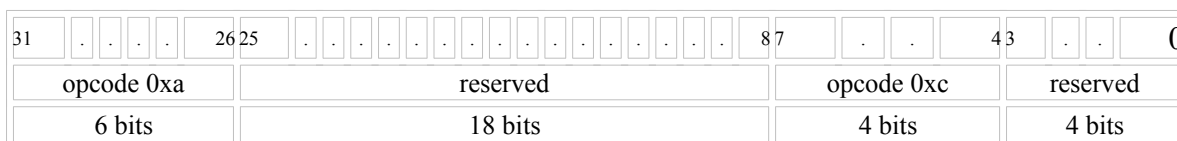
```

## Exceptions:

None

**lv.cust1**

## Reserved for Custom Vector Instructions

**lv.cust1**

### Format:

lv.cust1

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but instead by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

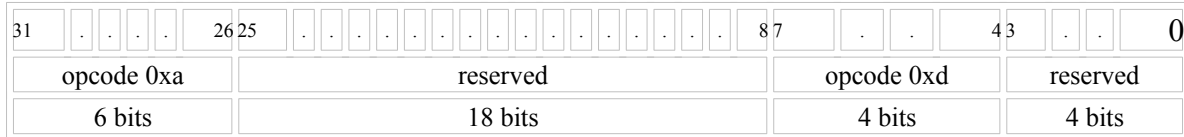
N/A

### Exceptions:

N/A

Instruction Class  
ORV DX64 II

**lv.cust2**      **Reserved for Custom Vector Instructions**      **lv.cust2**



### Format:

lv.cust2

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but instead by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

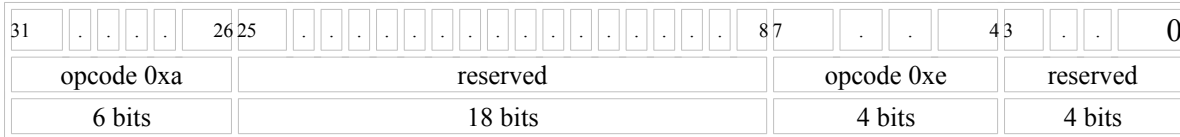
N/A

## Exceptions:

N/A

**lv.cust3**

## Reserved for Custom Vector Instructions

**lv.cust3**

### Format:

lv.cust3

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but instead by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

N/A

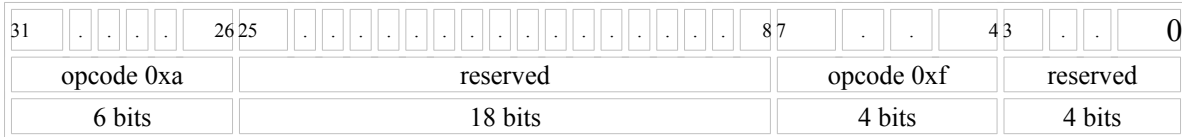
### Exceptions:

N/A

Instruction Class  
ORV DX64 II

**lv.cust4**

## Reserved for Custom Vector Instructions

**lv.cust4**

### Format:

lv.cust4

### Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but instead by the implementation itself.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

N/A

### Exceptions:

N/A

Instruction Class  
ORV DX64 II

# lv.madds.h      Vector Half-Word Elements      lv.madds.h

## Multiply Add Signed Saturated

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x54												
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits												

### Format:

lv.madds.h rD, rA, rB

### Description:

The signed half-word elements of general-purpose register rA are multiplied by the signed half-word elements of general-purpose register rB to form intermediate results. They are then added to the signed half-word VMAC elements to form the final results that are placed again in the VMAC registers. The intermediate result is placed into general-purpose register rD. If any of the final results exceeds the min/max value, it is saturated.

Note: The ORVDX instruction set is not completely specified. This instruction is incorrectly specified in that VMAC is not defined and implementation below does not match description.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[15:0]  ← sat32s(rA[15:0] * rB[15:0] + VMACLO[31:0])
rD[31:16] ← sat32s(rA[31:16] * rB[31:16] + VMACLO[63:32])
rD[47:32] ← sat32s(rA[47:32] * rB[47:32] + VMACHI[31:0])
rD[63:48] ← sat32s(rA[63:48] * rB[63:48] + VMACHI[63:32])

```

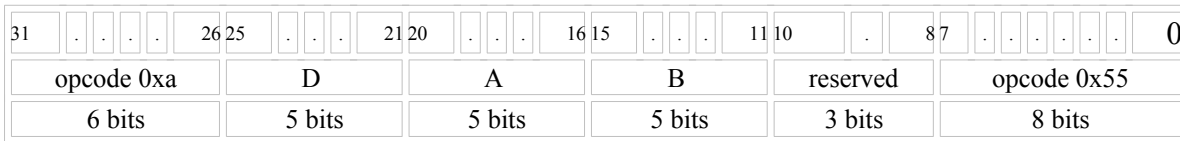
### Exceptions:

None

Instruction Class  
ORVDX64 I



## lv.max.b    Vector Byte Elements Maximum    lv.max.b



### Format:

lv.max.b rD, rA, rB

### Description:

The byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB, and the larger elements are selected to form the result elements. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]   ← rA[7:0]   > rB[7:0]   ? rA[7:0]   : rB[7:0]
rD[15:8]  ← rA[15:8]  > rB[15:8]  ? rA[15:8]  : rB[15:8]
rD[23:16] ← rA[23:16] > rB[23:16] ? rA[23:16] : rB[23:16]
rD[31:24] ← rA[31:24] > rB[31:24] ? rA[31:24] : rB[31:24]
rD[39:32] ← rA[39:32] > rB[39:32] ? rA[39:32] : rB[39:32]
rD[47:40] ← rA[47:40] > rB[47:40] ? rA[47:40] : rB[47:40]
rD[55:48] ← rA[55:48] > rB[55:48] ? rA[55:48] : rB[55:48]
rD[63:56] ← rA[63:56] > rB[63:56] ? rA[63:56] : rB[63:56]

```

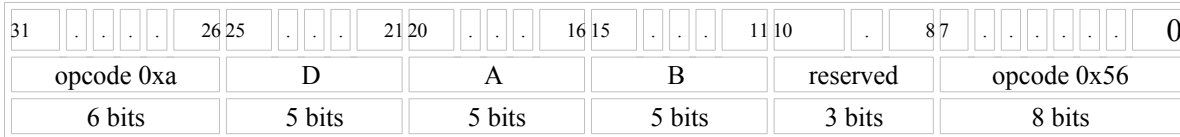
### Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.max.h      Vector Half-Word Elements      lv.max.h

## Maximum



### Format:

lv.max.h rD, rA, rB

### Description:

The half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB, and the larger elements are selected to form the result elements. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[15:0]  ← rA[15:0]  > rB[15:0]  ? rA[15:0]  : rB[15:0]
rD[31:16] ← rA[31:16] > rB[31:16] ? rA[31:16] : rB[31:16]
rD[47:32] ← rA[47:32] > rB[47:32] ? rA[47:32] : rB[47:32]
rD[63:48] ← rA[63:48] > rB[63:48] ? rA[63:48] : rB[63:48]

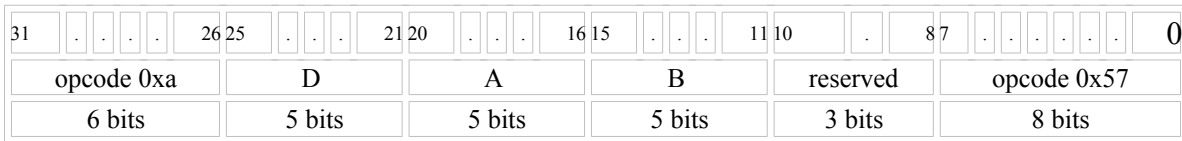
```

### Exceptions:

None

Instruction Class  
ORVDX64 I

## lv.merge.b    Vector Byte Elements Merge    lv.merge.b



### Format:

lv.merge.b rD, rA, rB

### Description:

The byte elements of the lower half of the general-purpose register rA are combined with the byte elements of the lower half of general-purpose register rB in such a way that the lowest element is from rB, the second element from rA, the third again from rB etc. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← rB[7:0]
rD[15:8]   ← rA[15:8]
rD[23:16]  ← rB[23:16]
rD[31:24]  ← rA[31:24]
rD[39:32]  ← rB[39:32]
rD[47:40]  ← rA[47:40]
rD[55:48]  ← rB[55:48]
rD[63:56]  ← rA[63:56]

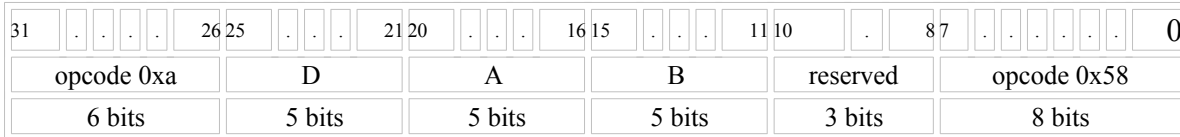
```

### Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.merge.h Vector Half-Word Elements Merge lv.merge.h



## Format:

lv.merge.h rD, rA, rB

## Description:

The half-word elements of the lower half of the general-purpose register rA are combined with the half-word elements of the lower half of general-purpose register rB in such a way that the lowest element is from rB, the second element from rA, the third again from rB etc. The result elements are placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[15:0]  ← rB[15:0]
rD[31:16] ← rA[31:16]
rD[47:32] ← rB[47:32]
rD[63:48] ← rA[63:48]

```

## Exceptions:

None

## lv.min.b      Vector Byte Elements Minimum      lv.min.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x59							
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits							

### Format:

lv.min.b rD, rA, rB

### Description:

The byte elements of general-purpose register rA are compared to the byte elements of general-purpose register rB, and the smaller elements are selected to form the result elements. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]   ← rA[7:0]   < rB[7:0]   ? rA[7:0]   : rB[7:0]
rD[15:8]  ← rA[15:8]  < rB[15:8]  ? rA[15:8]  : rB[15:8]
rD[23:16] ← rA[23:16] < rB[23:16] ? rA[23:16] : rB[23:16]
rD[31:24] ← rA[31:24] < rB[31:24] ? rA[31:24] : rB[31:24]
rD[39:32] ← rA[39:32] < rB[39:32] ? rA[39:32] : rB[39:32]
rD[47:40] ← rA[47:40] < rB[47:40] ? rA[47:40] : rB[47:40]
rD[55:48] ← rA[55:48] < rB[55:48] ? rA[55:48] : rB[55:48]
rD[63:56] ← rA[63:56] < rB[63:56] ? rA[63:56] : rB[63:56]

```

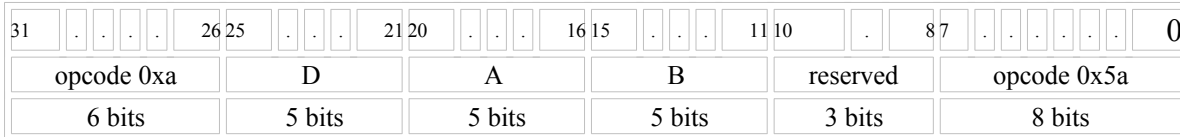
### Exceptions:

None

Instruction Class  
ORVDX64 I

**lv.min.h**

## Vector Half-Word Elements Minimum

**lv.min.h**

### Format:

```
lv.min.h rD, rA, rB
```

### Description:

The half-word elements of general-purpose register rA are compared to the half-word elements of general-purpose register rB, and the smaller elements are selected to form the result elements. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[15:0]  ← rA[15:0]  < rB[15:0]  ? rA[15:0]  : rB[15:0]
rD[31:16] ← rA[31:16] < rB[31:16] ? rA[31:16] : rB[31:16]
rD[47:32] ← rA[47:32] < rB[47:32] ? rA[47:32] : rB[47:32]
rD[63:48] ← rA[63:48] < rB[63:48] ? rA[63:48] : rB[63:48]

```

### Exceptions:

None

Instruction Class  
ORVDX64 I

# Vector Half-Word Elements

## lv.msubs.h      Multiply Subtract Signed      lv.msubs.h

### Saturated

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved			opcode 0x5b								
6 bits					5 bits					5 bits					5 bits					3 bits			8 bits								

### Format:

lv.msubs.h rD, rA, rB

### Description:

The signed half-word elements of general-purpose register rA are multiplied by the signed half-word elements of general-purpose register rB to form intermediate results. They are then subtracted from the signed half-word VMAC elements to form the final results that are placed again in the VMAC registers. The intermediate result is placed into general-purpose register rD. If any of the final results exceeds the min/max value, it is saturated.

Note: The ORV DX instruction set is not completely specified. This instruction is incorrectly specified in that VMAC is not defined and implementation below does not match description.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[15:0]  ← sat32s(VMACLO[31:0] - rA[15:0] * rB[15:0])
rD[31:16] ← sat32s(VMACLO[63:32] - rA[31:16] * rB[31:16])
rD[47:32] ← sat32s(VMACHI[31:0] - rA[47:32] * rB[47:32])
rD[63:48] ← sat32s(VMACHI[63:32] - rA[63:48] * rB[63:48])

```

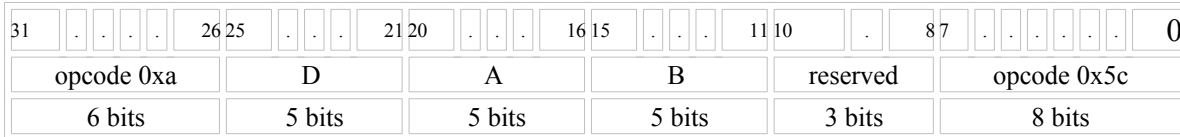
### Exceptions:

None

Instruction Class  
ORV DX64 I

**lv.muls.h**

## Vector Half-Word Elements Multiply Signed Saturated

**lv.muls.h**

### Format:

```
lv.muls.h rD, rA, rB
```

### Description:

The signed half-word elements of general-purpose register rA are multiplied by the signed half-word elements of general-purpose register rB to form the results. The result is placed into general-purpose register rD. If any of the final results exceeds the min/max value, it is saturated.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[15:0]  ← sat16s(rA[15:0] * rB[15:0])
rD[31:16] ← sat16s(rA[31:16] * rB[31:16])
rD[47:32] ← sat16s(rA[47:32] * rB[47:32])
rD[63:48] ← sat16s(rA[63:48] * rB[63:48])

```

### Exceptions:

None



**lv.nand****Vector Not And****lv.nand**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x5d							
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits							

**Format:**

```
lv.nand rD, rA, rB
```

**Description:**

The contents of general-purpose register rA are combined with the contents of general-purpose register rB in a bit-wise logical NAND operation. The result is placed into general-purpose register rD.

**32-bit Implementation:**

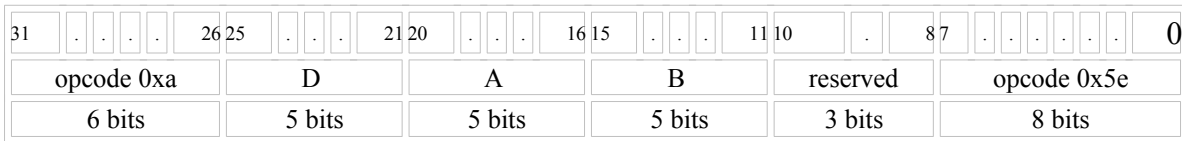
N/A

**64-bit Implementation:**

```
rD[63:0] ← rA[63:0] NAND rB[63:0]
```

**Exceptions:**

None

**lv.nor****Vector Not Or****lv.nor****Format:**

```
lv.nor rD, rA, rB
```

**Description:**

The contents of general-purpose register rA are combined with the contents of general-purpose register rB in a bit-wise logical NOR operation. The result is placed into general-purpose register rD.

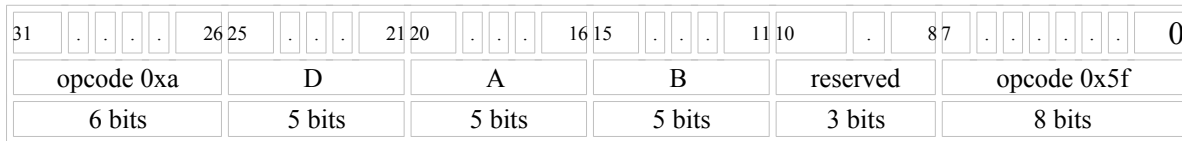
**32-bit Implementation:**

N/A

**64-bit Implementation:**

$$rD[63:0] \leftarrow rA[63:0] \text{ NOR } rB[63:0]$$
**Exceptions:**

None

**lv.or****Vector Or****lv.or****Format:**

```
lv.or rD, rA, rB
```

**Description:**

The contents of general-purpose register rA are combined with the contents of general-purpose register rB in a bit-wise logical OR operation. The result is placed into general-purpose register rD.

**32-bit Implementation:**

N/A

**64-bit Implementation:**

$$rD[63:0] \leftarrow rA[63:0] \text{ OR } rB[63:0]$$
**Exceptions:**

None

**Format:**

```
lv.pack.b rD, rA, rB
```

**Description:**

The lower half of the byte elements of the general-purpose register rA are truncated and combined with the lower half of the byte truncated elements of the general-purpose register rB in such a way that the lowest elements are from rB, and the highest elements from rA. The result elements are placed into general-purpose register rD.

**64-bit Implementation:**

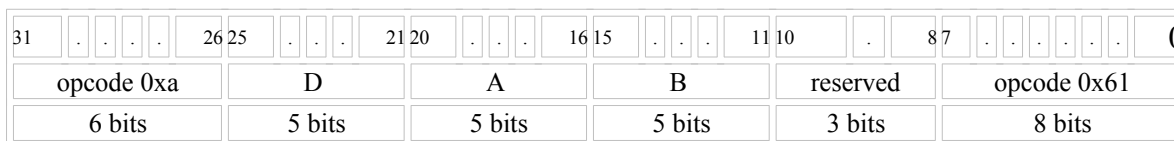
Instruction Class  
ORVDX64 I

```
rD[3:0]    ← rB[3:0]
rD[7:4]    ← rB[11:8]
rD[11:8]   ← rB[19:16]
rD[15:12]  ← rB[27:24]
rD[19:16]  ← rB[35:32]
rD[23:20]  ← rB[43:40]
rD[27:24]  ← rB[51:48]
rD[31:28]  ← rB[59:56]
rD[35:32]  ← rA[3:0]
rD[39:36]  ← rA[11:8]
rD[43:40]  ← rA[19:16]
rD[47:44]  ← rA[27:24]
rD[51:48]  ← rA[35:32]
rD[55:52]  ← rA[43:40]
rD[59:56]  ← rA[51:48]
rD[63:60]  ← rA[59:56]
```

**Exceptions:**

None

## lv.pack.h Vector Half-word Elements Pack lv.pack.h



### Format:

```
lv.pack.h rD, rA, rB
```

### Description:

The lower half of the half-word elements of the general-purpose register rA are truncated and combined with the lower half of the half-word truncated elements of the general-purpose register rB in such a way that the lowest elements are from rB, and the highest elements from rA. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← rB[7:0]
rD[15:8]   ← rB[23:16]
rD[23:16]  ← rB[39:32]
rD[31:24]  ← rB[55:48]
rD[39:32]  ← rA[7:0]
rD[47:40]  ← rA[23:16]
rD[55:48]  ← rA[39:32]
rD[63:56]  ← rA[55:48]

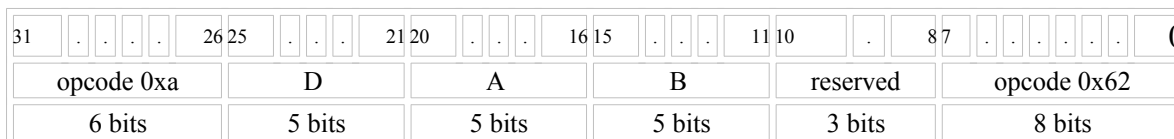
```

### Exceptions:

None

Instruction Class  
ORVDX64 I

## lv.packs.b Vector Byte Elements Pack Signed Saturated lv.packs.b



### Format:

lv.packs.b rD, rA, rB

### Description:

The lower half of the signed byte elements of the general-purpose register rA are truncated and combined with the lower half of the signed byte truncated elements of the general-purpose register rB in such a way that the lowest elements are from rB, and the highest elements from rA. If any truncated element exceeds a signed 4-bit value, it is saturated. The result elements are placed into general-purpose register rD.

### 64-bit Implementation:

```

rD[3:0]    ← sat4s(rB[7:0])
rD[7:4]    ← sat4s(rB[15:8])
rD[11:8]   ← sat4s(rB[23:16])
rD[15:12]  ← sat4s(rB[31:24])
rD[19:16]  ← sat4s(rB[39:32])
rD[23:20]  ← sat4s(rB[47:40])
rD[27:24]  ← sat4s(rB[55:48])
rD[31:28]  ← sat4s(rB[63:56])
rD[35:32]  ← sat4s(rA[7:0])
rD[39:36]  ← sat4s(rA[15:8])
rD[43:40]  ← sat4s(rA[23:16])
rD[47:44]  ← sat4s(rA[31:24])
rD[51:48]  ← sat4s(rA[39:32])
rD[55:52]  ← sat4s(rA[47:40])
rD[59:56]  ← sat4s(rA[55:48])
rD[63:60]  ← sat4s(rA[63:56])

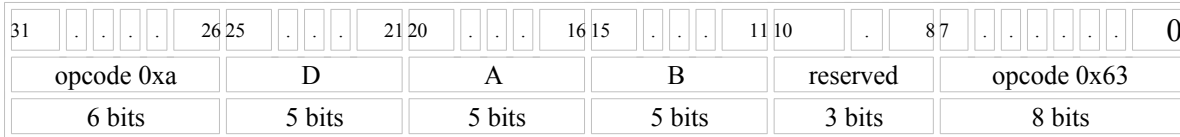
```

Exceptions:

None

Instruction Class  
ORV DX64 I

# lv.packs.h Vector Half-word Elements Pack Signed Saturated lv.packs.h



## Format:

lv.packs.h rD, rA, rB

## Description:

The lower half of the signed halfword elements of the general-purpose register rA are truncated and combined with the lower half of the signed half-word truncated elements of the general-purpose register rB in such a way that the lowest elements are from rB, and the highest elements from rA. If any truncated element exceeds a signed 8-bit value, it is saturated. The result elements are placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

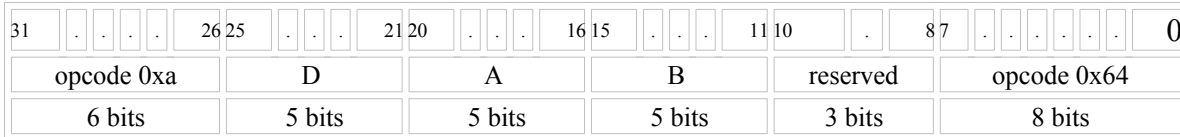
rD[7:0]    ← sat8s(rB[15:0])
rD[15:8]   ← sat8s(rB[31:16])
rD[23:16]  ← sat8s(rB[47:32])
rD[31:24]  ← sat8s(rB[63:48])
rD[39:32]  ← sat8s(rA[15:0])
rD[47:40]  ← sat8s(rA[31:16])
rD[55:48]  ← sat8s(rA[47:32])
rD[63:56]  ← sat8s(rA[63:48])

```

## Exceptions:

None

# lv.packus.b Vector Byte Elements Pack Unsigned Saturated lv.packus.b



## Format:

lv.packus.b rD, rA, rB

## Description:

The lower half of the unsigned byte elements of the general-purpose register rA are truncated and combined with the lower half of the unsigned byte truncated elements of the general-purpose register rB in such a way that the lowest elements are from rB, and the highest elements from rA. If any truncated element exceeds an unsigned 4-bit value, it is saturated. The result elements are placed into general-purpose register rD.

## 64-bit Implementation:

```

rD[3:0]    ← sat4u(rB[7:0])
rD[7:4]    ← sat4u(rB[15:8])
rD[11:8]   ← sat4u(rB[23:16])
rD[15:12]  ← sat4u(rB[31:24])
rD[19:16]  ← sat4u(rB[39:32])
rD[23:20]  ← sat4u(rB[47:40])
rD[27:24]  ← sat4u(rB[55:48])
rD[31:28]  ← sat4u(rB[63:56])
rD[35:32]  ← sat4u(rA[7:0])
rD[39:36]  ← sat4u(rA[15:8])
rD[43:40]  ← sat4u(rA[23:16])
rD[47:44]  ← sat4u(rA[31:24])
rD[51:48]  ← sat4u(rA[39:32])
rD[55:52]  ← sat4u(rA[47:40])
rD[59:56]  ← sat4u(rA[55:48])
rD[63:60]  ← sat4u(rA[63:56])

```

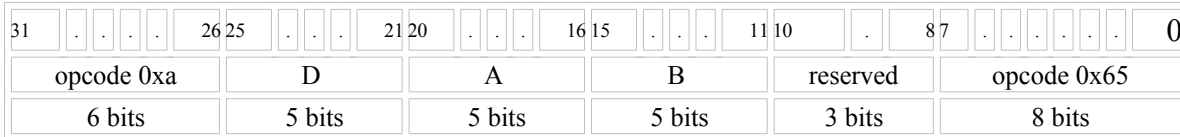
## Exceptions:

None

Instruction Class  
ORV DX64 I



# lv.packus.h Vector Half-word Elements Pack Unsigned Saturated lv.packus.h



## Format:

lv.packus.h rD, rA, rB

## Description:

The lower half of the unsigned halfword elements of the general-purpose register rA are truncated and combined with the lower half of the unsigned half-word truncated elements of the general-purpose register rB in such a way that the lowest elements are from rB, and the highest elements from rA. If any truncated element exceeds an unsigned 8-bit value, it is saturated. The result elements are placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[7:0]    ← sat8u(rB[15:0])
rD[15:8]   ← sat8u(rB[31:16])
rD[23:16]  ← sat8u(rB[47:32])
rD[31:24]  ← sat8u(rB[63:48])
rD[39:32]  ← sat8u(rA[15:0])
rD[47:40]  ← sat8u(rA[31:16])
rD[55:48]  ← sat8u(rA[47:32])
rD[63:56]  ← sat8u(rA[63:48])

```

## Exceptions:

None

Instruction Class  
ORV DX64 I

## lv.perm.n Vector Nibble Elements Permute lv.perm.n

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10		.	8	7	.	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x66							
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits							

### Format:

lv.perm.n rD, rA, rB

### Description:

The 4-bit elements of general-purpose register rA are permuted according to the corresponding 4-bit values in general-purpose register rB. The result elements are placed into general-purpose register rD.

### 64-bit Implementation:

```

rD[3:0]    ← rA[rB[3:0]*4+3:rB[3:0]*4]
rD[7:4]    ← rA[rB[7:4]*4+3:rB[7:4]*4]
rD[11:8]   ← rA[rB[11:8]*4+3:rB[11:8]*4]
rD[15:12]  ← rA[rB[15:12]*4+3:rB[15:12]*4]
rD[19:16]  ← rA[rB[19:16]*4+3:rB[19:16]*4]
rD[23:20]  ← rA[rB[23:20]*4+3:rB[23:20]*4]
rD[27:24]  ← rA[rB[27:24]*4+3:rB[27:24]*4]
rD[31:28]  ← rA[rB[31:28]*4+3:rB[31:28]*4]
rD[35:32]  ← rA[rB[35:32]*4+3:rB[35:32]*4]
rD[39:36]  ← rA[rB[39:36]*4+3:rB[39:36]*4]
rD[43:40]  ← rA[rB[43:40]*4+3:rB[43:40]*4]
rD[47:44]  ← rA[rB[47:44]*4+3:rB[47:44]*4]
rD[51:48]  ← rA[rB[51:48]*4+3:rB[51:48]*4]
rD[55:52]  ← rA[rB[55:52]*4+3:rB[55:52]*4]
rD[59:56]  ← rA[rB[59:56]*4+3:rB[59:56]*4]
rD[63:60]  ← rA[rB[63:60]*4+3:rB[63:60]*4]

```

Exceptions:

None

Instruction Class  
ORVDX64 I

## lv.rl.b      Vector Byte Elements Rotate Left      lv.rl.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x67							
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits							

### Format:

lv.rl.b rD, rA, rB

### Description:

The contents of byte elements of general-purpose register rA are rotated left by the number of bits specified in the lower 3 bits in each byte element of general-purpose register rB. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]   ← rA[7:0]   rl rB[2:0]
rD[15:8]  ← rA[15:8]  rl rB[10:8]
rD[23:16] ← rA[23:16] rl rB[18:16]
rD[31:24] ← rA[31:24] rl rB[26:24]
rD[39:32] ← rA[39:32] rl rB[34:32]
rD[47:40] ← rA[47:40] rl rB[42:40]
rD[55:48] ← rA[55:48] rl rB[50:48]
rD[63:56] ← rA[63:56] rl rB[58:56]

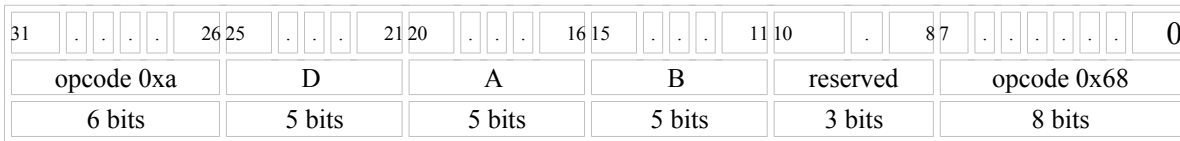
```

### Exceptions:

None

Instruction Class  
ORVDX64 I

## lv.rl.h Vector Half-Word Elements Rotate Left lv.rl.h



### Format:

lv.rl.h rD, rA, rB

### Description:

The contents of half-word elements of general-purpose register rA are rotated left by the number of bits specified in the lower 4 bits in each half-word element of general-purpose register rB. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

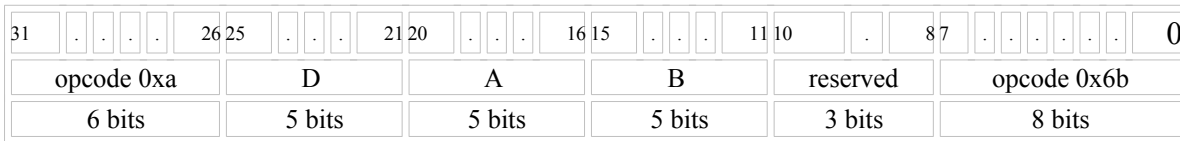
```

rD[15:0]  ← rA[15:0]  rl rB[3:0]
rD[31:16] ← rA[31:16] rl rB[19:16]
rD[47:32] ← rA[47:32] rl rB[35:32]
rD[63:48] ← rA[63:48] rl rB[51:48]

```

### Exceptions:

None

**lv.sll****Vector Shift Left Logical****lv.sll****Format:**

```
lv.sll rD, rA, rB
```

**Description:**

The contents of general-purpose register rA are shifted left by the number of bits specified in the lower 4 bits in each byte element of general-purpose register rB, inserting zeros into the low-order bits of rD. The result elements are placed into general-purpose register rD.

Note: The ORV DX instruction set is not completely specified. This instruction is incorrectly specified in that implementation below does not operate in a vector fashion and no element size is specified in the mnemonic. It may be a remnant of a template or lv.sll.b.

**32-bit Implementation:**

N/A

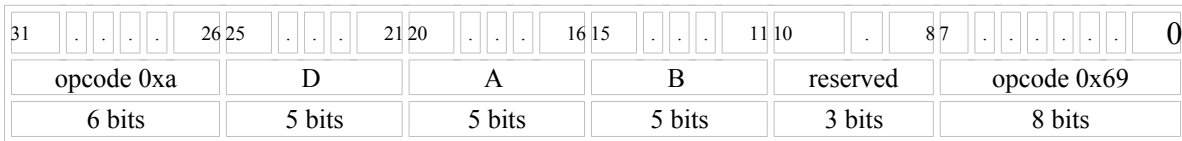
**64-bit Implementation:**

$$rD[63:0] \leftarrow rA[63:0] \ll rB[2:0]$$
**Exceptions:**

None

Instruction Class  
ORV DX64 I

## lv.sll.b Vector Byte Elements Shift Left Logical lv.sll.b



### Format:

lv.sll.b rD, rA, rB

### Description:

The contents of byte elements of general-purpose register rA are shifted left by the number of bits specified in the lower 3 bits in each byte element of general-purpose register rB, inserting zeros into the low-order bits. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← rA[7:0]    << rB[2:0]
rD[15:8]   ← rA[15:8]   << rB[10:8]
rD[23:16]  ← rA[23:16]  << rB[18:16]
rD[31:24]  ← rA[31:24]  << rB[26:24]
rD[39:32]  ← rA[39:32]  << rB[34:32]
rD[47:40]  ← rA[47:40]  << rB[42:40]
rD[55:48]  ← rA[55:48]  << rB[50:48]
rD[63:56]  ← rA[63:56]  << rB[58:56]

```

### Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.sll.h Vector Half-Word Elements Shift Left Logical

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x6a							
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits							

## Format:

lv.sll.h rD, rA, rB

## Description:

The contents of half-word elements of general-purpose register rA are shifted left by the number of bits specified in the lower 4 bits in each half-word element of general-purpose register rB, inserting zeros into the low-order bits. The result elements are placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

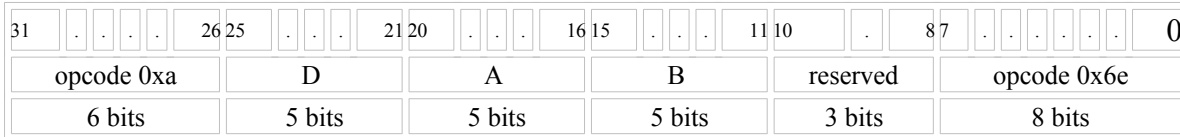
rD[15:0]  ← rA[15:0]  << rB[3:0]
rD[31:16] ← rA[31:16] << rB[19:16]
rD[47:32] ← rA[47:32] << rB[35:32]
rD[63:48] ← rA[63:48] << rB[51:48]

```

## Exceptions:

None

# lv.sra.b      Vector Byte Elements Shift Right Arithmetic      lv.sra.b



## Format:

lv.sra.b rD, rA, rB

## Description:

The contents of byte elements of general-purpose register rA are shifted right by the number of bits specified in the lower 3 bits in each byte element of general-purpose register rB, inserting the most significant bit of each element into the high-order bits. The result elements are placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[7:0]   ← rA[7:0]   sra rB[2:0]
rD[15:8]  ← rA[15:8]  sra rB[10:8]
rD[23:16] ← rA[23:16] sra rB[18:16]
rD[31:24] ← rA[31:24] sra rB[26:24]
rD[39:32] ← rA[39:32] sra rB[34:32]
rD[47:40] ← rA[47:40] sra rB[42:40]
rD[55:48] ← rA[55:48] sra rB[50:48]
rD[63:56] ← rA[63:56] sra rB[58:56]

```

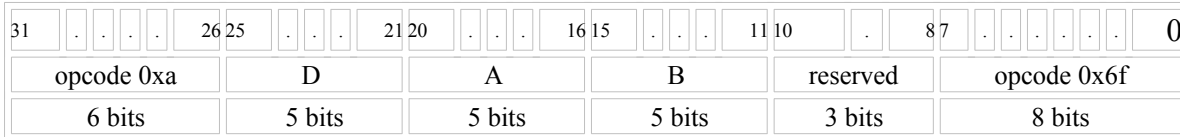
## Exceptions:

None

Instruction Class  
ORV DX64 I



# lv.sra.h Vector Half-Word Elements Shift Right Arithmetic



## Format:

lv.sra.h rD, rA, rB

## Description:

The contents of half-word elements of general-purpose register rA are shifted right by the number of bits specified in the lower 4 bits in each half-word element of general-purpose register rB, inserting the most significant bit of each element into the high-order bits. The result elements are placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[15:0]  ← rA[15:0]  sra rB[3:0]
rD[31:16] ← rA[31:16] sra rB[19:16]
rD[47:32] ← rA[47:32] sra rB[35:32]
rD[63:48] ← rA[63:48] sra rB[51:48]

```

## Exceptions:

None

**lv.srl****Vector Shift Right Logical****lv.srl**

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x70					
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits					

**Format:**

```
lv.srl rD, rA, rB
```

**Description:**

The contents of general-purpose register rA are shifted right by the number of bits specified in the lower 4 bits in each byte element of general-purpose register rB, inserting zeros into the high-order bits of rD. The result elements are placed into general-purpose register rD.

Note: The ORV DX instruction set is not completely specified. This instruction is incorrectly specified in that implementation below does not operate in a vector fashion and no element size is specified in the mnemonic. It may be a remnant of a template or lv.srl.b.

**32-bit Implementation:**

N/A

**64-bit Implementation:**

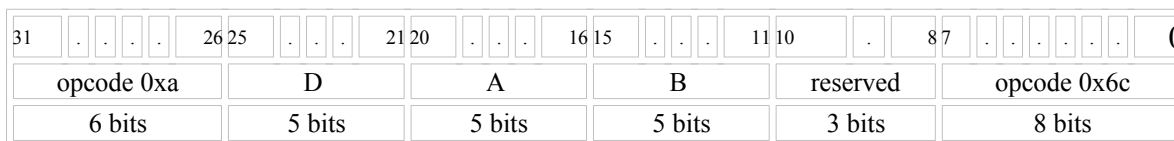
```
rD[63:0] ← rA[63:0] >> rB[2:0]
```

**Exceptions:**

None

Instruction Class  
ORV DX64 I

## lv.srl.b Vector Byte Elements Shift Right Logicallylv.srl.b



### Format:

lv.srl.b rD, rA, rB

### Description:

The contents of byte elements of general-purpose register rA are shifted right by the number of bits specified in the lower 3 bits in each byte element of general-purpose register rB, inserting zeros into the high-order bits. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← rA[7:0]    >> rB[2:0]
rD[15:8]   ← rA[15:8]   >> rB[10:8]
rD[23:16]  ← rA[23:16]  >> rB[18:16]
rD[31:24]  ← rA[31:24]  >> rB[26:24]
rD[39:32]  ← rA[39:32]  >> rB[34:32]
rD[47:40]  ← rA[47:40]  >> rB[42:40]
rD[55:48]  ← rA[55:48]  >> rB[50:48]
rD[63:56]  ← rA[63:56]  >> rB[58:56]

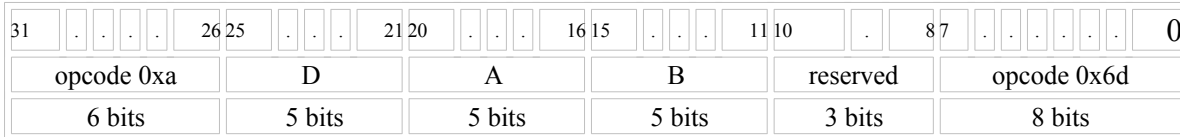
```

### Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.srl.h Vector Half-Word Elements Shift Right Logical



## Format:

lv.srl.h rD, rA, rB

## Description:

The contents of half-word elements of general-purpose register rA are shifted right by the number of bits specified in the lower 4 bits in each half-word element of general-purpose register rB, inserting zeros into the high-order bits. The result elements are placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

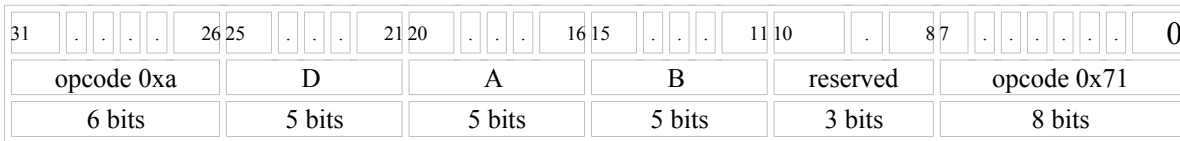
rD[15:0]  ← rA[15:0]  >> rB[3:0]
rD[31:16] ← rA[31:16] >> rB[19:16]
rD[47:32] ← rA[47:32] >> rB[35:32]
rD[63:48] ← rA[63:48] >> rB[51:48]

```

## Exceptions:

None

## lv.sub.b Vector Byte Elements Subtract Signed lv.sub.b



### Format:

lv.sub.b rD, rA, rB

### Description:

The byte elements of general-purpose register rB are subtracted from the byte elements of general-purpose register rA to form the result elements. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← rA[7:0]    - rB[7:0]
rD[15:8]   ← rA[15:8]   - rB[15:8]
rD[23:16]  ← rA[23:16]  - rB[23:16]
rD[31:24]  ← rA[31:24]  - rB[31:24]
rD[39:32]  ← rA[39:32]  - rB[39:32]
rD[47:40]  ← rA[47:40]  - rB[47:40]
rD[55:48]  ← rA[55:48]  - rB[55:48]
rD[63:56]  ← rA[63:56]  - rB[63:56]

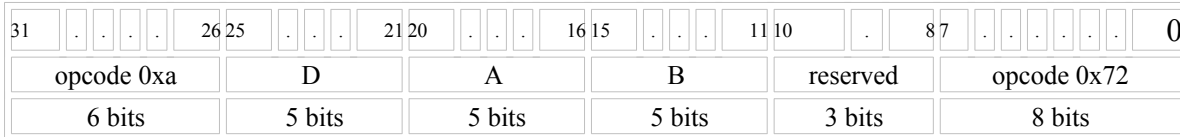
```

### Exceptions:

None

Instruction Class  
ORVDX64 I

# lv.sub.h Vector Half-Word Elements Subtract Signed lv.sub.h



## Format:

lv.sub.h rD, rA, rB

## Description:

The half-word elements of general-purpose register rB are subtracted from the half-word elements of general-purpose register rA to form the result elements. The result elements are placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

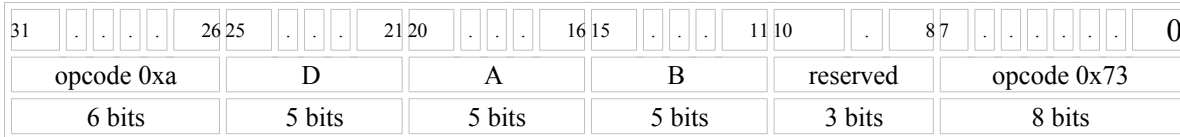
rD[15:0]  ← rA[15:0]  - rB[15:0]
rD[31:16] ← rA[31:16] - rB[31:16]
rD[47:32] ← rA[47:32] - rB[47:32]
rD[63:48] ← rA[63:48] - rB[63:48]

```

## Exceptions:

None

# lv.subs.b      Vector Byte Elements Subtract Signed Saturated      lv.subs.b



## Format:

lv.subs.b rD, rA, rB

## Description:

The byte elements of general-purpose register rB are subtracted from the byte elements of general-purpose register rA to form the result elements. If the result exceeds the min/max value for the destination data type, it is saturated to the min/max value and placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[7:0]    ← sat8s(rA[7:0]    - rB[7:0])
rD[15:8]   ← sat8s(rA[15:8]   - rB[15:8])
rD[23:16]  ← sat8s(rA[23:16]  - rB[23:16])
rD[31:24]  ← sat8s(rA[31:24]  - rB[31:24])
rD[39:32]  ← sat8s(rA[39:32]  - rB[39:32])
rD[47:40]  ← sat8s(rA[47:40]  - rB[47:40])
rD[55:48]  ← sat8s(rA[55:48]  - rB[55:48])
rD[63:56]  ← sat8s(rA[63:56]  - rB[63:56])

```

## Exceptions:

None

# lv.subs.h Vector Half-Word Elements Subtract Signed Saturated

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	7	.	.	.	.	.	0
opcode 0xa						D					A					B					reserved			opcode 0x74								
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits								

## Format:

lv.subs.h rD, rA, rB

## Description:

The half-word elements of general-purpose register rB are subtracted from the half-word elements of general-purpose register rA to form the result elements. If the result exceeds the min/max value for the destination data type, it is saturated to the min/max value and placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[15:0]  ← sat16s(rA[15:0]  - rB[15:0])
rD[31:16] ← sat16s(rA[31:16] - rB[31:16])
rD[47:32] ← sat16s(rA[47:32] - rB[47:32])
rD[63:48] ← sat16s(rA[63:48] - rB[63:48])

```

## Exceptions:

None



# lv.subu.b      Vector Byte Elements Subtract      lv.subu.b

## Unsigned

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	8	7	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x75						
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits						

### Format:

lv.subu.b rD, rA, rB

### Description:

The unsigned byte elements of general-purpose register rB are subtracted from the unsigned byte elements of general-purpose register rA to form the result elements. The result elements are placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← rA[7:0]    - rB[7:0]
rD[15:8]   ← rA[15:8]   - rB[15:8]
rD[23:16]  ← rA[23:16]  - rB[23:16]
rD[31:24]  ← rA[31:24]  - rB[31:24]
rD[39:32]  ← rA[39:32]  - rB[39:32]
rD[47:40]  ← rA[47:40]  - rB[47:40]
rD[55:48]  ← rA[55:48]  - rB[55:48]
rD[63:56]  ← rA[63:56]  - rB[63:56]

```

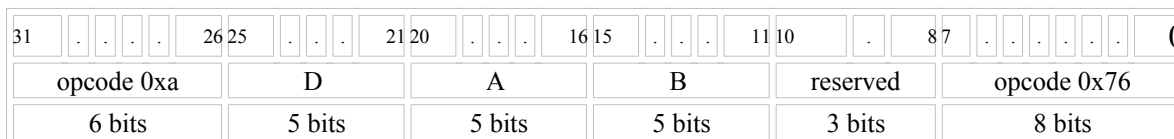
### Exceptions:

None

Instruction Class  
ORV DX64 I

**lv.subu.h**

## Vector Half-Word Elements Subtract Unsigned

**lv.subu.h****Format:**

```
lv.subu.h rD, rA, rB
```

**Description:**

The unsigned half-word elements of general-purpose register rB are subtracted from the unsigned half-word elements of general-purpose register rA to form the result elements. The result elements are placed into general-purpose register rD.

**32-bit Implementation:**

N/A

**64-bit Implementation:**

```

rD[15:0]  ← rA[15:0]  - rB[15:0]
rD[31:16] ← rA[31:16] - rB[31:16]
rD[47:32] ← rA[47:32] - rB[47:32]
rD[63:48] ← rA[63:48] - rB[63:48]

```

**Exceptions:**

None

Instruction Class  
ORV DX64 I

# lv.subus.b Vector Byte Elements Subtract Unsigned Saturated lv.subus.b

31	.	.	.	.	26	25	.	.	.	.	21	20	.	.	.	.	16	15	.	.	.	.	11	10	.	.	.	.	.	8	7	.	.	.	.	.	.	0
opcode 0xa					D					A					B					reserved					opcode 0x77													
6 bits					5 bits					5 bits					5 bits					3 bits					8 bits													

## Format:

lv.subus.b rD, rA, rB

## Description:

The unsigned byte elements of general-purpose register rB are subtracted from the unsigned byte elements of general-purpose register rA to form the result elements. If the result exceeds the min/max value for the destination data type, it is saturated to the min/max value and placed into general-purpose register rD.

## 32-bit Implementation:

N/A

## 64-bit Implementation:

```

rD[7:0]    ← sat8u(rA[7:0]    - rB[7:0])
rD[15:8]   ← sat8u(rA[15:8]   - rB[15:8])
rD[23:16]  ← sat8u(rA[23:16]  - rB[23:16])
rD[31:24]  ← sat8u(rA[31:24]  - rB[31:24])
rD[39:32]  ← sat8u(rA[39:32]  - rB[39:32])
rD[47:40]  ← sat8u(rA[47:40]  - rB[47:40])
rD[55:48]  ← sat8u(rA[55:48]  - rB[55:48])
rD[63:56]  ← sat8u(rA[63:56]  - rB[63:56])

```

## Exceptions:

None

Instruction Class  
ORV DX64 I

# lv.subus.h      Vector Half-Word Elements      lv.subus.h

## Subtract Unsigned Saturated

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	8	7	.	.	.	.	.	0
opcode 0xa						D					A					B					reserved			opcode 0x78							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

### Format:

lv.subus.h rD, rA, rB

### Description:

The unsigned half-word elements of general-purpose register rB are subtracted from the unsigned half-word elements of general-purpose register rA to form the result elements. If the result exceeds the min/max value for the destination data type, it is saturated to the min/max value and placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

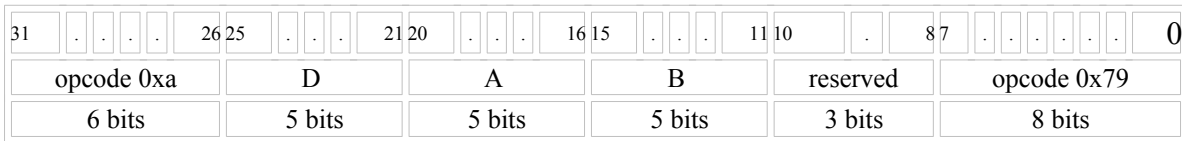
rD[15:0]  ← sat16u(rA[15:0]  - rB[15:0])
rD[31:16] ← sat16u(rA[31:16] - rB[31:16])
rD[47:32] ← sat16u(rA[47:32] - rB[47:32])
rD[63:48] ← sat16u(rA[63:48] - rB[63:48])

```

### Exceptions:

None

## lv.unpack.b Vector Byte Elements Unpack lv.unpack.b



### Format:

```
lv.unpack.b rD, rA, rB
```

### Description:

The lower half of the 4-bit elements in general-purpose register rA are sign-extended and placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

```

rD[7:0]    ← exts(rA[3:0])
rD[15:8]   ← exts(rA[7:4])
rD[23:16]  ← exts(rA[11:8])
rD[31:24]  ← exts(rA[15:12])
rD[39:32]  ← exts(rA[19:16])
rD[47:40]  ← exts(rA[23:20])
rD[55:48]  ← exts(rA[27:24])
rD[63:56]  ← exts(rA[31:28])

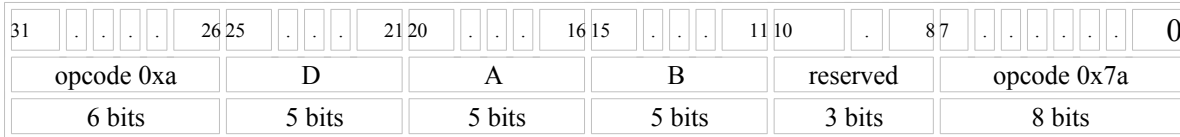
```

### Exceptions:

None

# lv.unpack.h Vector Half-Word Elements lv.unpack.h

## Unpack



### Format:

lv.unpack.h rD, rA, rB

### Description:

The lower half of the 8-bit elements in general-purpose register rA are sign-extended and placed into general-purpose register rD.

### 32-bit Implementation:

N/A

### 64-bit Implementation:

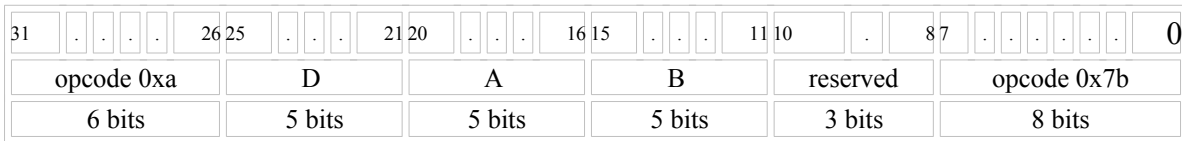
```

rD[15:0]  ← exts(rA[7:0])
rD[31:16] ← exts(rA[15:8])
rD[47:32] ← exts(rA[23:16])
rD[63:48] ← exts(rA[31:24])

```

### Exceptions:

None

**lv.xor****Vector Exclusive Or****lv.xor****Format:**

```
lv.xor rD, rA, rB
```

**Description:**

The contents of general-purpose register rA are combined with the contents of general-purpose register rB in a bit-wise logical XOR operation. The result is placed into general-purpose register rD.

**32-bit Implementation:**

N/A

**64-bit Implementation:**

$$rD[63:0] \leftarrow rA[63:0] \text{ XOR } rB[63:0]$$
**Exceptions:**

None

Instruction Class  
ORVDX64 I

## 6 Exception Model

This chapter describes the various exception types and their handling.

### 6.1 Introduction

The exception mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, information about the state of the processor is saved to certain registers and the processor begins execution at the address predetermined for each exception. Processing of exceptions begins in supervisor mode.

The OpenRISC 1000 architecture has special support for fast exception processing – also called fast context switch support. This allows very rapid interrupt processing. It is achieved with shadowing general-purpose and some special registers.

The architecture requires that all exceptions be handled in strict order with respect to the instruction stream. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream are required to complete before the exception is taken.

Exceptions can occur while an exception handler routine is executing, and multiple exceptions can become nested. Support for fast exceptions allows fast nesting of exceptions until all shadowed registers are used. If context switching is not implemented, nested exceptions should not occur.

### 6.2 Exception Classes

All exceptions can be described as precise or imprecise and either synchronous or asynchronous. Synchronous exceptions are caused by instructions and asynchronous exceptions are caused by events external to the processor.

Type	Exception
Asynchronous/nonmaskable	Bus Error, Reset
Asynchronous/maskable	External Interrupt, Tick Timer
Synchronous/precise	Instruction-caused exceptions
Synchronous/imprecise	None

**Table 6-1. Exception Classes**

Whenever an exception occurs, current PC is saved to current EPCR and new PC is set with the vector address according to 6-2.



Exception Type	Vector Offset	Causal Conditions
Reset	0x100	Caused by software or hardware reset.
Bus Error	0x200	The causes are implementation-specific, but typically they are related to bus errors and attempts to access invalid physical address.
Data Page Fault	0x300	No matching PTE found in page tables or page protection violation for load/store operations.
Instruction Page Fault	0x400	No matching PTE found in page tables or page protection violation for instruction fetch.
Tick Timer	0x500	Tick timer interrupt asserted.
Alignment	0x600	Load/store access to naturally not aligned location.
Illegal Instruction	0x700	Illegal instruction in the instruction stream.
External Interrupt	0x800	External interrupt asserted.
D-TLB Miss	0x900	No matching entry in DTLB (DTLB miss).
I-TLB Miss	0xA00	No matching entry in ITLB (ITLB miss).
Range	0xB00	If programmed in the SR, the setting of certain flags, like SR[OV], causes a range exception. On OpenRISC implementations with less than 32 GPRs when accessing unimplemented architectural GPRs. On all implementations if SR[CID] had to go out of range in order to process next exception.
System Call	0xC00	System call initiated by software.
Floating Point	0xD00	Caused by floating point instructions when FPCSR status flags are set by FPU and FPCSR[FPEE] is set
Trap	0xE00	Caused by the l.trap instruction or by debug unit.
Reserved	0xF00 – 0x1400	Reserved for future use.
Reserved	0x1500 – 0x1800	Reserved for implementation-specific exceptions.
Reserved	0x1900 – 0x1F00	Reserved for custom exceptions.

Table 6-2. Exception Types and Causal Conditions

## 6.3 Exception Processing

Whenever an exception occurs, the current/next PC is saved to the current EPCR. If the CPU implements delay-slot execution (CPUCFGR[ND] is not set) and the PC points to the delay-slot instruction, PC-4 is saved to the current EPCR and SR[DSX] is set. 6-3 defines what are current/next PC and effective address.

The SR is saved to the current ESR.

Current EPCR/ESR are identified by SR[CID]. If fast context switching is not implemented then current EPCR/ESR are always EPCR0/ESR0.

In addition, the current EEAR is set with the effective address in question if one of the following exceptions occurs: Bus Error, IMMU page fault, DMMU page fault, Alignment, I-TLB miss, D-TLB miss.

In the case of Floating Point exceptions the results are written back to registers before the exception branch occurs.

Exception	Priority	EPCR (no delay slot)	EPCR (delay slot)	EEAR
Reset	1	-	-	-
Bus Error	4 (insn) 9 (data)	Address of instruction that caused exception	Address of jump instruction before the instruction that caused exception	Load/store/fetch virtual EA
Data Page Fault	8	Address of instruction that caused exception	Address of jump instruction before the instruction that caused exception	Load/store virtual EA
Instruction Page Fault	3	Address of instruction that caused exception	Address of jump instruction before the instruction that caused exception	Instruction fetch virtual EA
Tick Timer	12	Address of next not executed instruction	Address of just executed jump instruction	-
Alignment	6	Address of instruction that caused exception	Address of jump instruction before the instruction that caused exception	Load/store virtual EA
Illegal Instruction	5	Address of instruction that caused exception	Address of jump instruction before the instruction that caused exception	Instruction fetch virtual EA
External Interrupt	12	Address of next not executed instruction	Address of just executed jump instruction	-
D-TLB Miss	7	Address of instruction that caused exception	Address of jump instruction before the instruction that caused exception	Load/store virtual EA
I-TLB Miss	2	Address of instruction that caused exception	Address of jump instruction before the instruction that caused exception	Instruction fetch virtual EA
Range	10	Address of instruction that caused exception	Address of jump instruction before the instruction that caused exception	-

Exception	Priority	EPCR (no delay slot)	EPCR (delay slot)	EEAR
System Call	7	Address of next not executed instruction	Address of just executed jump instruction	-
Floating Point	11	Address of next not executed instruction	Address of just executed jump instruction	-
Trap	7	Address of instruction that caused exception	Address of jump instruction before the instruction that caused exception	-

Table 6-3. Values of EPCR and EEAR After Exception

If fast context switching is used, SR[CID] is incremented with each new exception so that a new set of shadowed registers is used. If SR[CID] will overflow with the current exception, a range exception is invoked.

However, if SR[CE] is not set, fast context switching is not enabled. In this case all registers that will be modified by exception handler routine must first be saved.

All exceptions set a new SR where both MMUs are disabled (address translation disabled), supervisor mode is turned on, and tick timer exceptions and interrupts are disabled. (SR[DME]=0, SR[IME]=0, SR[SM]=1, SR[IEE]=0 and SR[TEE]=0).

When enough machine state information has been saved by the exception handler, SR[TTE] and SR[IEE] can be re-enabled so that tick timer and external interrupts are not blocked.

When returning from an exception handler with **l.rfe**, SR and PC are restored. If SR[CE] is set, CID will be automatically decremented and the previous machine state will be restored; otherwise, general-purpose registers previously saved by exception handler need to be restored as well.

### 6.3.1 Particular delay slot issues

Instructions placed in the delay slot will cause EPCR to be set to the address of the jump instruction, not the delay slot or target instruction. Because of this, two categories of instruction should never be placed in the delay slot:

1. Instructions altering the conditions of the jump itself. This is why **l.jr** must not have a delay slot instruction modify the target address register.
2. Instructions consistently causing an exception, such as **l.sys**. Normally **l.sys** returns to continue execution, but if placed in a delay slot it instead causes a repeat of the system call itself.

**l.trap** is generally used as a software breakpoint, so may not have the same concern.

## 6.4 Fast Context Switching (Optional)

Fast context switching is a technique that reduces register storing to stack when exceptions occur. Only one type of exception can be handled, so it is up to the software to

figure out what caused it. Using software, both interrupt handler invocation and thread switching can be handled very quickly. The hardware should be capable of switching between contexts in only one cycle.

Context can also be switched during an exception or by using a supervisor register CXR (context register) available only in supervisor mode. CXR is the same for all contexts.

### 6.4.1 Changing Context in Supervisor Mode

The read/write register CXR consists of two parts: the lower 16 bits represents the current context register set. The upper 16 bits represent the current CID. CCID cannot be accessed in user mode. Writing to CCID causes an immediate context change. Reading from CCID returns the running (current) context ID. The context where CID=0 is also called the main context.

BIT	31-16	15-0
Identifier	CCID	CCRS
Reset	0	0

CCRS has two functions:

- ✓ When an exception occurs, it holds the previous CID.
- ✓ It is used to access other context's registers.

### 6.4.2 Context Switch Caused by Exception

When an exception occurs and fast context switching is enabled, the CCID is copied to CCRS and then set to zero, thus switching to main context.

Functions of the main context are:

- ✓ Switching between threads
- ✓ Handling exceptions
- ✓ Preparing, loading, saving, and releasing context identifiers to/from the CID table

CXR should be stored in a general-purpose register as soon as possible, to allow further exception nesting.

The following table shows an example how the CID table could be used. Generally, there is no need that free exception contexts are equal.

CID	Function
7	Exception contexts
6	
5	
4	Thread contexts

CID	Function
3	
2	
1	
0	Main context

Four thread contexts are loaded, and software can switch between them freely using main context, running in supervisor mode. When an exception occurs, first need to be determined what caused it and switch to the next free exception context. Since exceptions can be nested, more free contexts may have to be available. Some of the contexts thus need to be stored to memory in order to switch to a new exception.

The algorithm used in the main context to handle context saving/restoring and switching can be kept as simple as possible. It should have enough (of its own) registers to store information such as:

- ✓ Current running CID
- ✓ Next exception
- ✓ Thread cycling info
- ✓ Pointers to context table in memory
- ✓ Copy of CXR

If the number of interrupts is significant, some sort of deferred interrupts calls mechanism can be used. The main context algorithm should store just I/O information passed by the interrupt for further execution and return from main context as soon as possible.

### 6.4.3 Accessing Other Contexts' Registers

This operation can be done only in supervisor mode. In the basic instruction set we have the `l.mtspr` and `l.mfspr` instructions that are used to access shadowed registers.

## 7 Memory Model

This chapter describes the OpenRISC 1000 weakly ordered memory model.

### 7.1 Memory

Memory is byte-addressed with halfword accesses aligned on 2-byte boundaries, singleword accesses aligned on 4-byte boundaries, and doubleword accesses aligned on 8-byte boundaries.

### 7.2 Memory Access Ordering

The OpenRISC 1000 architecture specifies a weakly ordered memory model for uniprocessor and shared memory multiprocessor systems. This model has the advantage of a higher-performance memory system but places the responsibility for strict access ordering on the programmer.

The order in which the processor performs memory access, the order in which those accesses complete in memory, and the order in which those accesses are viewed by another processor may all be different. Two means of enforcing memory access ordering are provided to allow programs in uniprocessor and multiprocessor system to share memory.

An OpenRISC 1000 processor implementation may also implement a more restrictive, strongly ordered memory model. Programs written for the weakly ordered memory model will automatically work on processors with strongly ordered memory model.

#### 7.2.1 Memory Synchronize Instruction

The **l.msinc** instruction permits the program to control the order in which load and store operations are performed. This synchronization is accomplished by requiring programs to indicate explicitly in the instruction stream, by inserting a memory sync instruction, that synchronization is required. The memory sync instruction ensures that all memory accesses initiated by a program have been performed before the next instruction is executed.

OpenRISC 1000 processor implementations, that implement the strongly-ordered memory model instead of the weakly-ordered one, can execute memory synchronization instruction as a no-operation instruction.

#### 7.2.2 Pages Designated as Weakly-Ordered-Memory

When a memory page is designated as a Weakly-Ordered-Memory (WOM) page, instructions and data can be accessed out-of-order and with prefetching. When a page is

designated as not WOM, instruction fetches and load/store operations are performed in-order without any prefetching.

OpenRISC 1000 scalar processor implementations, that implement strongly-ordered memory model instead of the weakly-ordered one and perform load and store operations in-order, are not required to implement the WOM bit in the MMU.

## 7.3 Atomicity

A memory access is atomic if it is always performed in its entirety with no visible fragmentation. Atomic memory accesses are specifically required to implement software semaphores and other shared structures in systems where two different processes on the same processor, or two different processors in a multiprocessor environment, access the same memory location with intent to modify it.

The OpenRISC 1000 architecture provides two dedicated instructions that together perform an atomic read-modify-write operation.

```
l.lwa    rD, I(rA)
l.swa    I(rA), rB
```

Instruction **l.lwa** loads single word from memory, creating a reservation for a subsequent conditional store operation. A special register, invisible to the programmer, is used to hold the address of the memory location, which is used in the atomic read-modify-write operation.

The reservation for a subsequent **l.swa** is cancelled if another store overlapping the same memory location occurs, another master writes overlapping same memory location (snoop hit), another **l.swa** (to any memory location) is executed, another **l.lwa** is executed or a context switch (exception) occur. Keep in mind that the overlapping stores may be byte or half-word size.

If a reservation is still valid when the corresponding **l.swa** is executed, **l.swa** stores general-purpose register **rB** into the memory and **SR[F]** is set.

If the reservation was cancelled, **l.swa** does not perform the store to memory and **SR[F]** is cleared.

In implementations that use a weakly-ordered memory model, **l.swa** and **l.lwa** will serve as synchronization points, similar to **l.msync**.

## 8 Memory Management

This chapter describes the virtual memory and access protection mechanisms for memory management within the OpenRISC 1000 architecture.

Note that this chapter describes the address translation mechanism from the perspective of the programming model. As such, it describes the structure of the page tables, the MMU conditions that cause MMU related exceptions and the MMU registers. The hardware implementation details that are invisible to the OpenRISC 1000 programming model, such as MMU organization and TLB size, are not contained in the architectural definition.

### 8.1 MMU Features

The OpenRISC 1000 memory management unit includes the following principal features:

- ✓ Support for effective address (EA) of 32 bits and 64 bits
- ✓ Support for implementation specific size of physical address spaces up to 35 address bits (32 GByte)
- ✓ Three different page sizes:
  - Level 0 pages (32 Gbyte; only with 64-bit EA) translated with D/I Area Translation Buffer (ATB)
  - Level 1 pages (16 MByte) translated with D/I Area Translation Buffer (ATB)
  - Level 2 pages (8 Kbyte) translated with D/I Translation Lookaside Buffer (TLB)
- ✓ Address translation using one-, two- or three-level page tables
- ✓ Powerful page based access protection with support for demand-paged virtual memory
- ✓ Support for simultaneous multi-threading (SMT)

### 8.2 MMU Overview

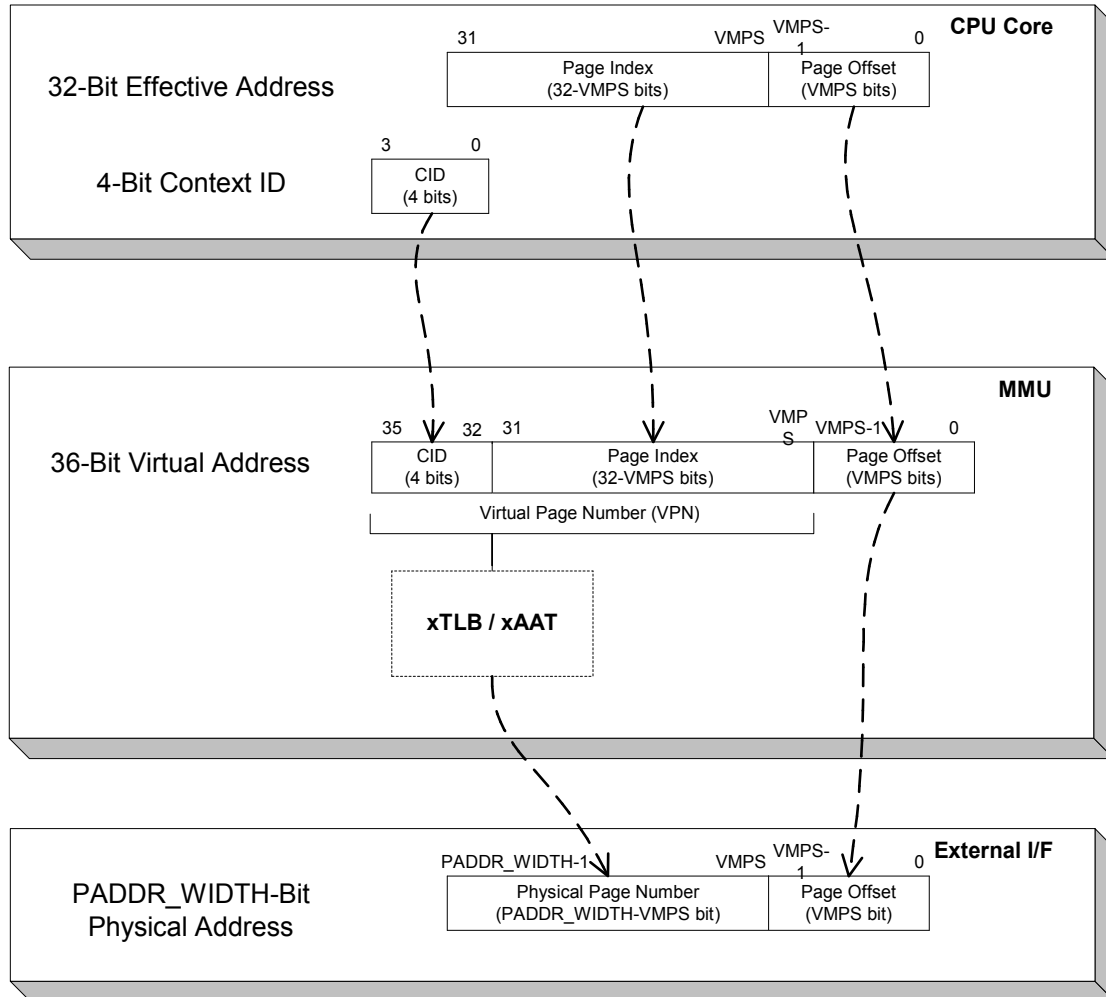
The primary functions of the MMU in an OpenRISC 1000 processor are to translate effective addresses to physical addresses for memory accesses. In addition, the MMU provides various levels of access protection on a page-by-page basis. Note that this chapter describes the conceptual model of the OpenRISC 1000 MMU and implementations may differ in the specific hardware used to implement this model.

Two general types of accesses generated by OpenRISC 1000 processors require address translation – instruction accesses generated by the instruction fetch unit, and data accesses generated by the load and store unit. Generally, the address translation mechanism is defined in terms of page tables used by OpenRISC 1000 processors to locate the effective to physical address mapping for instruction and data accesses.



The definition of page table data structures provides significant flexibility for the implementation of performance enhancement features in a wide range of processors. Therefore, the performance enhancements used to the page table information on-chip vary from implementation to implementation.

Translation lookaside buffers (TLBs) are commonly implemented in OpenRISC 1000 processors to keep recently-used page address translations on-chip. Although their exact implementation is not specified, the general concepts that are pertinent to the system software are described.



**Figure 8-1. Translation of Effective to Physical Address – Simplified block diagram for 32-bit processor implementations**

Large areas can be translated with optional facility called Area Translation Buffer (ATB). ATBs translate 16MB and 32GB pages. If xTLB and xATB have a match on the same virtual address, xTLB is used.

The MMU, together with the exception processing mechanism, provides the necessary support for the operating system to implement a paged virtual memory environment and for enforcing protection of designated memory areas.

## 8.3 MMU Exceptions

To complete any memory access, the effective address must be translated to a physical address. An MMU exception occurs if this translation fails.

TLB miss exceptions can happen only on OpenRISC 1000 processor implementations that do TLB reload in software.

The page fault exceptions that are caused by missing PTE in page table or page access protection can happen on any OpenRISC 1000 processor implementations.

EXCEPTION NAME	VECTOR OFFSET	CAUSING CONDITIONS
Data Page Fault	0x300	No matching PTE found in page tables or page protection violation for load/store operations.
Instruction Page Fault	0x400	No matching PTE found in page tables or page protection violation for instruction fetch.
DTLB Miss	0x900	No matching entry in DTLB.
ITLB Miss	0xA00	No matching entry in ITLB.

**Table 8-1. MMU Exceptions**

The vector offset addresses in table are subject to the presence and setting of the of the Exception Vector Base Address Register (EVBAR) may have configured the exceptions to be processed at a different offset, however the least-significant 12-bit offset address remain the same.

The state saved by the processor for each of the exceptions in Table 9-2 contains information that identifies the address of the failing instruction. Refer to the chapter entitled “Exception Processing” on page 274 for a more detailed description of exception processing.

## 8.4 MMU Special-Purpose Registers

8-2 summarizes the registers that the operating system uses to program the MMU. These registers are 32-bit special-purpose supervisor-level registers accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode only.

8-2 does not show two configuration registers that are implemented if implementation implements configuration registers. DMMUCFGR and IMMUCFGR describe capability of DMMU and IMMU.

Grp #	Reg #	Reg Name	USER MODE	SUPV MODE	Description
1	0	DMMUCR	–	R/W	Data MMU Control register

Grp #	Reg #	Reg Name	USER MODE	SUPV MODE	Description
1	1	DMMUPR	–	R/W	Data MMU Protection Register
1	2	DTLBEIR	–	W	Data TLB Entry Invalidate register
1	4-7	DATBMR0-DATBMR3	–	R/W	Data ATB Match registers
1	8-11	DATBTR0-DATBTR3	–	R/W	Data ATB Translate registers
1	512-639	DTLBW0MR0-DTLBW0MR127	–	R/W	Data TLB Match registers Way 0
1	640-767	DTLBW0TR0-DTLBW0TR127	–	R/W	Data TLB Translate registers Way 0
1	768-895	DTLBW1MR0-DTLBW1MR127	–	R/W	Data TLB Match registers Way 1
1	896-1023	DTLBW1TR0-DTLBW1TR127	–	R/W	Data TLB Translate registers Way 1
1	1024-1151	DTLBW2MR0-DTLBW2MR127	–	R/W	Data TLB Match registers Way 2
1	1152-1279	DTLBW2TR0-DTLBW2TR127	–	R/W	Data TLB Translate registers Way 2
1	1280-1407	DTLBW3MR0-DTLBW3MR127	–	R/W	Data TLB Match registers Way 3
1	1408-1535	DTLBW3TR0-DTLBW3TR127	–	R/W	Data TLB Translate registers Way 3
2	0	IMMUCR	–	R/W	Instruction MMU Control register
2	1	IMMUPR	–	R/W	Instruction MMU Protection Register
2	2	ITLBEIR	–	W	Instruction TLB Entry Invalidate register
2	4-7	IATBMR0-IATBMR3	–	R/W	Instruction ATB Match registers
2	8-11	IATBTR0-IATBTR3	–	R/W	Instruction ATB Translate registers
2	512-639	ITLBW0MR0-ITLBW0MR127	–	R/W	Instruction TLB Match registers Way 0
2	640-767	ITLBW0TR0-ITLBW0TR127	–	R/W	Instruction TLB Translate registers Way 0
2	768-895	ITLBW1MR0-ITLBW1MR127	–	R/W	Instruction TLB Match registers Way 1
2	896-1023	ITLBW1TR0-ITLBW1TR127	–	R/W	Instruction TLB Translate registers Way 1
2	1024-1151	ITLBW2MR0-ITLBW2MR127	–	R/W	Instruction TLB Match registers Way 2

Grp #	Reg #	Reg Name	USER MODE	SUPV MODE	Description
2	1152-1279	ITLBW2TR0-ITLBW2TR127	–	R/W	Instruction TLB Translate registers Way 2
2	1280-1407	ITLBW3MR0-ITLBW3MR127	–	R/W	Instruction TLB Match registers Way 3
2	1408-1535	ITLBW3TR0-ITLBW3TR127	–	R/W	Instruction TLB Translate registers Way 3

Table 8-2. List of MMU Special-Purpose Registers

As TLBs are noncoherent caches of PTEs, software that changes the page tables in any way must perform the appropriate TLB invalidate operations to keep the on-chip TLBs coherent with respect to the page tables in memory.

### 8.4.1 Data MMU Control Register (DMMUCR)

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

It provides general control of the DMMU.

Bit	31-10	9-1	0
Identifier	PTBP	Reserved	DTF
Reset	0	X	0
R/W	R/W	R	R/W

DTF	DTLB Flush 0 DTLB ready for operation 1 DTLB flush request/status
PTBP	Page Table Base Pointer N 22-bit pointer to the base of page directory/table

Table 8-3. DMMUCR Field Descriptions

The PTBP field in the DMMUCR is required only in implementations with hardware PTE reload support. Implementations that use software TLB reload are not required to implement this field because the page table base pointer is stored in a TLB miss exception handler's variable.

The DTF is optional and when implemented it flushes entire DTLB.

## 8.4.2 Data MMU Protection Register (DMMUPR)

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

It defines 7 protection groups indexed by PPI fields in PTEs.

Bit	31-28	27	26	25	24
Identifier	Reserved	UWE7	URE7	SWE7	SRE7
Reset	X	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W

Bit	23	22	21	20	19	18	17	16
Identifier	UWE6	URE6	SWE6	SRE6	UWE5	URE5	SWE5	SRE5
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit	15	14	13	12	11	10	9	8
Identifier	UWE4	URE4	SWE4	SRE4	UWE3	URE3	SWE3	SRE3
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit	7	6	5	4	3	2	1	0
Identifier	UWE2	URE2	SWE2	SRE2	UWE1	URE1	SWE1	SRE1
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

SREx	Supervisor Read Enable x 0 Load operation in supervisor mode not permitted 1 Load operation in supervisor mode permitted
SWEx	Supervisor Write Enable x 0 Store operation in supervisor mode not permitted 1 Store operation in supervisor mode permitted
UREx	User Read Enable x 0 Load operation in user mode not permitted 1 Load operation in user mode permitted
UWEx	User Write Enable x 0 Store operation in user mode not permitted 1 Store operation in user mode permitted

**Table 8-4. DMMUPR Field Descriptions**

A DMMUPR is required only in implementations with hardware PTE reload support. Implementations that use software TLB reload are not required to implement this register;

instead a TLB miss handler should have a software variable as replacement for the DMMUPR and it should do a software look-up operation and set DTLBW<sub>y</sub>TR<sub>x</sub> protection bits accordingly.

### 8.4.3 Instruction MMU Control Register (IMMUCR)

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

It provides general control of the IMMU.

Bit	31-10	9-1	0
Identifier	PTBP	Reserved	ITF
Reset	0	X	0
R/W	R/W	R	R/W

ITF	ITLB Flush 0 ITLB ready for operation 1 ITLB flush request/status
PTBP	Page Table Base Pointer N 22-bit pointer to the base of page directory/table

**Table 8-5. IMMUCR Field Descriptions**

The PTBP field in xMMUCR is required only in implementations with hardware PTE reload support. Implementations that use software TLB reload are not required to implement this field because the page table base pointer is stored in a TLB miss exception handler's variable.

The ITF is optional and when implemented it flushes entire ITLB.

### 8.4.4 Instruction MMU Protection Register (IMMUPR)

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

It defines 7 protection groups indexed by PPI fields in PTEs.

Bit	31-14	13	12	11	10	9	8
Identifier	Reserved	UXE7	SXE7	UXE6	SXE6	UXE5	SXE5
Reset	X	0	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W	R/W	R/W

Bit	7	6	5	4	3	2	1	0
Identifier	UXE4	SXE4	UXE3	SXE3	UXE2	SXE2	UXE1	SXE1
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

SXEx	Supervisor Execute Enable x 0 Instruction fetch in supervisor mode not permitted 1 Instruction fetch in supervisor mode permitted
UXEx	User Execute Enable x 0 Instruction fetch in user mode not permitted 1 Instruction fetch in user mode permitted

Table 8-6. IMMUPR Field Descriptions

The IMMUPR is required only in implementations with hardware PTE reload support. Implementations that use software TLB reload are not required to implement this register; instead the TLB miss handler should have a software variable as replacement for the IMMUPR register and it should do a software look-up operation and set ITLBW<sub>y</sub>TR<sub>x</sub> protection bits accordingly.

### 8.4.5 Instruction/Data TLB Entry Invalidate Registers (xTLBEIR)

The instruction/data TLB entry invalidate registers are special-purpose registers accessible with the l.mtspr/l.mfspr instructions in supervisor mode. They are 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit implementation.

The xTLBEIR is written with the effective address. The corresponding xTLB entry is invalidated in the local processor.

Bit	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets TLB entry inside TLB
----	---

Table 8-7. xTLBEIR Field Descriptions

## 8.4.6 Instruction/Data Translation Lookaside Buffer Way y Match Registers (xTLBWyMR0-xTLBWyMR127)

The xTLBWyMR registers are 32-bit special-purpose supervisor-level registers accessible with the l.mtspr/l.mfspr instructions in supervisor mode.

Together with the xTLBWyTR registers they cache translation entries used for translating virtual to physical address. A virtual address is formed from the EA generated during instruction fetch or load/store operation, and the SR[CID] field. xTLBWyMR registers hold a tag that is compared with the current virtual address generated by the CPU core. Together with the xTLBWyTR registers and match logic they form a core part of the xMMU.

Bit	31-13
Identifier	VPN
Reset	X
R/W	R/W

Bit	12-8	7-6	5-2	1	0
Identifier	Reserved	LRU	CID	PL1	V
Reset	X	0	X	0	0
R/W	R	R/W	R/W	R/W	R/W

V	Valid 0 TLB entry invalid 1 TLB entry valid
PL1	Page Level 1 0 Page level is 2 1 Page level is 1
CID	Context ID 0-15 TLB entry translates for CID
LRU	Last Recently used 0-3 Index in LRU queue (lower the number, more recent access)
VPN	Virtual Page Number 0-N Number of the virtual frame that must match EA

**Table 8-8. xTLBMR Field Descriptions**

The CID bits can be hardwired to zero if the implementation does not support fast context switching and SR[CID] bits.



### 8.4.7 Data Translation Lookaside Buffer Way y Translate Registers (DTLBW<sub>y</sub>TR0-DTLBW<sub>y</sub>TR127)

The DTLBW<sub>y</sub>TR registers are 32-bit special-purpose supervisor-level registers accessible with the l.mtspr/l.mfspr instructions in supervisor mode.

Together with the DTLBW<sub>y</sub>MR registers they cache translation entries used for translating virtual to physical address. A virtual address is formed from the EA generated during a load/store operation, and the SR[CID] field. Together with the DTLBW<sub>y</sub>MR registers and match logic they form a core of the DMMU.

Bit	31-13	12-10	9	8	7
Identifier	PPN	Reserved	SWE	SRE	UWE
Reset	X	X	X	X	X
R/W	R/W	R	R/W	R/W	R/W

Bit	6	5	4	3	2	1	0
Identifier	URE	D	A	WOM	WBC	CI	CC
Reset	X	X	X	X	X	X	X
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

CC	Cache Coherency 0 Data cache coherency is not enforced for this page 1 Data cache coherency is enforced for this page
CI	Cache Inhibit 0 Cache is enabled for this page 1 Cache is disabled for this page
WBC	Write-Back Cache 0 Data cache uses write-through strategy for data from this page 1 Data cache uses write-back strategy for data from this page
WOM	Weakly-Ordered Memory 0 Strongly-ordered memory model for this page 1 Weakly-ordered memory model for this page
A	Accessed 0 Page was not accessed 1 Page was accessed
D	Dirty 0 Page was not modified 1 Page was modified

URE	User Read Enable x 0 Load operation in user mode not permitted 1 Load operation in user mode permitted
UWE	User Write Enable x 0 Store operation in user mode not permitted 1 Store operation in user mode permitted
SRE	Supervisor Read Enable x 0 Load operation in supervisor mode not permitted 1 Load operation in supervisor mode permitted
SWE	Supervisor Write Enable x 0 Store operation in supervisor mode not permitted 1 Store operation in supervisor mode permitted
PPN	Physical Page Number 0-N Number of the physical frame in memory

Table 8-9. DTLBTR Field Descriptions

## 8.4.8 Instruction Translation Lookaside Buffer Way y Translate Registers (ITLBWyTR0-ITLBWyTR127)

The ITLBWyTR registers are 32-bit special-purpose supervisor-level registers accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

Together with the ITLBWyMR registers they cache translation entries used for translating virtual to physical address. A virtual address is formed from the EA generated during an instruction fetch operation, and the `SR[CID]` field. Together with the ITLBWyMR registers and match logic they form a core part of the IMMU.

Bit	31-13	12-8	7
Identifier	PPN	Reserved	UXE
Reset	X	X	X
R/W	R/W	R/W	R/W

Bit	6	5	4	3	2	1	0
Identifier	SXE	D	A	WOM	WBC	CI	CC
Reset	X	X	X	X	X	X	X
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

CC	Cache Coherency 0 Data cache coherency is not enforced for this page 1 Data cache coherency is enforced for this page
----	---

CI	Cache Inhibit 0 Cache is enabled for this page 1 Cache is disabled for this page
WBC	Write-Back Cache 0 Data cache uses write-through strategy for data from this page 1 Data cache uses write-back strategy for data from this page
WOM	Weakly-Ordered Memory 0 Strongly-ordered memory model for this page 1 Weakly-ordered memory model for this page
A	Accessed 0 Page was not accessed 1 Page was accessed
D	Dirty 0 Page was not modified 1 Page was modified
SXE	Supervisor Execute Enable x 0 Instruction fetch operation in supervisor mode not permitted 1 Instruction fetch operation in supervisor mode permitted
UXE	User Execute Enable x 0 Instruction fetch operation in user mode not permitted 1 Instruction fetch operation in user mode permitted
PPN	Physical Page Number 0-N Number of the physical frame in memory

Table 8-10. ITLBWyTR Field Descriptions

## 8.4.9 Instruction/Data Area Translation Buffer Match Registers (xATBMR0-xATBMR3)

The xATBMR registers are 32-bit special-purpose supervisor-level registers accessible with the l.mtspr/l.mfspr instructions in supervisor mode.

Together with the xATBTR registers they cache translation entries used for translating virtual to physical address of large address space areas. A virtual address is formed from the EA generated during an instruction fetch or load/store operation, and the SR[CID] field. xATBMR registers hold a tag that is compared with the current virtual address generated by the CPU core. Together with the xATBTR registers and match logic they form a core part of the xMMU.

Bit	31-10
Identifier	VPN
Reset	X
R/W	R/W

Bit	9-5	5	4-1	0
Identifier	Reserved	PS	CID	V
Reset	X	0	0	0
R/W	R	R/W	R/W	R/W

V	Valid 0 TLB entry invalid 1 TLB entry valid
CID	Context ID 0-15 TLB entry translates for CID
PS	Page Size 0 16 Mbyte page 1 32 Gbyte page
VPN	Virtual Page Number 0-N Number of the virtual frame that must match EA

Table 8-11. xATBMR Field Descriptions

The CID bits can be hardwired to zero if the implementation does not support fast context switching and SR[CID] bits.

## 8.4.10 Data Area Translation Buffer Translate Registers (DATBTR0-DATBTR3)

The DATBTR registers are 32-bit special-purpose supervisor-level registers accessible with the l.mtspr/l.mfspr instructions in supervisor mode.

Together with the DATBMR registers they cache translation entries used for translating virtual to physical address. A virtual address is formed from the EA generated during a load/store operation, and the SR[CID] field. Together with the DATBMR registers and match logic they form a core part of the DMMU.

Bit	31-10	9	8	7
Identifier	PPN	UWE	URE	SWE
Reset	X	X	X	X
R/W	R/W	R/W	R/W	R/W

Bit	6	5	4	3	2	1	0
Identifier	SRE	D	A	WOM	WBC	CI	CC
Reset	X	X	X	X	X	X	X
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

CC	Cache Coherency 0 Data cache coherency is not enforced for this page 1 Data cache coherency is enforced for this page
CI	Cache Inhibit 0 Cache is enabled for this page 1 Cache is disabled for this page
WBC	Write-Back Cache 0 Data cache uses write-through strategy for data from this page 1 Data cache uses write-back strategy for data from this page
WOM	Weakly-Ordered Memory 0 Strongly-ordered memory model for this page 1 Weakly-ordered memory model for this page
A	Accessed 0 Page was not accessed 1 Page was accessed
D	Dirty 0 Page was not modified 1 Page was modified
SRE	Supervisor Read Enable x 0 Load operation in supervisor mode not permitted 1 Load operation in supervisor mode permitted
SWE	Supervisor Write Enable x 0 Store operation in supervisor mode not permitted 1 Store operation in supervisor mode permitted
URE	User Read Enable x 0 Load operation in user mode not permitted 1 Load operation in user mode permitted
UWE	User Write Enable x 0 Store operation in user mode not permitted 1 Store operation in user mode permitted
PPN	Physical Page Number 0-N Number of the physical frame in memory

Table 8-12. DATBTR Field Descriptions

### 8.4.11 Instruction Area Translation Buffer Translate Registers (IATBTR0-IATBTR3)

The IATBTR registers are 32-bit special-purpose supervisor-level registers accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

Together with the IATBMR registers they cache translation entries used for translating virtual to physical address. A virtual address is formed from the EA generated during an instruction fetch operation, and the `SR[CID]` field. Together with the IATBMR registers and match logic they form a core part of the IMMU.

Bit	31-10	9-8	7
Identifier	PPN	Reserved	UXE
Reset	X	X	X
R/W	R/W	R/W	R/W

Bit	6	5	4	3	2	1	0
Identifier	SXE	D	A	WOM	WBC	CI	CC
Reset	X	X	X	X	X	X	X
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

CC	Cache Coherency 0 Data cache coherency is not enforced for this page 1 Data cache coherency is enforced for this page
CI	Cache Inhibit 0 Cache is enabled for this page 1 Cache is disabled for this page
WBC	Write-Back Cache 0 Data cache uses write-through strategy for data from this page 1 Data cache uses write-back strategy for data from this page
WOM	Weakly-Ordered Memory 0 Strongly-ordered memory model for this page 1 Weakly-ordered memory model for this page
A	Accessed 0 Page was not accessed 1 Page was accessed
D	Dirty 0 Page was not modified 1 Page was modified
SXE	Supervisor Execute Enable x 0 Instruction fetch operation in supervisor mode not permitted 1 Instruction fetch operation in supervisor mode permitted
UXE	User Execute Enable x 0 Instruction fetch operation in user mode not permitted 1 Instruction fetch operation in user mode permitted
PPN	Physical Page Number 0-N Number of the physical frame in memory

Table 8-13. IATBTR Field Descriptions

## 8.5 Address Translation Mechanism in 32-bit Implementations

Memory in an OpenRISC 1000 implementation with 32-bit effective addresses (EA) is divided into level 1 and level 2 pages. Translation is therefore based on two-level page table. However for virtual memory areas that do not need the smallest 8KB page granularity, only one level can be used.

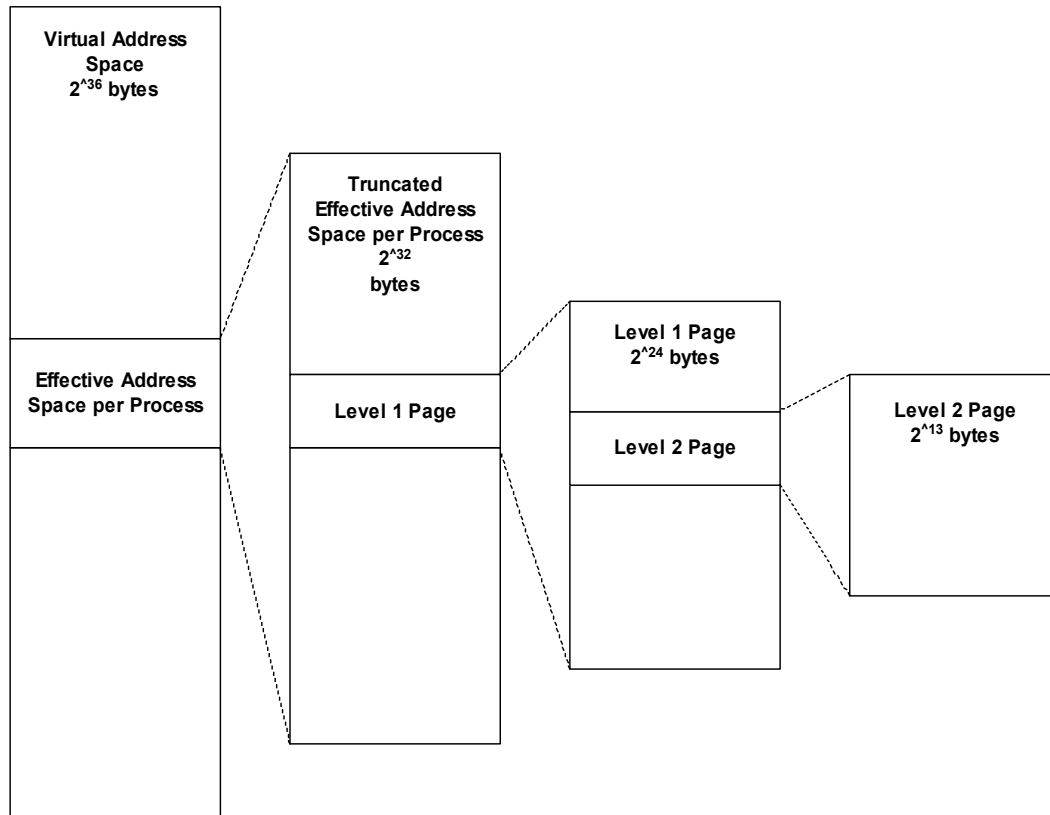
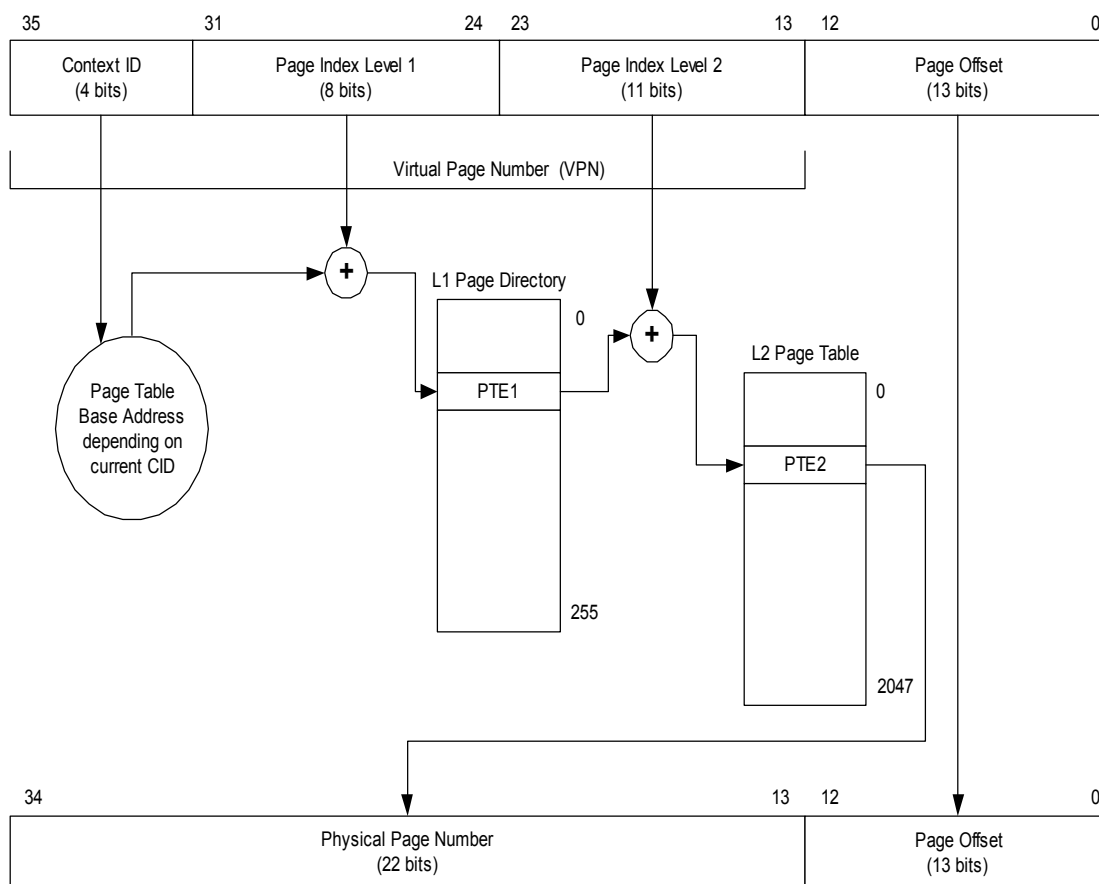


Figure 8-2. Memory Divided Into L1 and L2 pages

The first step in page address translation is to append the current SR[CID] bits as most significant bits to the 32-bit effective address, combining them into a 36-bit virtual address. This virtual address is then used to locate the correct page table entry (PTE) in the page tables in the memory. The physical page number is then extracted from the PTE and used in the physical address. Note that for increased performance, most processors implement on-chip translation lookaside buffers (TLBs) to cache copies of the recently-used PTEs.



**Figure 8-3. Address Translation Mechanism using Two-Level Page Table**

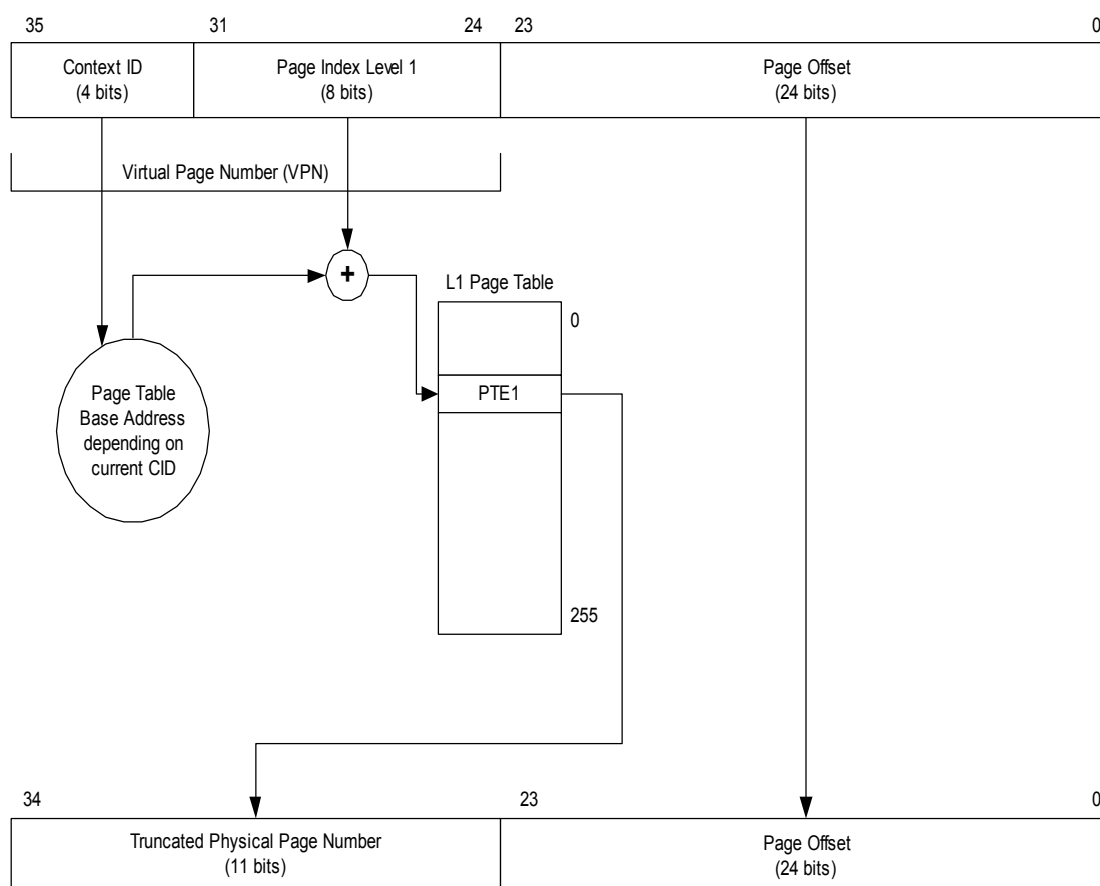
8-3 shows an overview of the two-level page table translation of a virtual address to a physical address:

- ✓ Bits 35..32 of the virtual address select the page tables for the current context (process)
- ✓ Bits 31..24 of the virtual address correspond to the level 1 page number within the current context's virtual space. The L1 page index is used to index the L1 page directory and to retrieve the PTE from it, or together with the L2 page index to match for the PTE in on-chip TLBs.
- ✓ Bits 23..13 of the virtual address correspond to the level 2 page number within the current context's virtual space. The L2 page index is used to index the L2 page table and to retrieve the PTE from it, or together with the L1 page index to match for the PTE in on-chip TLBs.
- ✓ Bits 12..0 of the virtual address are the byte offset within the page; these are concatenated with the PPN field of the PTE to form the physical address used to access memory

The OpenRISC 1000 two-level page table translation also allows implementation of segments with only one level of translation. This greatly reduces memory requirements



for the page tables since large areas of unused virtual address space can be covered only by level 1 PTEs.



**Figure 8-4. Address Translation Mechanism using only L1 Page Table**

8-4 shows an overview of the one-level page table translation of a virtual address to physical address:

- ✓ Bits 35..32 of the virtual address select the page tables for the current context (process)
- ✓ Bits 31..24 of the virtual address correspond to the level 1 page number within the current context's virtual space. The L1 page index is used to index the L1 page table and to retrieve the PTE from it, or to match for the PTE in on-chip TLBs.
- ✓ Bits 23..0 of the virtual address are the byte offset within the page; these are concatenated with the truncated PPN field of the PTE to form the physical address used to access memory

## 8.6 Address Translation Mechanism in 64-bit Implementations

Memory in OpenRISC 1000 implementations with 64-bit effective addresses (EA) is divided into level 0, level 1 and level 2 pages. Translation is therefore based on three-level page table. However for virtual memory areas that do not need the smallest page granularity of 8KB, two level translation can be used.

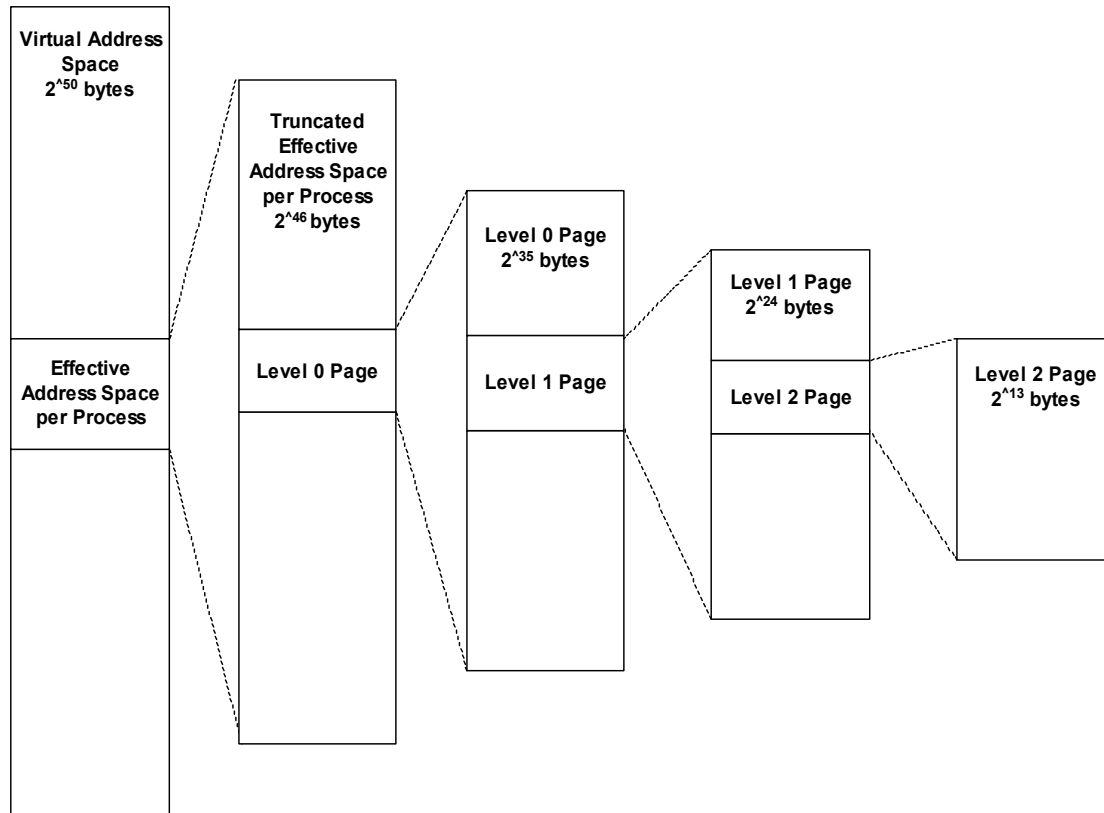
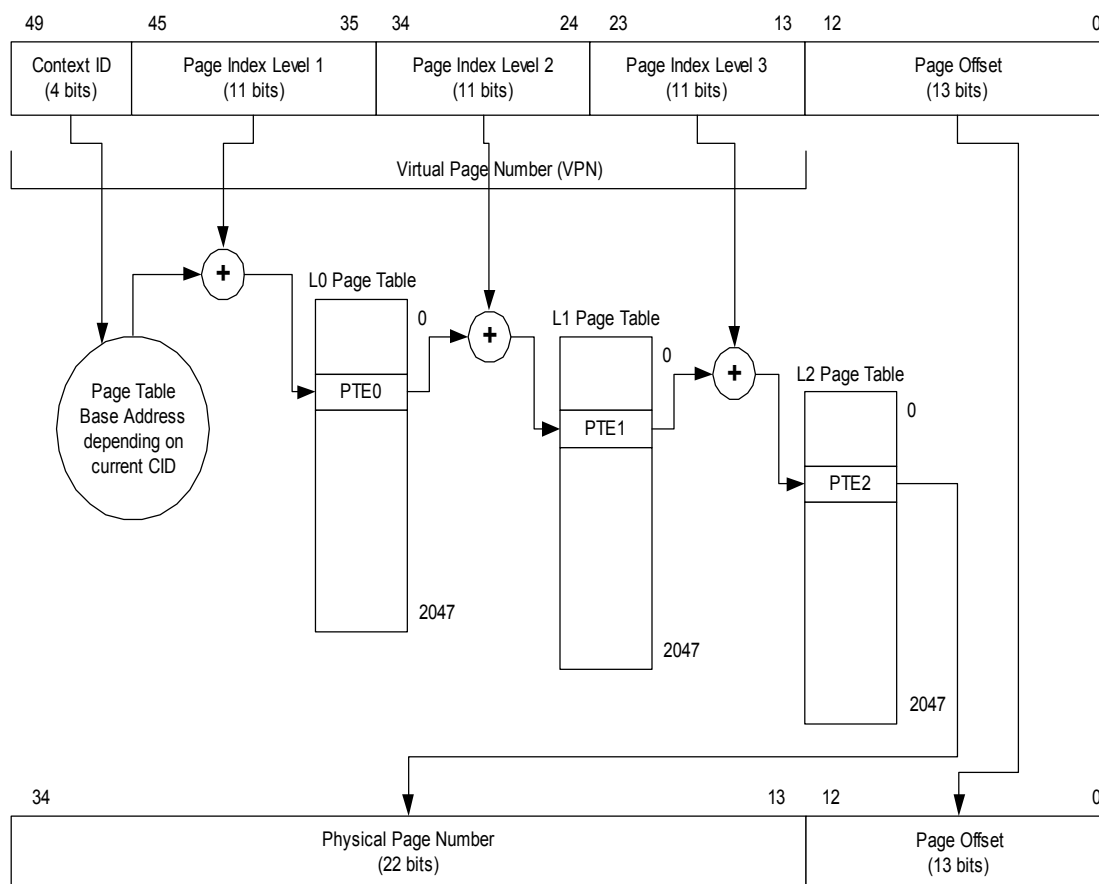


Figure 8-5. Memory Divided Into L0, L1 and L2 pages

The first step in page address translation is truncation of the 64-bit effective address into a 46-bit address. Then the current SR[CID] bits are appended as most significant bits. The 50-bit virtual address thus formed is then used to locate the correct page table entry (PTE) in the page tables in the memory. The physical page number is then extracted from the PTE and used in the physical address. Note that for increased performance, most processors implement on-chip translation lookaside buffers (TLBs) to cache copies of the recently-used PTEs.



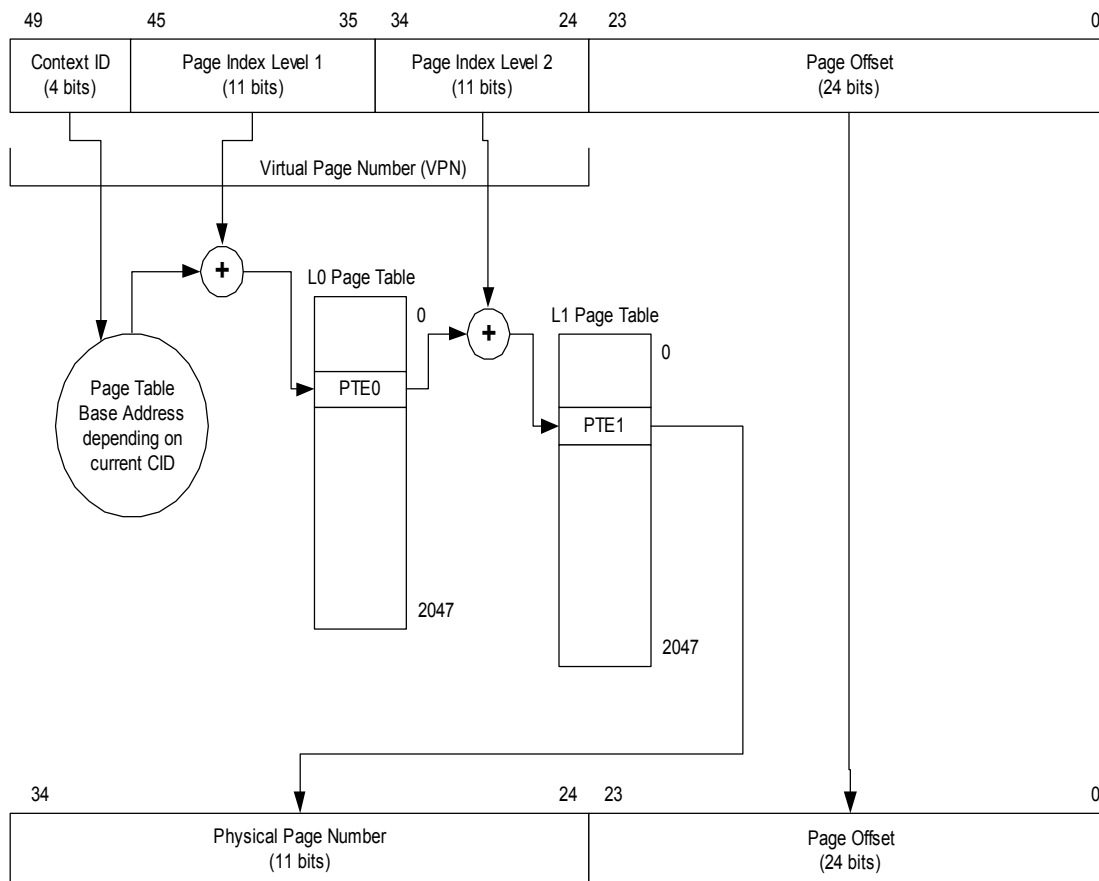
**Figure 8-6. Address Translation Mechanism using Three-Level Page Table**

8-6 shows an overview of the three-level page table translation of a virtual address to physical address:

- ✓ Bits 49..46 of the virtual address select the page tables for the current context (process)
- ✓ Bits 45..35 of the virtual address correspond to the level 0 page number within current context's virtual space. The L0 page index is used to index the L0 page directory and to retrieve the PTE from it, or together with the L1 and L2 page indexes to match for the PTE in on-chip TLBs.
- ✓ Bits 34..24 of the virtual address correspond to the level 1 page number within the current context's virtual space. The L1 page index is used to index the L1 page directory and to retrieve the PTE from it, or together with the L0 and L2 page indexes to match for the PTE in on-chip TLBs.
- ✓ Bits 23..13 of the virtual address correspond to the level 2 page number within the current context's virtual space. The L2 page index is used to index the L2 page table and to retrieve the PTE from it, or together with the L0 and L1 page indexes to match for the PTE in on-chip TLBs.

- ✓ Bits 12..0 of the virtual address are the byte offset within the page; these are concatenated with the truncated PPN field of the PTE to form the physical address used to access memory

The OpenRISC 1000 three-level page table translation also allows implementation of large segments with two levels of translation. This greatly reduces memory requirements for the page tables since large areas of unused virtual address space can be covered only by level 1 PTEs.



**Figure 8-7. Address Translation Mechanism using Two-Level Page Table**

8-7 shows an overview of the two-level page table translation of a virtual address to physical address:

- ✓ Bits 49..46 of the virtual address select the page tables for the current context (process)
- ✓ Bits 45..35 of the virtual address correspond to the level 0 page number within the current context's virtual space. The L0 page index is used to index the L0 page directory and to retrieve the PTE from it, or together with the L1 page index to match for the PTE in on-chip TLBs.

- ✓ Bits 34..24 of the virtual address correspond to the level 1 page number within the current context's virtual space. The L1 page index is used to index the L1 page table and to retrieve the PTE from it, or together with the L0 page index to match for the PTE in on-chip TLBs.
- ✓ Bits 23..0 of the virtual address are the byte offset within the page; these are concatenated with the truncated PPN field of the PTE to form the physical address used to access memory

## 8.7 Memory Protection Mechanism

After a virtual address is determined to be within a page covered by the valid PTE, the access is validated by the memory protection mechanism. If this protection mechanism prohibits the access, a page fault exception is generated.

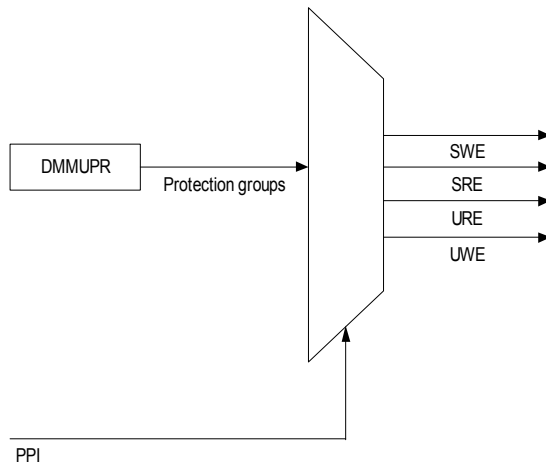
The memory protection mechanism allows selectively granting read access, write access or execute access for both supervisor and user modes. The page protection mechanism provides protection at all page level granularities.

Protection attribute	Meaning
DMMUPR[SREx]	Enable load operations in supervisor mode to the page.
DMMUPR[SWEx]	Enable store operations in supervisor mode to the page.
IMMUPR[SXEx]	Enable execution in supervisor mode of the page.
DMMUPR[UREx]	Enable load operations in user mode to the page.
DMMUPR[UWEx]	Enable store operations in user mode to the page.
IMMUPR[UXEx]	Enable execution in user mode of the page.

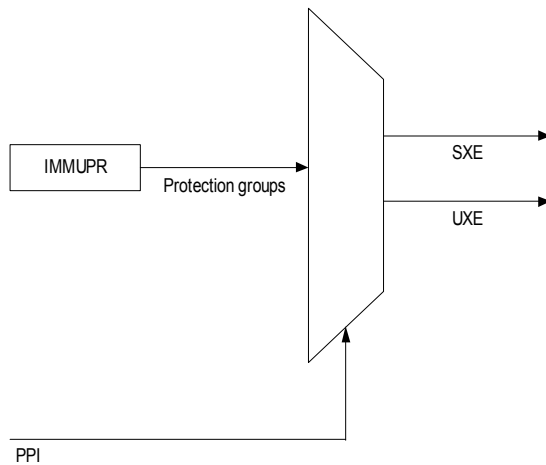
**Table 8-14. Protection Attributes**

8-14 lists page protection attributes defined in MMU protection registers. For the individual page the appropriate strategy out of seven possible strategies programmed in MMU protection registers is selected with the PPI field of the PTE.

In OpenRISC 1000 processors that do not implement TLB/ATB reload in hardware, protection registers are not needed.



**Figure 8-8. Selection of Page Protection Attributes for Data Accesses**

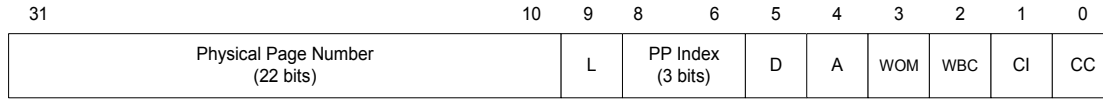


**Figure 8-9. Selection of Page Protection Attributes for Instruction Fetch Accesses**

## 8.8 Page Table Entry Definition

Page table entries (PTEs) are generated and placed in page tables in memory by the operating system. A PTE is 32 bits wide and is the same for 32-bit and 64-bit OpenRISC 1000 processor implementations.

A PTE translates a virtual memory area into a physical memory area. How much virtual memory is translated depends on which level the PTE resides. PTEs are either in page directories with L bit zeroed or in page tables with L bit set. PTEs in page directories point to next level page directory or to final page table that contains PTEs for actual address translation.

**Figure 8-10. Page Table Entry Format**

CC	Cache Coherency 0 Data cache coherency is not enforced for this page 1 Data cache coherency is enforced for this page
CI	Cache Inhibit 0 Cache is enabled for this page 1 Cache is disabled for this page
WBC	Write-Back Cache 0 Data cache uses write-through strategy for data from this page 1 Data cache uses write-back strategy for data from this page
WOM	Weakly-Ordered Memory 0 Strongly-ordered memory model for this page 1 Weakly-ordered memory model for this page
A	Accessed 0 Page was not accessed 1 Page was accessed
D	Dirty 0 Page was not modified 1 Page was modified
PPI	Page Protection Index 0 PTE is invalid 1-7 Selects a group of six bits from a set of seven protection attribute groups in xMMUCR
L	Last 0 PTE from page directory pointing to next page directory/table 1 Last PTE in a linked form of PTEs (describing the actual page)
PPN	Physical Page Number 0-N Number of the physical frame in memory

**Table 8-15. PTE Field Descriptions**

## 8.9 Page Table Search Operation

An implementation may choose to implement the page table search operation in either hardware or software. For all page table search operations data addresses are untranslated (i.e. the effective and physical base address of the page table are the same).

When implemented in software, two TLB miss exceptions are used to handle TLB reload operations. Also, the software is responsible for maintaining accessed and dirty bits in the page tables.

## 8.10 Page History Recording

The accessed (A) and dirty (D) bits reside in each PTE and keep information about the history of the page. The operating system uses this information to determine which areas of the main memory to swap to the disk and which areas of the memory to load back to the main memory (demand-paging).

The accessed (A) bit resides both in the PTE in page table and in the copy of PTE in the TLB. Each time the page is accessed by a load, store or instruction fetch operation, the accessed bit is set.

If the TLB reload is performed in software, then the software must also write back the accessed bit from the TLB to the page table.

In cases when access operation to the page fails, it is not defined whether the accessed bit should be set or not. Since the accessed bit is merely a hint to the operating system, it is up to the implementation to decide.

It is up to the operating system to determine when to explicitly clear the accessed bit for a given page.

The dirty (D) bit resides in both the PTE in page table and in the copy of PTE in the TLB. Each time the page is modified by a store operation, the dirty bit is set.

If TLB reload is performed in software, then the software must also write back the dirty bit from the TLB to the page table.

In cases when access operation to the page fails, it is not defined whether the dirty bit should be set or not. Since the dirty bit is merely a hint to the operating system, it is up to the implementation to decide. However implementation or TLB reload software must check whether page is actually writable before setting the dirty bit.

It is up to the operating system to determine when to explicitly clear the dirty bit for a given page.

## 8.11 Page Table Updates

Updates to the page tables include operations like adding a PTE, deleting a PTE and modifying a PTE. On multiprocessor systems exclusive access to the page table must be assured before it is modified.

TLBs are noncoherent caches of the page tables and must be maintained accordingly. Explicit software synchronization between TLB and page tables is required so that page tables and TLBs remain coherent.

Since the processor reloads PTEs even during updates of the page table, special care must be taken when updating page tables so that the processor does not accidentally use half modified page table entries.



## 9 Cache Model & Cache Coherency

This chapter describes the OpenRISC 1000 cache model and architectural control to maintain cache coherency in multiprocessor environment.

Note that this chapter describes the cache model and cache coherency mechanism from the perspective of the programming model. As such, it describes the cache management principles, the cache coherency mechanisms and the cache control registers. The hardware implementation details that are invisible to the OpenRISC 1000 programming model, such as cache organization and size, are not contained in the architectural definition.

The function of the cache management registers depends on the implementation of the cache(s) and the setting of the memory/cache access attributes. For a program to execute properly on all OpenRISC 1000 processor implementations, software should assume a Harvard cache model. In cases where a processor is implemented without a cache, the architecture guarantees that writing to cache registers will not halt execution. For example a processor without cache should simply ignore writes to cache management registers. A processor with a Stanford cache model should simply ignore writes to instruction cache management registers. In this manner, programs written for separate instruction and data caches will run on all compliant implementations.

### 9.1 Cache Special-Purpose Registers

9-1 summarizes the registers that the operating system uses to manage the cache(s).

For implementations that have unified cache, registers that control the data and instruction caches are merged and available at the same time both as data and instruction cache registers.

GRP #	REG #	REG NAME	USER MODE	SUPV MODE	DESCRIPTION
3	0	DCCR	–	R/W	Data Cache Control Register
3	1	DCBPR	W	W	Data Cache Block Prefetch Register
3	2	DCBFR	W	W	Data Cache Block Flush Register
3	3	DCBIR	–	W	Data Cache Block Invalidate Register
3	4	DCBWR	W	W	Data Cache Block Write-back Register
3	5	DCBLR	-	W	Data Cache Block Lock Register
4	0	ICCR	–	R/W	Instruction Cache Control Register
4	1	ICBPR	W	W	Instruction Cache Block PreFetch Register

GRP #	REG #	REG NAME	USER MODE	SUPV MODE	DESCRIPTION
4	2	ICBIR	W	W	Instruction Cache Block Invalidate Register
4	3	ICBLR	-	W	Instruction Cache Block Lock Register

Table 9-1. Cache Registers

### 9.1.1 Data Cache Control Register

The data cache control register is a 32-bit special-purpose register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

The DCCR controls the operation of the data cache.

Bit	31-8	7-0
Identifier	Reserved	EW
Reset	X	0
R/W	R	R/W

EW	Enable Ways 0000 0000 All ways disabled/locked ... 1111 1111 All ways enabled/unlocked
----	---

Table 9-2. DCCR Field Descriptions

If data cache does not implement way locking, the DCCR is not required to be implemented.

### 9.1.2 Instruction Cache Control Register

The instruction cache control register is a 32-bit special-purpose register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

The ICCR controls the operation of the instruction cache.

Bit	31-8	7-0
Identifier	Reserved	EW
Reset	X	0
R/W	R	R/W

EW	Enable Ways
	0000 0000 All ways disabled/locked
	1111 1111 All ways enabled/unlocked

**Table 9-3. ICCR Field Descriptions**

If the instruction cache does not implement way locking, the ICCR is not required to be implemented.

## 9.2 Cache Management

This section describes special-purpose cache management registers for both data and instruction caches.

Memory accesses caused by cache management are not recorded (unlike load or store instructions) and cannot invoke any exception.

Instruction caches do not need to be coherent with the memory or caches of other processors. Software must make the instruction cache coherent with modified instructions in the memory. A typical way to accomplish this is:

1. Data cache block write-back (update of the memory)
2. l.csync (wait for update to finish)
3. Instruction cache block invalidate (clear instruction cache block)
4. Flush pipeline

### 9.2.1 Data Cache Block Prefetch (Optional)

The data cache block prefetch register is an optional special-purpose register accessible with the l.mtspr/l.mfspr instructions in both user and supervisor modes. It is 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit implementations. An implementation may choose not to implement this register and ignore all writes to this register.

The DCBPR is written with the effective address and the corresponding block from memory is prefetched into the cache. Memory accesses are not recorded (unlike load or store instructions) and cannot invoke any exception.

A data cache block prefetch is used strictly for improving performance.

Bit	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block
----	--

**Table 9-4. DCBPR Field Descriptions**

### 9.2.2 Data Cache Block Flush

The data cache block flush register is a special-purpose register accessible with the `l.mtspr/l.mfspr` instructions in both user and supervisor modes. It is 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit implementations.

The DCBFR is written with the effective address. If coherency is required then the corresponding:

- ✓ Unmodified data cache block is invalidated in all processors.
- ✓ Modified data cache block is written back to the memory and invalidated in all processors.
- ✓ Missing data cache block in the local processor causes that modified data cache block in other processor is written back to the memory and invalidated. If other processors have unmodified data cache block, it is just invalidated in all processors.

If coherency is not required then the corresponding:

- ✓ Unmodified data cache block in the local processor is invalidated.
- ✓ Modified data cache block is written back to the memory and invalidated in local processor.
- ✓ Missing cache block in the local processor does not cause any action.

Bit	31-0
Identifier	EA
Reset	0
R/W	Write only

EA	Effective Address EA that targets byte inside cache block
----	--

**Table 9-5. DCBFR Field Descriptions**

### 9.2.3 Data Cache Block Invalidate

The data cache block invalidate register is a special-purpose register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode. It is 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit implementations.

The DCBIR is written with the effective address. If coherency is required then the corresponding:

- ✓ Unmodified data cache block is invalidated in all processors.
- ✓ Modified data cache block is invalidated in all processors.
- ✓ Missing data cache block in the local processor causes that data cache blocks in other processors are invalidated.

If coherency is not required then corresponding:

- ✓ Unmodified data cache block in the local processor is invalidated.
- ✓ Modified data cache block in the local processor is invalidated.
- ✓ Missing cache block in the local processor does not cause any action.

Bit	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block
----	--

**Table 9-6. DCBIR Field Descriptions**

## 9.2.4 Data Cache Block Write-Back

The data cache block write-back register is a special-purpose register accessible with the `l.mtspr/l.mfspr` instructions in both user and supervisor modes. It is 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit implementations.

The DCBWR is written with the effective address. If coherency is required then the corresponding data cache block in any of the processors is written back to memory if it was modified. If coherency is not required then the corresponding data cache block in the local processor is written back to memory if it was modified.

Bit	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block
----	--

**Table 9-7. DCBWR Field Descriptions**

## 9.2.5 Data Cache Block Lock (Optional)

The data cache block lock register is an optional special-purpose register accessible with the `l.mtspr/l.mfspr` instructions in both user and supervisor modes. It is 32 bits wide in a 32-bit implementation and 64 bits wide in a 64-bit implementation.

The DCBLR is written with the effective address. The corresponding data cache block in the local processor is locked.

If all blocks of the same set in all cache ways are locked, then the cache refill may automatically unlock the least-recently used block.

Bit	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block
----	--

**Table 9-8. DCBLR Field Descriptions**

### 9.2.6 Instruction Cache Block Prefetch (Optional)

The instruction cache block prefetch register is an optional special-purpose register accessible with the `l.mtspr/l.mfspr` instructions in both user and supervisor modes. It is 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit implementations. An implementation may choose not to implement this register and ignore all writes to this register.

The ICBPR is written with the effective address and the corresponding block from memory is prefetched into the instruction cache.

Instruction cache block prefetch is used strictly for improving performance.

Bit	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block
----	--

**Table 9-9. ICBPR Field Descriptions**

### 9.2.7 Instruction Cache Block Invalidate

The instruction cache block invalidate register is a special-purpose register accessible with the `l.mtspr/l.mfspr` instructions in both user and supervisor modes. It is 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit implementations.

The ICBIR is written with the effective address. If coherency is required then the corresponding instruction cache blocks in all processors are invalidated. If coherency is not required then the corresponding instruction cache block is invalidated in the local processor.

Bit	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block
----	--

**Table 9-10. ICBIR Field Descriptions**

## 9.2.8 Instruction Cache Block Lock (Optional)

The instruction cache block lock register is an optional special-purpose register accessible with the l.mtspr/l.mfspr instructions in both user and supervisor modes. It is 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit implementations.

The ICBLR is written with the effective address. The corresponding instruction cache block in the local processor is locked.

If all blocks of the same set in all cache ways are locked, then the cache refill may automatically unlock the least-recently used block.

Missing cache block in the local processor does not cause any action.

Bit	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block
----	--

**Table 9-11. ICBLR Field Descriptions**

## 9.3 Cache/Memory Coherency

The primary role of the cache coherency system is to synchronize cache content with other caches and with the memory and to provide the same image of the memory to all devices using the memory.

The architecture provides several features to implement cache coherency. In systems that do not provide cache coherency with the PTE attributes (because they do not implement a memory management unit), it may be provided through explicit cache management.

Cache coherency in systems with virtual memory can be provided on a page-by-page basis with PTE attributes. The attributes are:

- ✓ Cache Coherent (CC Attribute)
- ✓ Caching-Inhibited (CI Attribute)
- ✓ Write-Back Cache (WBC Attribute)

When the memory/cache attributes are changed, it is imperative that the cache contents should reflect the new attribute settings. This usually means that cache blocks must be flushed or invalidated.

### 9.3.1 Pages Designated as Cache Coherent Pages

This attribute improves performance of the systems where cache coherency is performed with hardware and is relatively slow. Memory pages that do not need cache coherency are marked with CC=0 and only memory pages that need cache coherency are marked with CC=1. When an access to shared resource is made, the local processor will assert some kind of cache coherency signal and other processors will respond if they have a copy of the target location in their caches.

To improve performance of uniprocessor systems, memory pages should not be designated as CC=1.

### 9.3.2 Pages Designated as Caching-Inhibited Pages

Memory accesses to memory pages designated with CI=1 are always performed directly into the main memory, bypassing all caches. Memory pages designated with CI=1 are not loaded into the cache and the target content should never be available in the cache. To prevent any accident copy of the target location in the cache, whenever the operating system sets a memory page to be caching-inhibited, it should flush the corresponding cache blocks.

Multiple accesses may be merged into combined accesses except when individual accesses are separated by **l.msync** or **l.csync** or **l.psync**.

### 9.3.3 Pages Designated as Write-Back Cache Pages

Store accesses to memory pages designated with WBC=0 are performed both in data cache and memory. If a system uses multilevel hierarchy caches, a store must be performed to at least the depth in the memory hierarchy seen by other processors and devices.



Multiple stores may be merged into combined stores except when individual stores are separated by **l.msync** or **l.sync** or **l.psync**. A store operation may cause any part of the cache block to be written back to main memory.

Store accesses to memory pages designated with WBC=1 are performed only to the local data cache. Data from the local data cache can be copied to other caches and to main memory when copy-back operation is required. WBC=1 improves system performance, however it requires cache snooping hardware support in data cache controllers to guarantee cache coherency.

# 10 Multicore Support

This chapter describes the OpenRISC 1000 support for multicore system configurations. This section is targeted at hardware integrators and operating system designers.

## 10.1 Introduction

Multicore support is made possible by architecture facilities which include:

- Atomic memory operations as described in the Atomicity section
- Cache Coherency between multiple cores as described in the Cache/Memory Coherency section
- Core Identification registers to identify which processor is running

For a CPU architecture these features should be enough, but for a system design there are some additional considerations that need to be made. This chapter introduces some suggestions for OpenRISC multicore architectures to handle:

- Inter processor communication
- Multicore bootstrapping
- Timer Synchronization

## 10.2 Inter Processor Communication

In a multicore configuration each processor needs a way to communicate with other processors. This is needed for message sending and interrupt balancing. In OpenRISC each core has a full interrupt controller with 32 interrupt lines as described in Programmable Interrupt Controller (Optional). In multicore configurations the internal PIC is leveraged by routing all interrupts to all cores. The Open Multi-Processor Interrupt Controller (OMPIC) is a memory mapped programmable interrupt source providing a mechanism for Inter Processor Interrupts (IPI) enabling message sending and interrupt balancing.

The OMPIC supports up to 8192 cores via 13 bit **DST\_CORE** addressing field.

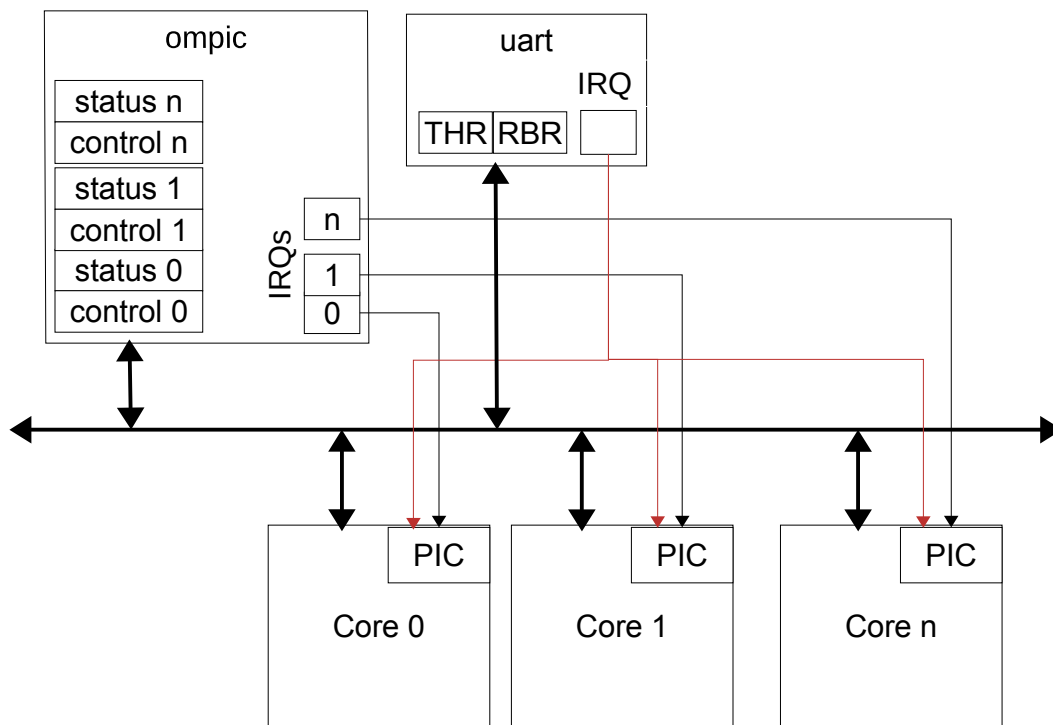


Figure 10-1: Multicore Interconnect with OMPIC

The above 10-1 shows a multicore system connected with OMPIC. The example uart device is connected to each core. The uart interrupt would typically be masked in all but one core.

Each core communicates with other cores by writing requests to its own designated **control** register specifying the destination core in **DST\_CORE**. When the **control** register is written to with the **IRQ\_GEN** field asserted the associated IRQ line will be raised to signal the destination core of a pending message. In the destination core's interrupt handler it shall write to its own **control** register with **IRQ\_ACK** asserted to clear the interrupt. It will then read its own **status** register to receive the data message. It is important to ack the IRQ then read the status register in this order to ensure messages are not lost.

### 10.2.1 OMPIC Control Registers

The OMPIC control registers are registers which are written to to send messages to another core or to Acknowledge interrupts. Cores will typically only write to their own control register. It is not typically useful to read from the control register

The status registers are located at memory mapped addresses  $\$OMPIC\_BASE + (\text{Core ID}) * 8$ . For example:

- $\$OMPIC\_BASE + 0x0$
- $\$OMPIC\_BASE + 0x8$

- \$OMPIC\_BASE + 0x10

Bit	31	30	29-16	15-0
Identifier	IRQ_ACK	IRQ_GEN	DST_CORE	DATA
Reset	0	0	0	0
R/W	W	W	W	W

IRQ_ACK	IRQ Acknowledge If asserted, clears the IRQ of the core associated with this control register.
IRQ_GEN	IRQ Generate If asserted, raises the IRQ of the core designated by <b>DST_CORE</b> .
DST_CORE	Destination Core The core to perform the operation on.
DATA	Data The data to send to the destination core.

Table 10-1. OMPIC Control Field Descriptions

## 10.2.2 OMPIC Status Registers

The OMPIC status registers are read only registers which are updated upon writes to control registers.

The status registers are located at memory mapped addresses  $\$OMPIC\_BASE + ((Core\ ID) * 8) + 4$ . For example:

- \$OMPIC\_BASE + 0x4
- \$OMPIC\_BASE + 0xc
- \$OMPIC\_BASE + 0x14

Bit	31	30	29-16	15-0
Identifier	Reserved	IRQ_PEND	SRC_CORE	DATA
Reset	X	0	0	0
R/W	-	R	R	R

IRQ_PEND	IRQ Pending Signals that the IRQ for this core is pending to be serviced.
SRC	Source Core The core that sent the last message to this core.
DATA	Data The pending data to be received for this core.

Table 10-2. OMPIC Status Field Descriptions

## 10.3 Temporary Storage

During exception handling it is often required to temporarily store register values before the exception stack frame is initialized or even in the case that no stack is required. It is recommended to use Shadow Registers for this temporary storage mechanism. For example:

```
#define SPR_GPR_BASE      (0 + 1024)
#define SPR_SHADOW_GPR(x) ((x) + SPR_GPR_BASE + 32)

l.mtspr r0,r5,SPR_SHADOW_GPR(5)
.. handle exception ..
l.mfspr r5,r0,SPR_SHADOW_GPR(5)
l.rfe
```

Note, if this method is used it means that fast context switching cannot be used with the multicore system.

## 10.4 Multicore bootstrapping

When booting a multicore OpenRISC system, upon reset all cores will begin execution at the reset vector. It is recommended that the core 0, the primary core, performs all hardware initialization and signals the secondary cores to initialize.

The secondary cores should wait to initialize until a signal is received from the primary core. Secondary cores can wait during to initialize by either spinning waiting for the initialization signal or by engaging the Power Management Doze mode and waiting for an interrupt.

The initialization signal is typically a variable stored in memory. The variable initially will be 0, the primary core will set it to the id of the core to initialize signaling each core to boot one by one.

## 10.5 Timer Synchronization

When running a multicore OpenRISC system it is typically useful for the Tick Timers of all cores to be synchronized. This is not guaranteed as processor reset and timer enablement may not have been triggered at the same time. To synchronize the Tick Timer it is recommended to either provide an external global timer device or use a software synchronization routine.

# 11 Debug Unit (Optional)

This chapter describes the OpenRISC 1000 debug facility. The debug unit assists software developers in debugging their systems. It provides support for watchpoints, breakpoints and program-flow control registers.

Watchpoints and breakpoint are events triggered by program- or data-flow matching the conditions programmed in the debug registers. Watchpoints do not interfere with the execution of the program-flow except indirectly when they cause a breakpoint. Watchpoints can be counted by Performance Counters Unit.

Breakpoint, unlike watchpoints, also suspends execution of the current program-flow and start trap exception processing. Breakpoint is optional consequence of watchpoints.

## 11.1 Features

The OpenRISC 1000 architecture defines eight sets of debug registers. Additional debug register sets can be defined by the implementation itself. The debug unit is optional and the presence of an implementation is indicated by the UPR[DUP] bit.

- ✓ Optional implementation
- ✓ Eight architecture defined sets of debug value/compare registers
- ✓ Match signed/unsigned conditions on instruction fetch EA, load/store EA and load/store data
- ✓ Combining match conditions for complex watchpoints
- ✓ Watchpoints can be counted by Performance Counters Unit
- ✓ Watchpoints can generate a breakpoint (trap exception)
- ✓ Counting watchpoints for generation of additional watchpoints

DVR/DCR pairs are used to compare instruction fetch or load/store EA and load/store data to the value stored in DVRs. Matches can be combined into more complex matches and used for generation of watchpoints. Watchpoints can be counted and reported as breakpoint.

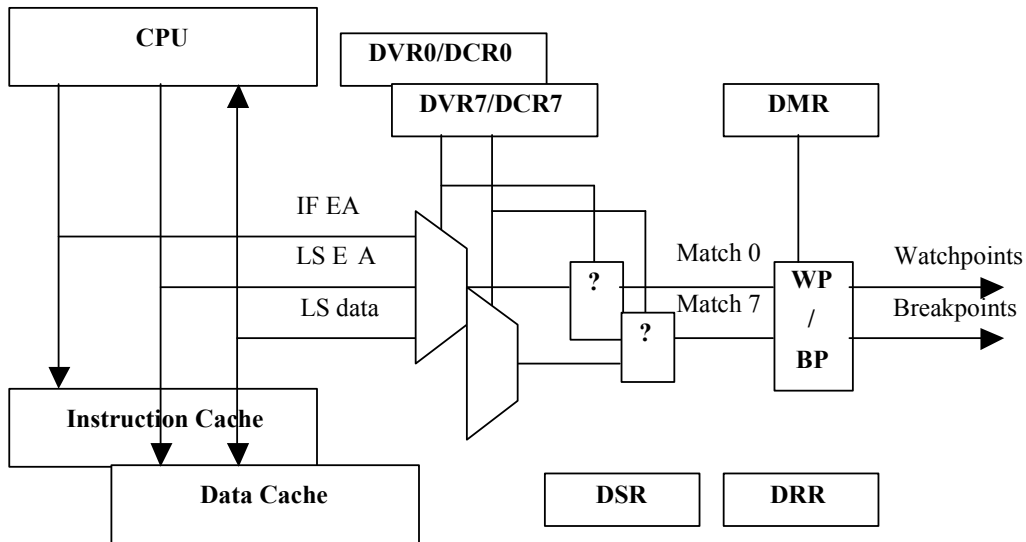


Figure 11-1. Block Diagram of Debug Support

## 11.2 Debug Value Registers (DVR0-DVR7)

The debug value registers are 32-bit special-purpose supervisor-level registers accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

The DVRs are programmed with the watchpoint addresses or data by the resident debug software or by the development interface. Their value is compared to the fetch or load/store EA or to the load/store data according to the corresponding DCR. Based on the settings of the corresponding DCR a watchpoint is generated.

Bit	31-0
Identifier	VALUE
Reset	0
R/W	R/W

VALUE	Watchpoint/Breakpoint Address/Data
-------	------------------------------------

Table 11-1. DVR Field Descriptions

## 11.3 Debug Control Registers (DCR0-DCR7)

The debug control registers are 32-bit special-purpose supervisor-level registers accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

The DCRs are programmed with the watchpoint settings that define how DVRs are compared to the instruction fetch or load/store EA or to the load/store data.

Bit	31-8	7-5	4	3-1	0
Identifier	Reserved	CT	SC	CC	DP
Reset	X	0	0	0	0
R/W	R	R/W	R/W	R/W	R

DP	DVR/DCR Present 0 Corresponding DVR/DCR pair is not present 1 Corresponding DVR/DCR pair is present
CC	Compare Condition 000 Masked 001 Equal 010 Less than 011 Less than or equal 100 Greater than 101 Greater than or equal 110 Not equal 111 Reserved
SC	Signed Comparison 0 Compare using unsigned integers 1 Compare using signed integers
CT	Compare To 000 Comparison disabled 001 Instruction fetch EA 010 Load EA 011 Store EA 100 Load data 101 Store data 110 Load/Store EA 111 Load/Store data

Table 11-2. DCR Field Descriptions

## 11.4 Debug Mode Register 1 (DMR1)

The debug mode register 1 is a 32-bit special-purpose supervisor-level register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

The DMR1 is programmed with the watchpoint/breakpoint settings that define how DVR/DCR pairs operate and is set by the resident debug software or by the development interface.



Bit	31-25	23	22	21-20	19-18	17-16
Identifier	Reserved	BT	ST	Res	CW9	CW8
Reset	X	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W	R/W

Bit	15-14	13-12	11-10	9-8	7-6	5-4	3-2	1-0
Identifier	CW7	CW6	CW5	CW4	CW3	CW2	CW1	CW0
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

CW0	Chain Watchpoint 0 00 Watchpoint 0 = Match 0 01 Watchpoint 0 = Match 0 & External Watchpoint 10 Watchpoint 0 = Match 0   External Watchpoint 11 Reserved
CW1	Chain Watchpoint 1 00 Watchpoint 1 = Match 1 01 Watchpoint 1 = Match 1 & Watchpoint 0 10 Watchpoint 1 = Match 1   Watchpoint 0 11 Reserved
CW2	Chain Watchpoint 2 00 Watchpoint 2 = Match 2 01 Watchpoint 2 = Match 2 & Watchpoint 1 10 Watchpoint 2 = Match 2   Watchpoint 1 11 Reserved
CW3	Chain Watchpoint 3 00 Watchpoint 3 = Match 3 01 Watchpoint 3 = Match 3 & Watchpoint 2 10 Watchpoint 3 = Match 3   Watchpoint 2 11 Reserved
CW4	Chain Watchpoint 4 00 Watchpoint 4 = Match 4 01 Watchpoint 4 = Match 4 & External Watchpoint 10 Watchpoint 4 = Match 4   External Watchpoint 11 Reserved
CW5	Chain Watchpoint 5 00 Watchpoint 5 = Match 5 01 Watchpoint 5 = Match 5 & Watchpoint 4 10 Watchpoint 5 = Match 5   Watchpoint 4 11 Reserved

CW6	Chain Watchpoint 6 00 Watchpoint 6 = Match 6 01 Watchpoint 6 = Match 6 & Watchpoint 5 10 Watchpoint 6 = Match 6   Watchpoint 5 11 Reserved
CW7	Chain Watchpoint 7 00 Watchpoint 7 = Match 7 01 Watchpoint 7 = Match 7 & Watchpoint 6 10 Watchpoint 7 = Match 7   Watchpoint 6 11 Reserved
CW8	Chain Watchpoint 8 00 Watchpoint 8 = Watchpoint counter 0 match 01 Watchpoint 8 = Watchpoint counter 0 match & Watchpoint 3 10 Watchpoint 8 = Watchpoint counter 0 match   Watchpoint 3 11 Reserved
CW9	Chain Watchpoint 9 00 Watchpoint 9 = Watchpoint counter 1 match 01 Watchpoint 9 = Watchpoint counter 1 match & Watchpoint 7 10 Watchpoint 9 = Watchpoint counter 1 match   Watchpoint 7 11 Reserved
ST	Single-step Trace 0 Single-step trace disabled 1 Every executed instruction causes trap exception
BT	Branch Trace 0 Branch trace disabled 1 Every executed branch instruction causes trap exception

Table 11-3. DMR1 Field Descriptions

## 11.5 Debug Mode Register 2 (DMR2)

The debug mode register 2 is a 32-bit special-purpose supervisor-level register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

The DMR2 is programmed with the watchpoint/breakpoint settings that define which watchpoints generate a breakpoint and which watchpoint counters are enabled. When a breakpoint happens WBS provides information which watchpoint or several watchpoints caused breakpoint condition. WBS bits are sticky and should be cleared by writing 0 of them every time a breakpoint condition is processed. DMR2 is set by the resident debug software or by the development interface.

Bit	31-22	21-12	11-2	1	0
Identifier	WBS	WGB	AWTC	WCE1	WCE0
Reset	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W

WCE0	Watchpoint Counter Enable 0 0 Counter 0 disabled 1 Counter 0 enabled
WCE1	Watchpoint Counter Enable 1 0 Counter 1 disabled 1 Counter 1 enabled
AWTC	Assign Watchpoints to Counter 00 0000 0000 All Watchpoints increment counter 0 00 0000 0001 Watchpoint 0 increments counter 1 ... 00 0000 1111 First four watchpoints increment counter 1, rest increment counter 0 ... 11 1111 1111 All watchpoints increment counter 1
WGB	Watchpoints Generating Breakpoint (trap exception) 00 0000 0000 Breakpoint disabled 00 0000 0001 Watchpoint 0 generates breakpoint ... 01 0000 0000 Watchpoint counter 0 generates breakpoint ... 11 1111 1111 All watchpoints generate breakpoint
WBS	Watchpoints Breakpoint Status 00 0000 0000 No watchpoint caused breakpoint 00 0000 0001 Watchpoint 0 caused breakpoint ... 01 0000 0000 Watchpoint counter 0 caused breakpoint ... 11 1111 1111 Any watchpoint could have caused breakpoint

Table 11-4. DMR2 Field Descriptions

## 11.6 Debug Watchpoint Counter Register (DWCR0-DWCR1)

The debug watchpoint counter registers are 32-bit special-purpose supervisor-level registers accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

The DWCRs contain 16-bit counters that count watchpoints programmed in the DMR. The value in a DWCR can be accessed by the resident debug software or by the development interface. DWCRs also contain match values. When a counter reaches the match value, a watchpoint is generated.

Bit	31-16	15-0
Identifier	MATCH	COUNT
Reset	0	0
R/W	R/W	R/W

COUNT	Number of watchpoints programmed in DMR N 16-bit counter of generated watchpoints assigned to this counter
MATCH	N 16-bit value that when matched generates a watchpoint

**Table 11-5. DWCR Field Descriptions**

## 11.7 Debug Stop Register (DSR)

The debug stop register is a 32-bit special-purpose supervisor-level register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

The DSR specifies which exceptions cause the core to stop the execution of the exception handler and turn over control to development interface. It can be programmed by the resident debug software or by the development interface.

Bit	31-14	13	12	11	10	9	8
Identifier	Reserved	TE	FPE	SCE	RE	IME	DME
Reset	X	0	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W	R/W	R/W

Bit	7	6	5	4	3	2	1	0
Identifier	INTE	IIE	AE	TTE	IPFE	DPFE	BUSEE	RSTE
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

RSTE	Reset Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
BUSEE	Bus Error Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
DPFE	Data Page Fault Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
IPFE	Instruction Page Fault Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface

TTE	Tick Timer Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
AE	Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
IIE	Illegal Instruction Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
INTE	Interrupt Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
DME	DTLB Miss Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
IME	ITLB Miss Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
RE	Range Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
SCE	System Call Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
FPE	Floating Point Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
TE	Trap Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface

Table 11-6. DSR Field Descriptions

## 11.8 Debug Reason Register (DRR)

The debug reason register is a 32-bit special-purpose supervisor-level register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

The DRR specifies which event caused the core to stop the execution of program flow and turned control over to the development interface. It should be cleared by the resident debug software or by the development interface.

Bit	31-14	13	12	11	10	9	8
Identifier	Reserved	TE	FPE	SCE	RE	IME	DME
Reset	X	0	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W	R/W	R/W

Bit	7	6	5	4	3	2	1	0
Identifier	INTE	IIE	AE	TTE	IPFE	DPFE	BUSEE	RSTE
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

RSTE	Reset Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
BUSEE	Bus Error Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
DPFE	Data Page Fault Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
IPFE	Instruction Page Fault Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
TTE	Tick Timer Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
AE	Alignment Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
IIE	Illegal Instruction Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
INTE	Interrupt Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
DME	DTLB Miss Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
IME	ITLB Miss Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
RE	Range Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface

SCE	System Call Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
FPE	Floating Point Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface
TE	Trap Exception 0 This exception did not transfer control to the development I/F 1 This exception transferred control to the development interface

**Table 11-7. DRR Field Descriptions**

## 12 Performance Counters Unit (Optional)

This chapter describes the OpenRISC 1000 performance counters facility. Performance counters can be used to count predefined events such as L1 instruction or data cache misses, branch instructions, pipeline stalls etc.

Data from the Performance Counters Unit can be used for the following:

- ✓ To improve performance by developing better application level algorithms, better optimized operating system routines and for improvements in the hardware architecture of these systems (e.g. memory subsystems).
- ✓ To improve future OpenRISC implementations and add future enhancements to the OpenRISC architecture.
- ✓ To help system developers debug and test their systems.

### 12.1 Features

The OpenRISC 1000 architecture defines eight performance counters. Additional performance counters can be defined by the implementation itself. The Performance Counters Unit is optional and the presence of an implementation is indicated by the UPR[PCUP] bit.

- ✓ Optional implementation.
- ✓ Eight architecture defined performance counters
- ✓ Eight custom performance counters
- ✓ Programmable counting conditions.

### 12.2 Performance Counters Count Registers (PCCR0-PCCR7)

The performance counters count registers are 32-bit special-purpose supervisor-level registers accessible with the l.mtspr/l.mfspr instructions in supervisor mode. Read access in user mode is possible, if it is enabled in SR[SUMRA].

They are counters of the events programmed in the PCMR registers.

Bit	31-0
Identifier	COUNT
Reset	0
R/W	R/W



COUNT	Event counter
-------	---------------

Table 12-1. PCCR0 Field Descriptions

## 12.3 Performance Counters Mode Registers (PCMR0-PCMR7)

The performance counters mode registers are 32-bit special-purpose supervisor-level registers accessible with the l.mtspr/l.mfspr instructions in supervisor mode.

They define which events the performance counters unit counts.

Bit	31-26	25-15	14	13	12	11	10
Identifier	Reserved	WPE	DDS	ITLBM	DTLBM	BS	LSUS
Reset	X	0	0	0	0	0	0
R/W	Read Only	R/W	R/W	R/W	R/W	R/W	R/W

Bit	9	8	7	6	5	4	3	2	1	0
Identifier	IFS	ICM	DCM	IF	SA	LA	CIUM	CISM	Reserved	CP
Reset	0	0	0	0	0	0	0	0	0	1
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R

CP	Counter Present 0 Counter not present 1 Counter present
CISM	Count in Supervisor Mode 0 Counter disabled in supervisor mode 1 Counter counts events in supervisor mode
CIUM	Count in User Mode 0 Counter disabled in user mode 1 Counter counts events in user mode
LA	Load Access event 0 Event ignored 1 Count load accesses
SA	Store Access event 0 Event ignored 1 Count store accesses
IF	Instruction Fetch event 0 Event ignored 1 Count instruction fetches

DCM	Data Cache Miss event 0 Event ignored 1 Count data cache missed
ICM	Instruction Cache Miss event 0 Event ignored 1 Count instruction cache misses
IFS	Instruction Fetch Stall event 0 Event ignored 1 Count instruction fetch stalls
LSUS	LSU Stall event 0 Event ignored 1 Count LSU stalls
BS	Branch Stalls event 0 Event ignored 1 Count branch stalls
DTLBM	DTLB Miss event 0 Event ignored 1 Count DTLB misses
ITLBM	ITLB Miss event 0 Event ignored 1 Count ITLB misses
DDS	Data Dependency Stalls event 0 Event ignored 1 Count data dependency stalls
WPE	Watchpoint Events 000 0000 0000 All watchpoint events ignored 000 0000 0001 Watchpoint 0 counted ... 111 1111 1111 All watchpoints counted

**Table 12-2. PCMR Field Descriptions**

# 13 Power Management (Optional)

This chapter describes the OpenRISC 1000 power management facility. The power management facility is optional and implementation may choose which features to implement, and which not. UPR[PMP] indicates whether power management is implemented or not.

Note that this chapter describes the architectural control of power management from the perspective of the programming model. As such, it does not describe technology specific optimizations or implementation techniques.

## 13.1 Features

The OpenRISC 1000 architecture defines five architectural features for minimizing power consumption:

- ✓ slow down feature
- ✓ doze mode
- ✓ sleep mode
- ✓ suspend mode
- ✓ dynamic clock gating feature

The slow down feature takes advantage of the low-power dividers in external clock generation circuitry to enable full functionality, but at a lower frequency so that power consumption is reduced.

The slow down feature is software controlled with the 4-bit value in PMR[SDF]. A lower value specifies higher expected performance from the processor core. Whether this value controls a processor clock frequency or some other implementation specific feature is irrelevant to the controlling software. Usually PMR[SDF] is dynamically set by the operating system's idle routine, that monitors the usage of the processor core.

When software initiates the doze mode, software processing on the core suspends. The clocks to the processor internal units are disabled except to the internal tick timer and programmable interrupt controller. However other on-chip blocks (outside of the processor block) can continue to function as normal.

The processor should leave doze mode and enter normal mode when a pending interrupt occurs.

In sleep mode, all processor internal units are disabled and clocks gated. Optionally, an implementation may choose to lower the operating voltage of the processor core.

The processor should leave sleep mode and enter normal mode when a pending interrupt occurs.

In suspend mode, all processor internal units are disabled and clocks gated. Optionally, an implementation may choose to lower the operating voltage of the processor core.

The processor enters normal mode when it is reset. Software may implement a reset exception handler that refreshes system memory and updates the RISC with the state prior to the suspension.

If enabled, the clock-gating feature automatically disables clock subtrees to major processor internal units on a clock cycle basis. These blocks are usually the CPU, FPU/VU, IC, DC, IMMU and DMMU. This feature can be used in a combination with other power management features and low-power modes.

Cache or MMU blocks that are already disabled when software enables this feature, have completely disabled clock subtrees until clock gating is disabled or until the blocks are again enabled.

## 13.2 Power Management Register (PMR)

The power management register is a 32-bit special-purpose supervisor-level register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

PMR is used to enable or disable power management features and modes.

Bit	31-7	7	6	5	4	3-0
Identifier	Reserved	SUME	DCGE	SME	DME	SDF
Reset	X	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W	R/W

SDF	Slow Down Factor 0 Full speed 1-15 Logarithmic clock frequency reduction
DME	Doze Mode Enable 0 Doze mode not enabled 1 Doze mode enabled
SME	Sleep Mode Enable 0 Sleep mode not enabled 1 Sleep mode enabled
DCGE	Dynamic Clock Gating Enable 0 Dynamic clock gating not enabled 1 Dynamic clock gating enabled
SUME	Suspend Mode Enable 0 Suspend mode not enabled 1 Suspend mode enabled

**Table 13-1. PMR Field Descriptions**

# 14 Programmable Interrupt Controller (Optional)

This chapter describes the OpenRISC 1000 level one programmable interrupt controller. The interrupt controller facility is optional and an implementation may chose whether or not to implement it. If it is not implemented, interrupt input is directly connected to interrupt exception inputs. UPR[PICP] specifies whether the programmable interrupt controller is implemented or not.

The Programmable Interrupt Controller has two special-purpose registers and 32 maskable interrupt inputs. If implementation requires permanent unmasked interrupt inputs, it can use interrupt inputs [1:0] and PICMR[1:0] should be fixed to one.

## 14.1 Features

The OpenRISC 1000 architecture defines an interrupt controller facility with up to 32 interrupt inputs:

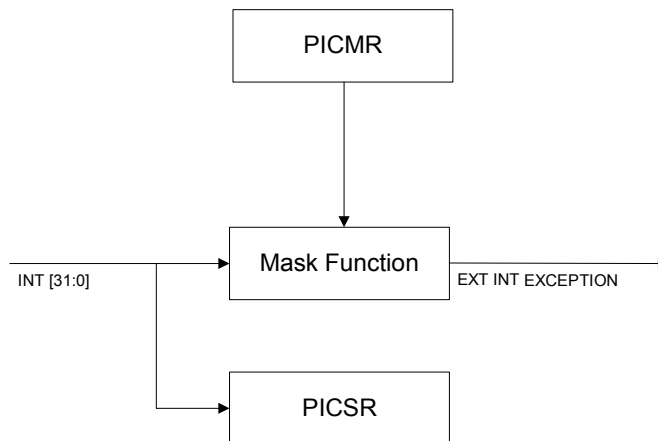


Figure 14-1. Programmable Interrupt Controller Block Diagram

## 14.2 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 32 programmable interrupt sources.

Bit	31-0
Identifier	IUM
Reset	0
R/W	R/W

IUM	<p>Interrupt UnMask</p> <p>0x00000000 All interrupts are masked</p> <p>0x00000001 Interrupt input 0 is enabled, all others are masked</p> <p>...</p> <p>0xFFFFFFFF All interrupt inputs are enabled</p>
-----	---

Table 14-1. PICMR Field Descriptions

## 14.3 PIC Status Register (PICSR)

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

PICSR is used to determine the status of each PIC interrupt input. PIC can support level-triggered interrupts or combination of level-triggered and edge-triggered. Most implementations today only support level-triggered interrupts.

For level-triggered implementations bits in PICSR simply represent level of interrupt inputs. Interrupts are cleared by taking appropriate action at the device to negate the source of the interrupt. Writing a '1' or a '0' to bits in the PICSR that reflect a level-triggered source must have no effect on PICSR content.

The atomic way to clear an interrupt source which is edge-triggered is by writing a '1' to the corresponding bit in the PICSR. This will clear the underlying latch for the edge-triggered source. Writing a '0' to the corresponding bit in the PICSR has no effect on the underlying latch.

Bit	31-0
Identifier	IS
Reset	0
R/W	R/(W*)

IS	<p>Interrupt Status</p> <p>0x00000000 All interrupts are inactive</p> <p>0x00000001 Interrupt input 0 is pending</p> <p>...</p> <p>0xFFFFFFFF All interrupts are pending</p>
----	--

Table 14-2. PICSR Field Descriptions

# 15 Tick Timer Facility (Optional)

This chapter describes the OpenRISC 1000 tick timer facility. It is optional and an implementation may choose whether or not to implement it. UPR[TTP] specifies whether or not the tick timer facility is present.

The Tick Timer is used to schedule operating system and user tasks on regular time basis or as a high precision time reference.

The Tick Timer facility is enabled with TTMR[M]. TTCR is incremented with each clock cycle and a tick timer interrupt can be asserted whenever the lower 28 bits of TTCR match TTMR[TP] and TTMR[IE] is set.

TTCR restarts counting from zero when a match event happens and TTMR[M] is 0x1. If TTMR[M] is 0x2, TTCR is stopped when match event happens and TTCR must be changed to start counting again. When TTMR[M] is 0x3, TTCR keeps counting even when match event happens.

## 15.1 Features

The OpenRISC 1000 architecture defines a tick timer facility with the following features:

- ✓ Maximum timer count of  $2^{32}$  clock cycles
- ✓ Maximum time period of  $2^{28}$  clock cycles between interrupts
- ✓ Maskable tick timer interrupt
- ✓ Single run, restartable counter, or continues counter

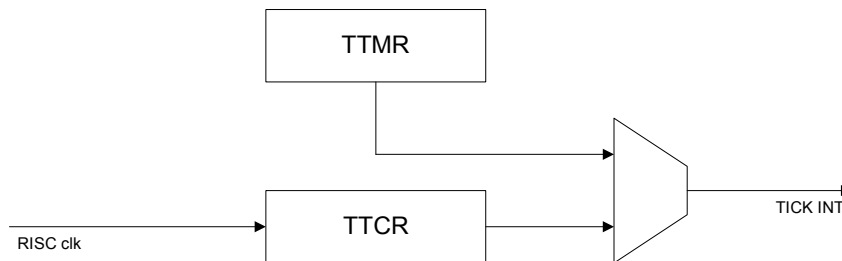


Figure 15-1. Tick Timer Block Diagram

## 15.2 Timer interrupts

A timer interrupt will happen everytime TTMR[IE] bit is set and TTMR[TP] matches the lower 28 bits of the TTCR SPR, the top 4 bits are ignored for the comparison. When an interrupt is pending the TTMR[IP] bit will be set and the interrupt will be asserted to the cpu core until it is cleared by writting a 0 to the TTMR[IP] bit. However, if the TTMR[IE] bit was not set when a match condition occurred no interrupt will be asserted and the TTMR[IP] bit won't be set unless it has not been cleared from a previous interrupt. The TTMR[IE] bit is not meant as a mask bit, SR[TEE] is provided for that purpose.

## 15.3 Timer modes

It is up to the programmer to ensure that the TTCR SPR is set to a sane value before the timer mode is programmed. When the timing mode is programmed into the timer by setting TTMR[M], the TTCR SPR is not preset to any predefined value, including 0. If the lower 28 bits of the TTCR SPR is numerically greater than what was programmed into TTMR[TP] then the timer will only assert the timer interrupt when the lower 28 bits of the TTCR SPR have wrapped around to 0 and counted up to the match value programmed into TTMR[TP].

### 15.3.1 Disabled timer

In this mode the timer does not increment the TTCR spr. Though note that the timer interrupt is independent from the timer mode and as such the timer interrupt is not disabled when the timer is disabled.

### 15.3.2 Auto-restart timer

When the timer is set to auto-restart mode, the timer will reset the TTCR spr to 0 as soon as the lower 28 bits of the TTCR spr match TTMR[TP] and the timer interrupt will be asserted to the cpu core if the TTMR[IE] bit has been set.

### 15.3.3 One-shot timer

In one-shot timing mode, the timer stops counting as soon as a match condition has been reached. Although the timer has in effect been disabled (and can't be restarted by writting to the TTCR spr) the TTMR[M] bits shall still indicate that the timer is in one-shot mode and not that it has been disabled. Care should be taken that the timer interrupt has been masked (or disabled) after the match condition has been reached, or else the cpu core will get a spurious timer interrupt.



### 15.3.4 Continuous timer

In the event that a match condition has been reached, the counter does not stop but rather keeps counting from the value of the TTCR spr and the timer interrupt will be asserted if the TTMR[IE] bit has been set.

## 15.4 Tick Timer Mode Register (TTMR)

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

The TTMR is programmed with the time period of the tick timer as well as with the mode bits that control operation of the tick timer.

Bit	31-30	29	28	27-0
Identifier	M	IE	IP	TP
Reset	0	0	0	X
R/W	R/W	R/W	R	R/W

TP	Time Period 0x00000000 Shortest comparison time period ... 0xFFFFFFFF Longest comparison time period
IP	Interrupt Pending 0 Tick timer interrupt is not pending 1 Tick timer interrupt pending (write '0' to clear it)
IE	Interrupt Enable 0 Tick timer does not generate tick timer interrupt 1 Tick timer generates tick timer interrupt when TTMR[TP] matches TTCR[27:0]
M	Mode 00 Tick timer is disabled 01 Timer is restarted when TTMR[TP] matches TTCR[27:0] 10 Timer stops when TTMR[TP] matches TTCR[27:0] (change TTCR to resume counting) 11 Timer does not stop when TTMR[TP] matches TTCR[27:0]

**Table 15-1. TTMR Field Descriptions**

## 15.5 Tick Timer Count Register (TTCR)

The tick timer count register is a 32-bit special-purpose register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode and as read-only register in user mode if enabled in `SR[SUMRA]`.

TTCR holds the current value of the timer.

Bit	31-0
Identifier	CNT
Reset	0
R/W	R/W

CNT	Count 32-bit incrementing counter
-----	--------------------------------------

**Table 15-2. TTCR Field Descriptions**

# 16 OpenRISC 1000 Implementations

## 16.1 Overview

Implementations of the OpenRISC 1000 architecture come in different configurations and version releases.

Version and unit present registers both identify the model, version and its configuration. Detailed configuration for some units is available in configuration registers.

An operating system can read VR, UPR and the configuration registers, and adjust its own operation if required. Operating systems ported on a particular OpenRISC version should run on different configurations of this version without modifications.

## 16.2 Version Register (VR)

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

It identifies the version (model) and revision level of the OpenRISC 1000 processor. It also specifies the possible template on which this implementation is based.

**This register is deprecated, and the AVR and VR2 SPR should be used to determine more accurately the version information.**

Bit	31-24	23-16	15-7	6	5-0
Identifier	VE R	CFG	Reserved	UVRP	REV
Reset	-	-	x	-	-
R/W	R	R	R	R	R

REV	Revision 0..63 A 6-bit number that identifies various releases of a particular version. This number is changed for each revision of the device.
UVRP	Updated Version Registers Present A bit indicating that the AVR and VR2 SPRs are available and should be used to determine version information.

CFG	<p>Configuration Template</p> <p>0..99 An 8-bit number that identifies particular configuration. However this is just for operating systems that do not use information provided by configuration registers and thus are not truly portable across different configurations of one implementation version.</p> <p>Configurations that do implement configuration registers must have their CFG smaller than 50 and configurations that do not implement configuration registers must have their CFG 50 or bigger.</p>
VER	<p>Version</p> <p>0x10..0x19 An 8-bit number that identifies a particular processor version and version of the OpenRISC architecture. Values below 0x10 and above 0x19 are illegal for OpenRISC 1000 processor implementations.</p>

Table 16-1. VR Field Descriptions

## 16.3 Unit Present Register (UPR)

The unit present register is a 32-bit special-purpose supervisor-level register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

It identifies the present units in the processor. It has a bit for each possible unit or functionality. The lower sixteen bits identify the presence of units defined in the OpenRISC 1000 architecture. The upper sixteen bits define the presence of custom units.

Bit	31-24	23-11	10	9	8	7
Identifier	CUP	Reserved	TTP	PMP	PICP	PCUP
Reset	-	-	-	-	-	-
R/W	R	R	R	R	R	R

Bit	6	5	4	3	2	1	0
Identifier	DUP	MP	IMP	DMP	ICP	DCP	UP
Reset	-	-	-	-	-	-	-
R/W	R	R	R	R	R	R	R

UP	<p>UPR Present</p> <p>0 UPR is not present</p> <p>1 UPR is present</p>
DCP	<p>Data Cache Present</p> <p>0 Unit is not present</p> <p>1 Unit is present</p>
ICP	<p>Instruction Cache Present</p> <p>0 Unit is not present</p> <p>1 Unit is present</p>

DMP	Data MMU Present 0 Unit is not present 1 Unit is present
IMP	Instruction MMU Present 0 Unit is not present 1 Unit is present
MP	MAC Present 0 Unit is not present 1 Unit is present
DUP	Debug Unit Present 0 Unit is not present 1 Unit is present
PCUP	Performance Counters Unit Present 0 Unit is not present 1 Unit is present
PMP	Power Management Present 0 Unit is not present 1 Unit is present
PICP	Programmable Interrupt Controller Present 0 Unit is not present 1 Unit is present
TTP	Tick Timer Present 0 Unit is not present 1 Unit is present
CUP	Custom Units Present

Table 16-2. UPR Field Descriptions

## 16.4 CPU Configuration Register (CPUCFGR)

The CPU configuration register is a 32-bit special-purpose supervisor-level register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

It specifies CPU capabilities and configuration.

Bit	31-14	14	13	12	11	10
Identifier	Reserved	AECSR	ISRP	EVBARP	AVRP	ND
Reset	-	-	-	-	-	-
R/W	R	R	R	R	R	R

Bit	9	8	7	6	5	4	3-0
Identifier	OV64S	OF64S	OF32S	OB64S	OB32S	CGF	NSGF
Reset	-	-	-	-	-	-	-
R/W	R	R	R	R	R	R	R

NSGF	Number of Shadow GPR Files 0 Zero shadow GPR files 15 Fifteen shadow GPR Files
CGF	Custom GPR File 0 GPR file has 32 registers 1 GPR file has less than 32 registers
OB32S	ORBIS32 Supported 0 Not supported 1 Supported
OB64S	ORBIS64 Supported 0 Not supported 1 Supported
OF32S	ORFPX32 Supported 0 Not supported 1 Supported
OF64S	ORFPX64 Supported 0 Not supported 1 Supported
OV64S	ORVDX64 Supported 0 Not supported 1 Supported
ND	No Delay-Slot 0 CPU executes delay slot of jump/branch instructions before taking jump/branch 1 CPU does not execute instructions in delay slot if taking jump/branch
AVRP	Architecture Version Register (AVR) Present 0 AVR not present 1 AVR present
EVBARP	Exception Vector Base Address Register (EVBAR) Present 0 EVBAR not present 1 EVBAR present
ISRP	Implementation-Specific Registers (ISR0-7) Preset 0 ISRs not present 1 ISRs present
AECSR	Arithmetic Exception Control Register (AECR) and Arithmetic Exception Status Register (AESR) present 0 AECR and AESR not present 1 AECR and AESR present

Table 16-3. CPUCFGR Field Descriptions

## 16.5 DMMU Configuration Register (DMMUCFGR)

The DMMU configuration register is a 32-bit special-purpose supervisor-level register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

It specifies the DMMU capabilities and configuration.

Bit	31-12
Identifier	Reserved
Reset	-
R/W	R

Bit	11	10	9	8	7-5	4-2	1-0
Identifier	HTR	TEIRI	PRI	CRI	NAE	NTS	NTW
Reset	-	-	-	-	-	-	-
R/W	R	R	R	R	R	R	R

NTW	Number of TLB Ways 0 DTLB has one way ... 3 DTLB has four ways
NTS	Number of TLB Sets (entries per way) 0 DTLB has one set (entries per way) ... 7 DTLB has 128 sets (entries per way)
NAE	Number of ATB Entries 0 DATB does not exist 1 DATB has one entry ... 4 DATB has four entries 5..7 Invalid values
CRI	Control Register Implemented 0 DMMUCR not implemented 1 DMMUCR implemented
PRI	Protection Register Implemented 0 DMMUPR not implemented 1 DMMUPR implemented
TEIRI	TLB Entry Invalidate Register Implemented 0 DTLBEIR not implemented 1 DTLBEIR implemented

HTR	Hardware TLB Reload 0 TLB Entry reloaded in software 1 TLB Entry reloaded in hardware
-----	---

Table 16-4. DMMUCFGR Field Descriptions

## 16.6IMMU Configuration Register (IMMUCFGR)

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

It specifies IMMU capabilities and configuration.

Bit	31-12
Identifier	Reserved
Reset	-
R/W	R

Bit	11	10	9	8	7-5	4-2	1-0
Identifier	HTR	TEIRI	PRI	CRI	NAE	NTS	NTW
Reset	-	-	-	-	-	-	-
R/W	R	R	R	R	R	R	R

NTW	Number of TLB Ways 0 ITLB has one way ... 3 ITLB has four ways
NTS	Number of TLB Sets (entries per way) 0 ITLB has one set (entries per way) ... 7 ITLB has 128 sets (entries per way)
NAE	Number of ATB Entries 0 IATB does not exist 1 IATB has one entry ... 4 IATB has four entries 5..7 Invalid values
CRI	Control Register Implemented 0 IMMUCR not implemented 1 IMMUCR implemented
PRI	Protection Register Implemented 0 IMMUPR not implemented 1 IMMUPR implemented



TEIRI	TLB Entry Invalidate Register Implemented 0 ITLBEIR not implemented 1 ITLBEIR implemented
HTR	Hardware TLB Reload 0 ITLB Entry reloaded in software 1 ITLB Entry reloaded in hardware

Table 16-5. IMMUCFGR Field Descriptions

## 16.7DC Configuration Register (DCCFGR)

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

It specifies data cache capabilities and configuration.

Bit	31-15	14	13	12
Identifier	Reserved	CBWBRI	CBFRI	CBLRI
Reset	-	-	-	-
R/W	R	R	R	R

Bit	11	10	9	8	7	6-3	2-0
Identifier	CBPRI	CBIRI	CCRI	CWS	CBS	NCS	NCW
Reset	-	-	-	-	-	-	-
R/W	R	R	R	R	R	R	R

NCW	Number of Cache Ways 0 DC has one way ... 5 DC has thirty-two ways
NCS	Number of Cache Sets (cache blocks per way) 0 DC has one set (cache blocks per way) ... 10 DC has 1024 sets (cache blocks per way)
BS	Cache Block Size 0 Cache block size 16 bytes 1 Cache block size 32 bytes
CWS	Cache Write Strategy 0 Cache write-through 1 Cache write-back
CCRI	Cache Control Register Implemented 0 Register is not implemented 1 Register is implemented

CBIRI	Cache Block Invalidate Register Implemented 0 Register is not implemented 1 Register is implemented
CBPRI	Cache Block Prefetch Register Implemented 0 Register is not implemented 1 Register is implemented
CBLRI	Cache Block Lock Register Implemented 0 Register is not implemented 1 Register is implemented
CBFRI	Cache Block Flush Register Implemented 0 Register is not implemented 1 Register is implemented
CBWBRI	Cache Block Write-Back Register Implemented 0 Register is not implemented 1 Register is implemented

Table 16-6. DCCFGR Field Descriptions

## 16.8 IC Configuration Register (ICCFGR)

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

It specifies instruction cache capabilities and configuration.

Bit	31-13	12
Identifier	Reserved	CBLRI
Reset	-	-
R/W	R	R

Bit	11	10	9	8	7	6-3	2-0
Identifier	CBPRI	CBIRI	CCRI	Res	CBS	NCS	NCW
Reset	-	-	-	-	-	-	-
R/W	R	R	R	R	R	R	R

NCW	Number of Cache Ways 0 IC has one way ... 5 IC has thirty-two ways
NCS	Number of Cache Sets (cache blocks per way) 0 IC has one set (cache blocks per way) ... 10 IC has 1024 sets (cache blocks per way)

BS	Cache Block Size 0 Cache block size 16 bytes 1 Cache block size 32 bytes
CCRI	Cache Control Register Implemented 0 Register is not implemented 1 Register is implemented
CBIRI	Cache Block Invalidate Register Implemented 0 Register is not implemented 1 Register is implemented
CBPRI	Cache Block Prefetch Register Implemented 0 Register is not implemented 1 Register is implemented
CBLRI	Cache Block Lock Register Implemented 0 Register is not implemented 1 Register is implemented

Table 16-7. ICCFGR Field Descriptions

## 16.9 Debug Configuration Register (DCFGR)

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

It specifies debug unit capabilities and configuration.

Bit	31-4	3	2-0
Identifier	Reserved	WPCI	NDP
Reset	-	-	-
R/W	R	R	R

NDP	Number of Debug Pairs 0 Debug unit has one DCR/DVR pair ... 7 Debug unit has eight DCR/DVR pairs
WPCI	Watchpoint Counters Implemented 0 Watchpoint counters not implemented 1 Watchpoint counters implemented

Table 16-8. DCFGR Field Descriptions

## 16.10 Performance Counters Configuration Register (PCCFGR)

The performance counters configuration register is a 32-bit special-purpose supervisor-level register accessible with the `l.mtspr/l.mfspr` instructions in supervisor mode.

It specifies performance counters unit capabilities and configuration.

Bit	31-3	2-0
Identifier	Reserved	NPC
Reset	-	-
R/W	R	R

NPC	Number of Performance Counters 0 One performance counter ... 7 Eight performance counters
-----	--

**Table 16-9. PCCFGR Field Descriptions**

## 16.11 Version Register 2 (VR2)

The version register 2 is a 32-bit special-purpose supervisor-level register accessible with the `l.mfspr` instruction in supervisor mode.

It holds implementation-specific version information. It is intended to replace the VR register.

The value in the CPUID field should correspond to an implementation list held on the site which hosts this document. It is most likely that a master list will also be maintained at [openrisc.io](http://openrisc.io).

Its presence is indicated by the UVRP bit in the Version Register (VR).

Bit	31-24	23-0
Identifier	CPUID	VER
Reset	-	-
R/W	R	R

CPUID	CPU Identification Number Implementation-specific identification number. Each implementation should have a unique identification number.
VER	Version Implementation-specific version number. This field, if interpreted as an unsigned 24-bit number, should increase for each new version. The implementation reference manual should document the meaning of this value.

Table 16-10. VR2 Field Descriptions

## 16.12 Architecture Version Register (AVR)

The architecture version register is a 32-bit special-purpose supervisor-level register accessible with the `l.mfspr` instruction in supervisor mode.

It indicates the most recent version the implementation contains features from. The implementation must at least implement an accurate set of feature-presence bits in the appropriate registers according to that version of the architecture spec, so the presence of each of that version's features can be checked. Its presence is indicated by the AVR bit in the CPU Configuration Register (CPUCFGR).

Bit	31-24	23-16	15-8	7-0
Identifier	MAJ	MIN	REV	Reserved
Reset	-	-	-	-
R/W	R	R	R	R

MAJ	Major Architecture Version Number
MIN	Minor Architecture Version Number
REV	Architecture Revision Number

Table 16-11. AVR Field Descriptions

## 16.13 Exception Vector Base Address Register (EVBAR)

The architecture version register is a 32-bit special-purpose supervisor-level register accessible with the `l.mfspr` / `l.mtspr` instructions in supervisor mode.

This optional register can be used to apply an offset to the exception vector addresses. Its presence is indicated by the EVBARP bit in the CPU Configuration Register (CPUCFGR).

If SR[EPH] is set, this value is logically ORed with the offset that provides.

Bit	31-13	12-0
Identifier	EVBA	Reserved
Reset	-	-
R/W	R/W	R

EVBA	Exception Vector Base Address Location for the start of exception vectors. Its reset value is implementation-specific.
------	---

Table 16-12. EVBAR Field Descriptions

## 16.14 Arithmetic Exception Control Register (AECR)

The arithmetic exception control register is a 32-bit special-purpose supervisor-level register accessible with the `l.mfspr/ l.mtspr` instructions in supervisor mode.

This optional register can be used for fine-grained control over which arithmetic operations trigger overflow exceptions when the OVE bit is set in the Supervision Register (SR). Its presence is indicated by the AECSRP bit in the CPU Configuration Register (CPUCFGR).

Bit	31-7	6	5
Identifier	Reserved	OVMACADDE	CYMACADDE
Reset	-	0	0
R/W	R	R/W	R/W

Bit	4	3	2	1	0
Identifier	DBZE	OVMULE	CYMULE	OVADDE	CYADDE
Reset	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W

CYADDE	Carry on Add Exception Carry flag set by unsigned overflow on integer addition and subtraction instructions causes exception
OVADDE	Overflow on Add Exception Overflow flag set by signed overflow on integer addition and subtraction instructions causes exception
CYMULE	Carry on Multiply Exception Carry flag set by unsigned overflow on integer multiplication instructions causes exception
OVMULE	Overflow on Multiply Exception Overflow flag set by signed overflow on integer multiplication instructions causes exception
DBZE	Divide By Zero Exception Overflow flag set by divide-by-zero on integer division instruction, or carry flag set by divide-by-zero on <code>l.divu</code> instruction, causes exception
CYMACADDE	Carry on MAC Addition Exception Carry flag set by unsigned overflow on integer addition stage of MAC instructions causes exception
OVMACADDE	Overflow on MAC Addition Exception Overflow flag set by signed overflow on integer addition stage of MAC instructions causes exception

**Table 16-13. EACR Field Descriptions**

## 16.15 Arithmetic Exception Status Register (AESR)

The arithmetic exception status register is a 32-bit special-purpose supervisor-level register accessible with the `l.mfspr/l.mtspr` instructions in supervisor mode.

This optional register indicates which arithmetic operations triggered an exception. The exceptions are triggered when the OVE bit is set in the Supervision Register (SR), and the overflow or carry flag is set according to any conditions with the corresponding bit set in the Arithmetic Exception Control Register (AECR).

This register will indicate which condition in the Arithmetic Exception Control Register (AECR) caused the exception by setting the corresponding bit. The bits can be cleared by writing '0' to them. The exception will occur due to the arithmetic operation, not due to the flags in this register being set, so failing to clear the flag before returning from exception with `SR[CY]` or `SR[OV]` set will not cause another exception..

Its presence is indicated by the AECSRP bit in the CPU Configuration Register (CPUCFGR).

Bit	31-7	6	5
Identifier	Reserved	OVMACADDE	CYMACADDE
Reset	-	0	0
R/W	R	R/W	R/W

Bit	4	3	2	1	0
Identifier	DBZE	OVMULE	CYMULE	OVADDE	CYADDE
Reset	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W

CYADDE	Carry on Add Exception Carry flag set by unsigned overflow on integer addition and subtraction instructions caused exception
OVADDE	Overflow on Add Exception Overflow flag set by signed overflow on integer addition and subtraction instructions caused exception
CYMULE	Carry on Multiply Exception Carry flag set by unsigned overflow on integer multiplication instructions caused exception
OVMULE	Overflow on Multiply Exception Overflow flag set by signed overflow on integer multiplication instructions caused exception
DBZE	Divide By Zero Exception Overflow flag set by divide-by-zero on integer division instruction, or carry flag set by divide-by-zero on <code>l.divu</code> instruction, caused exception

CYMACADDE	Carry on MAC Addition Exception Carry flag set by unsigned overflow on integer addition stage of MAC instructions caused exception
OVMACADDE	Overflow on MAC Addition Exception Overflow flag set by signed overflow on integer addition stage of MAC instructions caused exception

Table 16-14. EASR Field Descriptions

## 16.16 Implementation-Specific Registers (ISR0-7)

The implementation-specific registers are 32-bit special-purpose supervisor-level register accessible with the `l.mfspr` instruction in supervisor mode.

They are SPR space which can be used by implementations for any purpose. Their presence is indicated by the ISRP bit in the CPU Configuration Register (CPUCFGR).



# 17 Application Binary Interface

The ABI is currently defined only for 32-bit OpenRISC. When a toolchain is developed for 64-bit, this section will need updating.

## 17.1 Data Representation

### 17.1.1 Fundamental Types

Scalar types in the ISO/ANSI C language are based on memory operands definitions from the chapter entitled “Addressing Modes and Operand Conventions” on page 22. Similar relations between architecture and language types can be used for any other language.

Type	C TYPE	SIZEOF	ALIGNMENT (BYTES)	OPENRISC EQUIVALENT
Integral	char	1	1	Signed byte
	signed char			
	unsigned char	1	1	Unsigned byte
	short	2	2	Signed halfword
	signed short			
	unsigned short	2	2	Unsigned halfword
	int	4	4	Signed singleword
	signed int long signed long enum			
Pointer	unsigned int	4	4	Unsigned singleword
	long long signed long long	8	4	Signed doubleword
	unsigned long long	8	4	Unsigned doubleword
Floating-point	Any-type * Any-type (*) ()	4	4	Unsigned singleword
	float	4	4	Single precision float
	double	8	4	Double precision float

**Table 17-1. Scalar Types**

Prior versions of this table specified a native 8-byte alignment for 8-byte values. Since current OR1200 implementation never required this, and the compiler did not implement it, the specification has changed to match the 32-bit OpenRISC platform in use.

A null pointer of any type must be zero. All floating-point types are IEEE-754 compliant.

The OpenRISC programming model introduces a set of fundamental vector data types, as described by 17-2. For vector assignments both sides of an assignment must be of the same vector type.

VECTOR TYPE	SIZEOF	ALIGNMENT (BYTES)	OPENRISC EQUIVALENT
Vector char Vector signed char	8	8	Vector of signed bytes
Vector unsigned char	8	8	Vector of unsigned bytes
Vector short Vector signed short	8	8	Vector of signed halfwords
Vector unsigned short	8	8	Vector of unsigned halfwords
Vector int Vector signed int Vector long Vector signed long	8	8	Vector of signed singlewords
Vector unsigned int	8	8	Vector of unsigned singlewords
Vector float	8	8	Vector of single-precisions

**Table 17-2. Vector Types**

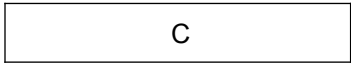
For alignment restrictions of all types see the section entitled “Aligned and Misaligned Accesses” on page 22.

## 17.1.2 Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned element.

- ✓ An array uses the alignment of its elements.
- ✓ Structures and unions can require padding to meet alignment restrictions. Each element is assigned to the lowest aligned address.

```
struct {
    char    C;
};
```



**Figure 17-1. Byte aligned, sizeof is 1**

```

struct {
    char    C;
    char    D;
    short   S;
    long    N;
};

```

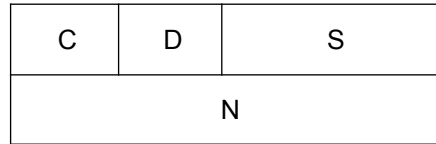


Figure 17-2. No padding, sizeof is 8

```

struct {
    char C;
    double D;
    short S;
}

```

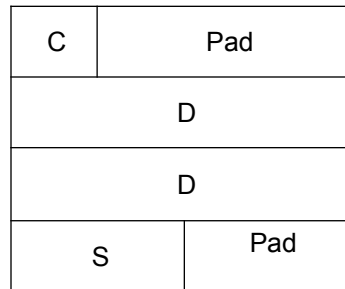


Figure 17-3. Padding, sizeof is 16

### 17.1.3 Bit-fields

C structure and union definitions can have elements defined by a specified number of bits. 17-3 describes valid bit-field types and their ranges.

Bit-field Type	Width w [bits]	Range
signed char char unsigned char	1 to 8	$-2^{w-1}$ to $2^{w-1}-1$ 0 to $2^w-1$ 0 to $2^w-1$
signed short short unsigned short	1 to 16	$-2^{w-1}$ to $2^{w-1}-1$ 0 to $2^w-1$ 0 to $2^w-1$
signed int int enum unsigned int signed long long unsigned long	1 to 32	$-2^{w-1}$ to $2^{w-1}-1$ 0 to $2^w-1$ 0 to $2^w-1$ 0 to $2^w-1$ $-2^{w-1}$ to $2^{w-1}-1$ 0 to $2^w-1$ 0 to $2^w-1$

Table 17-3. Bit-Field Types and Ranges

Bit-fields follow the same alignment rules as aggregates and unions, with the following additions:

- ✓ Bit-fields are allocated from most to least significant (from left to right)
- ✓ A bit-field must entirely reside in a storage unit appropriate for its declared type.
- ✓ Bit-fields may share a storage unit with other struct/union elements, including elements that are not bit-fields. Struct elements occupy different parts of the storage unit.
- ✓ Unnamed bit-fields' types do not affect the alignment of a structure or union

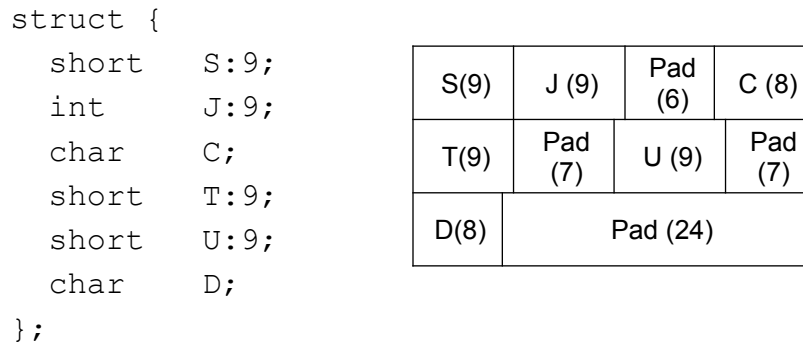


Figure 17-4. Storage unit sharing and alignment padding, sizeof is 12

## 17.2 Function Calling Sequence

This section describes the standard function calling sequence, including stack frame layout, register usage, parameter passing, and so on. The standard calling sequence requirements apply only to global functions, however it is recommended that all functions use the standard calling sequence.

### 17.2.1 Register Usage

The OpenRISC 1000 architecture defines 32 general-purpose registers. These registers are 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit implementations.

Register	Preserved across function calls	Usage
R31	No	Temporary register
R30	Yes	Callee-saved register
R29	No	Temporary register
R28	Yes	Callee-saved register
R27	No	Temporary register
R26	Yes	Callee-saved register
R25	No	Temporary register

Register	Preserved across function calls	Usage
R24	Yes	Callee-saved register
R23	No	Temporary register
R22	Yes	Callee-saved register
R21	No	Temporary register
R20	Yes	Callee-saved register
R19	No	Temporary register
R18	Yes	Callee-saved register
R17	No	Temporary register
R16	Yes	Callee-saved register
R15	No	Temporary register
R14	Yes	Callee-saved register
R13	No	Temporary register
R12	No	Temporary register for 64-bit RVH - Return value upper 32 bits of 64-bit value on 32-bit system
R11	No	RV – Return value
R10	Yes	Thread Local Storage
R9	Yes	LR – Link address register
R8	No	Function parameter word 5
R7	No	Function parameter word 4
R6	No	Function parameter word 3
R5	No	Function parameter word 2
R4	No	Function parameter word 1
R3	No	Function parameter word 0
R2	Yes	FP - Frame pointer (optional)
R1	Yes	SP - Stack pointer
R0	-	Fixed to zero

Table 17-4. General-Purpose Registers

Some registers have assigned roles:

R0 [Zero] Holds a zero value.

R1 [SP] The **stack pointer** holds the limit of the current stack frame. The first 128 bytes below the stack pointer are reserved for leaf functions, and below that are undefined. Stack pointer must be word aligned at all times.

R2 [FP] The **frame pointer** holds the address of the previous stack frame. Incoming function parameters reside in the previous stack frame and can be accessed at positive offsets from FP. The compiler may use this register for other purposes if instructed.

R3 through R8 **General-purpose parameters** use up to 6 general-purpose registers. Parameters beyond the sixth word appear on the stack.

R9 [LR]	<b>Link address</b> is the location of the function call instruction and is used to calculate where program execution should return after function completion.
R10 [TLS]	<b>Thread Local Storage</b> host the address of this context's thread local storage structure. This mechanism, as normally provided by the compiler, allows designated variables to have one instance per thread.
R11 [RV]	<b>Return value</b> of the function. For <i>void</i> functions a value is not defined. For functions returning a union or structure, a pointer to the result is placed into return value register.
R12 [RVH]	<b>Return value high</b> of the function. For functions returning 32-bit values this register can be considered temporary register. Note that this holds the less significant bits on big-endian implementations; 32-bit values still go in RV.

On big-endian implementations, R11 is used for the high 32 bits of 64-bit return values and R12 is used for the low 32 bits. On little-endian implementations this is reversed. This matches register order with memory storage.

Furthermore, an OpenRISC 1000 implementation might have several sets of shadowed general-purpose registers. These shadowed registers are used for fast context switching and sets can be switched only by the operating system.

## 17.2.2 The Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. 17-5 shows the stack frame organization.

Position	Contents	Frame
FP + 4N	Parameter N	Previous
...	...	
FP + 0	First stack parameter	
FP – 4	Return address	Current
FP – 8	Previous FP value	
FP – 12	Function variables	
...	...	
SP + 0	Subfunction call parameters	
SP – 4 SP – 128	For use by leaf functions w/o function prologue/ epilogue	Future
SP – 132 SP – 2536	For use by exception handlers	

**Table 17-5. Stack Frame**

When no compiler optimization is in place, the stack pointer always points to the end of the latest allocated stack frame. However when optimization is in effect the stack pointer may not be updated, so that up to 128 bytes beyond the current stack pointer are in use.

Optimized code will in general not use the frame pointer, freeing it up for use as another temporary register.

All frames must be word aligned.

The first 128 bytes below the current stack frame are reserved for use by optimized code. Exception handlers must guarantee that they will not use this area.

### 17.2.3 Parameter Passing

Functions receive up to their first 6 arguments in general-purpose parameter registers. No register holds more than one argument, and 64-bit arguments use two adjacent words. If there are more than six words, the remaining arguments are passed on the stack. Structure and union arguments are passed as pointers.

All 64-bit arguments in a 32-bit system are passed using a pair of words when available, in the same way as for other arguments. 64-bit arguments are not aligned. For example *long long arg1*, *long arg2*, *long long arg3* are passed in the following way: *arg1* in *r3&r4*, *arg2* in *r5*, *arg3* in *r6&r7*.

On big-endian implementations the high 32 bits are passed in the lower numbered register of the pair. On little-endian implementations this is reversed.

Individual arguments are not split across registers and stack, and variadic arguments are always put on the stack. For example, `printf(char *fmt, ...)` only takes one register argument, `fmt`.

For C++, the first argument word is the `this` pointer.

### 17.2.4 Functions Returning Scalars or No Value

A function that returns an integral, pointer or vector/floating-point value places its result in the general-purpose RV register. *void* functions put no particular value in GPR[RV] register.

64-bit return values also use the RVH register, which is otherwise undefined and not preserved across function calls.

### 17.2.5 Functions Returning Structures or Unions

A function that returns a structure or union places the address of the structure or union in the general-purpose RV register.

A function that returns a structure by value expects the location where that structure is to be placed to be supplied in function parameter word 0 (R3).

## 17.3 Operating System Interface

### 17.3.1 Exception Interface

The OpenRISC 1000 exception mechanism allows the processor to change to supervisor mode as a result of external signals, errors or execution of certain instructions. When an exception occurs the following events happen:

- ✓ The address of the interrupted instruction, supervisor register and EA (when relevant) are saved into EPCR, ESR and EEAR registers

- ✓ The machine mode is changed to supervisor mode as per section 6.3, Exception Processing. This includes disabling MMUs and exceptions.
- ✓ The execution resumes from a predefined exception vector address which is different for every exception

Exception Type	Vector Offset[11:0]	SIGNAL	Example
Reset	0x100	None	Reset
Bus Error	0x200	SIGBUS	Unexisting physical location, bus parity error.
Data Page Fault	0x300	SIGSEGV	Unmapped data location or protection violation.
Instruction Page Fault	0x400	SIGSEGV	Unmapped instruction location or protection violation
Tick Timer Interrupt	0x500	None	Process scheduling
Alignment	0x600	SIGBUS	Unaligned data
Illegal Instruction	0x700	SIGILL	Illegal/unimplemented instruction
External Interrupt	0x800	None	Device has asserted an interrupt
D-TLB Miss	0x900	None	DTLB software reload needed
I-TLB Miss	0xA00	None	ITLB software reload needed
Range	0xB00	SIGSEGV	Arithmetic overflow
System Call	0xC00	None	Instruction l.sys
Trap	0xE00	SIGTRAP	Instruction l.trap or debug unit exception.

**Table 17-6. Hardware Exceptions and Signals**

The significant bits (31-12) of the vector offset address for each exception depend on the setting of the Supervision Register (SR)'s EPH bit and presence and setting of the Exception Vector Base Address Register (EVBAR), which can specify an offset. For example, in the absence of the EVBAR and with SR[EPH] clear, the offset is zero.

The operating system handles an exception either by completing the faulting exception in a manner transparent to the application, if possible, or by delivering a signal to the application. 17-6 shows how hardware exceptions can be mapped to signals if the operating system cannot complete the faulting exception.

## 17.3.2 Virtual Address Space

For user programs to execute in virtual address space, the memory management unit (MMU) must be enabled. The MMU translates virtual address generated by the running process into physical address. This allows the process to run anywhere in the physical memory and additionally page to a secondary storage.

Processes typically begin with three logical segments, commonly referred as “text”, “data” and “stack”. Additional segments may exist or can be created by the operating system.



### 17.3.3 Page Size

Memory is organized into pages, which are the system's smallest units of memory allocation. The basic page size is 8KB with some implementations supporting 16MB and 32GB pages.

### 17.3.4 Virtual Address Assignments

Processes have full access to the entire virtual address space. However the size of a process can be limited by several factors such as a process size limit parameter, available physical memory and secondary storage.

0xFFFF_FFFF	Reserved system area
Start of Stack Growing Down	Stack
Growing Up	Heap
	.bss
Start of Data Segments	.data
Start of Program Code	.text
Start of Dynamic Segment Area	Shared Objects
0x0000_2000	
0x0000_0000	Unmapped

**Table 17-7. Virtual Address Configuration**

Page at location 0x0 is usually reserved to catch dereferences of NULL pointers.

Usually the beginning address of “.text”, “.data” and “.bss” segments are defined when linking the executable file. The heap is adjusted with facilities such as *malloc* and *free*. The dynamic segment area is adjusted with *mmap*, and the stack size is limited with *setrlimit*.

### 17.3.5 Stack

Every process has its own stack that is not tied to a fixed area in its address space. Since the stack can change differently for each call of a process, a process should use the stack pointer in general-purpose register r1 to access stack data.

### 17.3.6 Processor Execution Modes

The OpenRISC 1000 provides two execution modes: user and supervisor. Processes run in user mode and the operating system's kernel runs in supervisor mode. A Process must execute the *l.sys* instruction to switch to supervisor mode, hence requesting service from the operating system. It is suggested that system calls use the same argument passing model as used with function calls, except additional register r11 specifies system call id.

## 17.4 Position-Independent Code

This section needs to be written. Position-independent code is desired for proper dynamic linking support, which remains to be implemented.

## 17.5 ELF

The OpenRISC tools use the ELF object file formats and DWARF debugging information formats, as described in *System V Application Binary Interface*, from the Santa Cruz Operation, Inc. ELF and DWARF provide a suitable basis for representing the information needed for embedded applications. Other object file formats are available, such as COFF. This section describes particular fields in the ELF and DWARF formats that differ from the base standards for those formats.

### 17.5.1 Header Convention

The *e\_machine* member of the ELF header contains the decimal value 33906 (hexadecimal 0x8472) that is defined as the name EM\_OR32.

The *e\_ident* member of the ELF header contains values as shown in 17-8.

OR32 ELF e_ident Fields		
e_ident[EI_CLASS]	ELFCLASS32	For all 32-bit implementations
e_ident[EI_DATA]	ELFDATA2MSB	For all implementations

**Table 17-8. e\_ident Field Values**

The *e\_flags* member of the ELF header contains values as shown in 17-9.

OR32 ELF e_flags		
HAS_RELOC	0x01	Contains relocation entries
EXEC_P	0x02	Is directly executable
HAS_LINENO	0x04	Has line number information
HAS_DEBUG	0x08	Has debugging information
HAS_SYMS	0x10	Has symbols
HAS_LOCALS	0x20	Has local symbols
DYNAMIC	0x40	Is dynamic object
WP_TEXT	0x80	Text section is write protected
D_PAGED	0x100	Is dynamically paged

Table 17-9. e\_flags Field Values

## 17.5.2 Sections

There are no OpenRISC section requirements beyond the base ELF standards.

## 17.5.3 Relocation

This section describes values and algorithms used for relocations. In particular, it describes values the compiler/assembler must leave in place and how the linker modifies those values.

Name	Value	Size	Calculation
R_OR32_NONE	0	0	None
R_OR32_32	1	32	A
R_OR32_16	2	16	A & 0xffff
R_OR32_8	3	8	A & 0xff
R_OR32_CONST	4	16	A & 0xffff
R_OR32_CONSTH	5	16	(A >> 16) & 0xffff
R_OR32_JUMPTARG	6	28	(S + A - P) >> 2

Key *S* indicates the final value assigned to the symbol referenced in the relocation record. Key *A* is the added value specified in the relocation record. Key *P* indicates the address of the relocation (e.g., the address being modified).

# 18 Machine code reference

This section contains a table of all instructions including their instruction format.

OPC	Instruction	Mnemonic	Function	Class
0x00	000000NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN	I.j	Jump	I
0x01	000001NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN	I.jal	Jump and Link	I
0x02	000010DDDDNNNNNNNNNNNNNNNNNNNNNNNNNNNN	I.adrp	Compute Instruction Relative Address	II
0x03	000011NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN	I.bnf	Branch if No Flag	I
0x04	000100NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN	I.bf	Branch if Flag	I
0x05	00010101-----KKKKKKKKKKKKKKKKKKKK	I.nop	No Operation	I
0x06	000110DDDD----0KKKKKKKKKKKKKKKKKKKK	I.movhi	Move Immediate High	I
0x06	000110DDDD----1000000000000000000000	I.macrc	MAC Read and Clear	II
0x08	001000000000000000KKKKKKKKKKKKKKKKKK	I.sys	System Call	I
0x08	001000010000000000KKKKKKKKKKKKKKKKKK	I.trap	Trap	II
0x08	001000100000000000000000000000000000	I.msinc	Memory Synchronization	II
0x08	001000101000000000000000000000000000	I.psinc	Pipeline Synchronization	II
0x08	001000110000000000000000000000000000	I.csinc	Context Synchronization	II
0x09	001001-----	I.rfe	Return From Exception	I
0x0A	001010-----1100----	Iv.cust1	Reserved for Custom Vector Instructions	II
0x0A	001010-----1101----	Iv.cust2	Reserved for Custom Vector Instructions	II
0x0A	001010-----1110----	Iv.cust3	Reserved for Custom Vector Instructions	II

OPC	Instruction	Mnemonic	Function	Class
0x0A	001010-----1111----	lv.cust4	Reserved for Custom Vector Instructions	II
0x0A	001010DDDDAAAAABBBBB---00010000	lv.all_eq.b	Vector Byte Elements All Equal	I
0x0A	001010DDDDAAAAABBBBB---00010001	lv.all_eq.h	Vector Half-Word Elements All Equal	I
0x0A	001010DDDDAAAAABBBBB---00010010	lv.all_ge.b	Vector Byte Elements All Greater Than or Equal To	I
0x0A	001010DDDDAAAAABBBBB---00010011	lv.all_ge.h	Vector Half-Word Elements All Greater Than or Equal To	I
0x0A	001010DDDDAAAAABBBBB---00010100	lv.all_gt.b	Vector Byte Elements All Greater Than	I
0x0A	001010DDDDAAAAABBBBB---00010101	lv.all_gt.h	Vector Half-Word Elements All Greater Than	I
0x0A	001010DDDDAAAAABBBBB---00010110	lv.all_le.b	Vector Byte Elements All Less Than or Equal To	I
0x0A	001010DDDDAAAAABBBBB---00010111	lv.all_le.h	Vector Half-Word Elements All Less Than or Equal To	I
0x0A	001010DDDDAAAAABBBBB---00011000	lv.all_lt.b	Vector Byte Elements All Less Than	I
0x0A	001010DDDDAAAAABBBBB---00011001	lv.all_lt.h	Vector Half-Word Elements All Less Than	I
0x0A	001010DDDDAAAAABBBBB---00011010	lv.all_ne.b	Vector Byte Elements All Not Equal	I
0x0A	001010DDDDAAAAABBBBB---00011011	lv.all_ne.h	Vector Half-Word Elements All Not Equal	I
0x0A	001010DDDDAAAAABBBBB---00100000	lv.any_eq.b	Vector Byte Elements Any Equal	I

OPC	Instruction	Mnemonic	Function	Class
0x0A	001010DDDDDAAAAABBBBB---00100001	lv.any_eq.h	Vector Half-Word Elements Any Equal	I
0x0A	001010DDDDDAAAAABBBBB---00100010	lv.any_ge.b	Vector Byte Elements Any Greater Than or Equal To	I
0x0A	001010DDDDDAAAAABBBBB---00100011	lv.any_ge.h	Vector Half-Word Elements Any Greater Than or Equal To	I
0x0A	001010DDDDDAAAAABBBBB---00100100	lv.any_gt.b	Vector Byte Elements Any Greater Than	I
0x0A	001010DDDDDAAAAABBBBB---00100101	lv.any_gt.h	Vector Half-Word Elements Any Greater Than	I
0x0A	001010DDDDDAAAAABBBBB---00100110	lv.any_le.b	Vector Byte Elements Any Less Than or Equal To	I
0x0A	001010DDDDDAAAAABBBBB---00100111	lv.any_le.h	Vector Half-Word Elements Any Less Than or Equal To	I
0x0A	001010DDDDDAAAAABBBBB---00101000	lv.any_lt.b	Vector Byte Elements Any Less Than	I
0x0A	001010DDDDDAAAAABBBBB---00101001	lv.any_lt.h	Vector Half-Word Elements Any Less Than	I
0x0A	001010DDDDDAAAAABBBBB---00101010	lv.any_ne.b	Vector Byte Elements Any Not Equal	I
0x0A	001010DDDDDAAAAABBBBB---00101011	lv.any_ne.h	Vector Half-Word Elements Any Not Equal	I
0x0A	001010DDDDDAAAAABBBBB---00110000	lv.add.b	Vector Byte Elements Add Signed	I
0x0A	001010DDDDDAAAAABBBBB---00110001	lv.add.h	Vector Half-Word Elements Add Signed	I
0x0A	001010DDDDDAAAAABBBBB---00110010	lv.adds.b	Vector Byte Elements Add Signed Saturated	I

OPC	Instruction	Mnemonic	Function	Class
0x0A	001010DDDDDAAAAABBBBB---00110011	lv.adds.h	Vector Half-Word Elements Add Signed Saturated	I
0x0A	001010DDDDDAAAAABBBBB---00110100	lv.addu.b	Vector Byte Elements Add Unsigned	I
0x0A	001010DDDDDAAAAABBBBB---00110101	lv.addu.h	Vector Half-Word Elements Add Unsigned	I
0x0A	001010DDDDDAAAAABBBBB---00110110	lv.addus.b	Vector Byte Elements Add Unsigned Saturated	I
0x0A	001010DDDDDAAAAABBBBB---00110111	lv.addus.h	Vector Half-Word Elements Add Unsigned Saturated	I
0x0A	001010DDDDDAAAAABBBBB---00111000	lv.and	Vector And	I
0x0A	001010DDDDDAAAAABBBBB---00111001	lv.avg.b	Vector Byte Elements Average	I
0x0A	001010DDDDDAAAAABBBBB---00111010	lv.avg.h	Vector Half-Word Elements Average	I
0x0A	001010DDDDDAAAAABBBBB---01000000	lv.cmp_eq.b	Vector Byte Elements Compare Equal	I
0x0A	001010DDDDDAAAAABBBBB---01000001	lv.cmp_eq.h	Vector Half-Word Elements Compare Equal	I
0x0A	001010DDDDDAAAAABBBBB---01000010	lv.cmp_ge.b	Vector Byte Elements Compare Greater Than or Equal To	I
0x0A	001010DDDDDAAAAABBBBB---01000011	lv.cmp_ge.h	Vector Half-Word Elements Compare Greater Than or Equal To	I
0x0A	001010DDDDDAAAAABBBBB---01000100	lv.cmp_gt.b	Vector Byte Elements Compare Greater Than	I
0x0A	001010DDDDDAAAAABBBBB---01000101	lv.cmp_gt.h	Vector Half-Word Elements Compare Greater Than	I

OPC	Instruction	Mnemonic	Function	Class
0x0A	001010DDDDDDAAAAABBBBB---01000110	lv.cmp_le.b	Vector Byte Elements Compare Less Than or Equal To	I
0x0A	001010DDDDDDAAAAABBBBB---01000111	lv.cmp_le.h	Vector Half-Word Elements Compare Less Than or Equal To	I
0x0A	001010DDDDDDAAAAABBBBB---01001000	lv.cmp_lt.b	Vector Byte Elements Compare Less Than	I
0x0A	001010DDDDDDAAAAABBBBB---01001001	lv.cmp_lt.h	Vector Half-Word Elements Compare Less Than	I
0x0A	001010DDDDDDAAAAABBBBB---01001010	lv.cmp_ne.b	Vector Byte Elements Compare Not Equal	I
0x0A	001010DDDDDDAAAAABBBBB---01001011	lv.cmp_ne.h	Vector Half-Word Elements Compare Not Equal	I
0x0A	001010DDDDDDAAAAABBBBB---01010100	lv.madds.h	Vector Half-Word Elements Multiply Add Signed Saturated	I
0x0A	001010DDDDDDAAAAABBBBB---01010101	lv.max.b	Vector Byte Elements Maximum	I
0x0A	001010DDDDDDAAAAABBBBB---01010110	lv.max.h	Vector Half-Word Elements Maximum	I
0x0A	001010DDDDDDAAAAABBBBB---01010111	lv.merge.b	Vector Byte Elements Merge	I
0x0A	001010DDDDDDAAAAABBBBB---01011000	lv.merge.h	Vector Half-Word Elements Merge	I
0x0A	001010DDDDDDAAAAABBBBB---01011001	lv.min.b	Vector Byte Elements Minimum	I
0x0A	001010DDDDDDAAAAABBBBB---01011010	lv.min.h	Vector Half-Word Elements Minimum	I



OPC	Instruction	Mnemonic	Function	Class
0x0A	001010DDDDAAAAABBBBB---01011011	lv.msubs.h	Vector Half-Word Elements Multiply Subtract Signed Saturated	I
0x0A	001010DDDDAAAAABBBBB---01011100	lv.muls.h	Vector Half-Word Elements Multiply Signed Saturated	II
0x0A	001010DDDDAAAAABBBBB---01011101	lv.nand	Vector Not And	I
0x0A	001010DDDDAAAAABBBBB---01011110	lv.nor	Vector Not Or	I
0x0A	001010DDDDAAAAABBBBB---01011111	lv.or	Vector Or	I
0x0A	001010DDDDAAAAABBBBB---01100000	lv.pack.b	Vector Byte Elements Pack	I
0x0A	001010DDDDAAAAABBBBB---01100001	lv.pack.h	Vector Half-word Elements Pack	I
0x0A	001010DDDDAAAAABBBBB---01100010	lv.packs.b	Vector Byte Elements Pack Signed Saturated	I
0x0A	001010DDDDAAAAABBBBB---01100011	lv.packs.h	Vector Half-word Elements Pack Signed Saturated	I
0x0A	001010DDDDAAAAABBBBB---01100100	lv.packus.b	Vector Byte Elements Pack Unsigned Saturated	I
0x0A	001010DDDDAAAAABBBBB---01100101	lv.packus.h	Vector Half-word Elements Pack Unsigned Saturated	I
0x0A	001010DDDDAAAAABBBBB---01100110	lv.perm.n	Vector Nibble Elements Permute	I
0x0A	001010DDDDAAAAABBBBB---01100111	lv.rl.b	Vector Byte Elements Rotate Left	I
0x0A	001010DDDDAAAAABBBBB---01101000	lv.rl.h	Vector Half-Word Elements Rotate Left	I
0x0A	001010DDDDAAAAABBBBB---01101001	lv.sll.b	Vector Byte Elements Shift Left Logical	I

OPC	Instruction	Mnemonic	Function	Class
0x0A	001010DDDDDDAAAAABBBBB---01101010	lv.sll.h	Vector Half-Word Elements Shift Left Logical	I
0x0A	001010DDDDDDAAAAABBBBB---01101011	lv.sll	Vector Shift Left Logical	I
0x0A	001010DDDDDDAAAAABBBBB---01101100	lv.srl.b	Vector Byte Elements Shift Right Logical	I
0x0A	001010DDDDDDAAAAABBBBB---01101101	lv.srl.h	Vector Half-Word Elements Shift Right Logical	I
0x0A	001010DDDDDDAAAAABBBBB---01101110	lv.sra.b	Vector Byte Elements Shift Right Arithmetic	I
0x0A	001010DDDDDDAAAAABBBBB---01101111	lv.sra.h	Vector Half-Word Elements Shift Right Arithmetic	I
0x0A	001010DDDDDDAAAAABBBBB---01110000	lv.srl	Vector Shift Right Logical	I
0x0A	001010DDDDDDAAAAABBBBB---01110001	lv.sub.b	Vector Byte Elements Subtract Signed	I
0x0A	001010DDDDDDAAAAABBBBB---01110010	lv.sub.h	Vector Half-Word Elements Subtract Signed	I
0x0A	001010DDDDDDAAAAABBBBB---01110011	lv.subs.b	Vector Byte Elements Subtract Signed Saturated	I
0x0A	001010DDDDDDAAAAABBBBB---01110100	lv.subs.h	Vector Half-Word Elements Subtract Signed Saturated	I
0x0A	001010DDDDDDAAAAABBBBB---01110101	lv.subu.b	Vector Byte Elements Subtract Unsigned	I
0x0A	001010DDDDDDAAAAABBBBB---01110110	lv.subu.h	Vector Half-Word Elements Subtract Unsigned	I

OPC	Instruction	Mnemonic	Function	Class
0x0A	001010DDDDDAAAAABBBBB---01110111	lv.subus.b	Vector Byte Elements Subtract Unsigned Saturated	I
0x0A	001010DDDDDAAAAABBBBB---01111000	lv.subus.h	Vector Half-Word Elements Subtract Unsigned Saturated	I
0x0A	001010DDDDDAAAAABBBBB---01111001	lv.unpack.b	Vector Byte Elements Unpack	I
0x0A	001010DDDDDAAAAABBBBB---01111010	lv.unpack.h	Vector Half-Word Elements Unpack	I
0x0A	001010DDDDDAAAAABBBBB---01111011	lv.xor	Vector Exclusive Or	I
0x11	010001-----BBBBB-----	l.jr	Jump Register	I
0x12	010010-----BBBBB-----	l.jalr	Jump and Link Register	I
0x13	010011-----AAAAAIIIIIIIIIIIIIIIIIIII	l.maci	Multiply Immediate Signed and Accumulate	I
0x1A	011010DDDDDAAAAIIIIIIIIIIIIIIIIIIII	l.lf	Load Single Float Word with NaN Boxing	I
0x1B	011011DDDDDAAAAIIIIIIIIIIIIIIIIIIII	l.lwa	Load Single Word Atomic	I
0x1C	011100-----	l.cust1	Reserved for ORBIS32/64 Custom Instructions	I
0x1D	011101-----	l.cust2	Reserved for ORBIS32/64 Custom Instructions	I
0x1E	011110-----	l.cust3	Reserved for ORBIS32/64 Custom Instructions	I
0x1F	011111-----	l.cust4	Reserved for ORBIS32/64 Custom Instructions	I
0x20	100000DDDDDAAAAIIIIIIIIIIIIIIIIIIII	l.ld	Load Double Word	I

OPC	Instruction	Mnemonic	Function	Class
0x21	100001DDDDDAAAAAIIIIIIIIIIIIIIIIIIII	I.lwz	Load Single Word and Extend with Zero	I
0x22	100010DDDDDAAAAAIIIIIIIIIIIIIIIIIIII	I.lws	Load Single Word and Extend with Sign	I
0x23	100011DDDDDAAAAAIIIIIIIIIIIIIIIIIIII	I.lbz	Load Byte and Extend with Zero	I
0x24	100100DDDDDAAAAAIIIIIIIIIIIIIIIIIIII	I.lbs	Load Byte and Extend with Sign	I
0x25	100101DDDDDAAAAAIIIIIIIIIIIIIIIIIIII	I.lhz	Load Half Word and Extend with Zero	I
0x26	100110DDDDDAAAAAIIIIIIIIIIIIIIIIIIII	I.lhs	Load Half Word and Extend with Sign	I
0x27	100111DDDDDAAAAAIIIIIIIIIIIIIIIIIIII	I.addi	Add Immediate Signed	I
0x28	101000DDDDDAAAAAIIIIIIIIIIIIIIIIIIII	I.addic	Add Immediate Signed and Carry	I
0x29	101001DDDDDAAAAAKKKKKKKKKKKKKKKKKK	I.andi	And with Immediate Half Word	I
0x2A	101010DDDDDAAAAAKKKKKKKKKKKKKKKKKK	I.ori	Or with Immediate Half Word	I
0x2B	101011DDDDDAAAAAIIIIIIIIIIIIIIIIIIII	I.xori	Exclusive Or with Immediate Half Word	I
0x2C	101100DDDDDAAAAAIIIIIIIIIIIIIIIIIIII	I.muli	Multiply Immediate Signed	II
0x2D	101101DDDDDAAAAAKKKKKKKKKKKKKKKKKK	I.mfspr	Move From Special-Purpose Register	I
0x2E	101110DDDDDAAAAA-----00LLLLLL	I.slli	Shift Left Logical with Immediate	I
0x2E	101110DDDDDAAAAA-----01LLLLLL	I.srli	Shift Right Logical with Immediate	I
0x2E	101110DDDDDAAAAA-----10LLLLLL	I.srai	Shift Right Arithmetic with Immediate	I

OPC	Instruction	Mnemonic	Function	Class
0x2E	101110DDDDAAAAA-----11LLLLLL	I.rori	Rotate Right with Immediate	II
0x2F	10111100000AAAAAIIIIIIIIIIIIIIIIII	I.sfeqi	Set Flag if Equal Immediate	I
0x2F	10111100001AAAAAIIIIIIIIIIIIIIIIII	I.sfnei	Set Flag if Not Equal Immediate	I
0x2F	10111100010AAAAAIIIIIIIIIIIIIIIIII	I.sfgtui	Set Flag if Greater Than Immediate Unsigned	I
0x2F	10111100011AAAAAIIIIIIIIIIIIIIIIII	I.sfgeui	Set Flag if Greater or Equal Than Immediate Unsigned	I
0x2F	10111100100AAAAAIIIIIIIIIIIIIIIIII	I.sfltui	Set Flag if Less Than Immediate Unsigned	I
0x2F	10111100101AAAAAIIIIIIIIIIIIIIIIII	I.sfleui	Set Flag if Less or Equal Than Immediate Unsigned	I
0x2F	10111101010AAAAAIIIIIIIIIIIIIIIIII	I.sfgtsi	Set Flag if Greater Than Immediate Signed	I
0x2F	10111101011AAAAAIIIIIIIIIIIIIIIIII	I.sfgesi	Set Flag if Greater or Equal Than Immediate Signed	I
0x2F	10111101100AAAAAIIIIIIIIIIIIIIIIII	I.sfltsi	Set Flag if Less Than Immediate Signed	I
0x2F	10111101101AAAAAIIIIIIIIIIIIIIIIII	I.sflesi	Set Flag if Less or Equal Than Immediate Signed	I
0x30	110000KKKKKAAAAABBBBBKKKKKKKKKKKK	I.mtspr	Move To Special-Purpose Register	I
0x31	110001-----AAAAABBBBB-----0001	I.mac	Multiply Signed and Accumulate	II
0x31	110001-----AAAAABBBBB-----0011	I.macu	Multiply Unsigned and Accumulate	II
0x31	110001-----AAAAABBBBB-----0010	I.msb	Multiply Signed and Subtract	II

OPC	Instruction	Mnemonic	Function	Class
0x31	110001-----AAAAABBBBB-----0100	l.msbu	Multiply Unsigned and Subtract	II
0x32	110010-----AAAAABBBBB---00001000	lf.sfeq.s	Set Flag if Equal Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBB---00001001	lf.sfne.s	Set Flag if Not Equal Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBB---00001010	lf.sfgt.s	Set Flag if Greater Than Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBB---00001011	lf.sfge.s	Set Flag if Greater or Equal Than Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBB---00001100	lf.sflt.s	Set Flag if Less Than Floating-Point Single-Precision	I
0x32	110010-----AAAAABBBBB---00001101	lf.sfle.s	Set Flag if Less or Equal Than Floating-Point Single-Precision	I
0x32	110010-----AAAAABBBBB-0000011000	lf.sfeq.d	Set Flag if Equal Floating-Point Double-Precision	I
0x32	110010-----AAAAABBBBB-0000011001	lf.sfne.d	Set Flag if Not Equal Floating-Point Double-Precision	I
0x32	110010-----AAAAABBBBB-0000011010	lf.sfgt.d	Set Flag if Greater Than Floating-Point Double-Precision	I
0x32	110010-----AAAAABBBBB-0000011011	lf.sfge.d	Set Flag if Greater or Equal Than Floating-Point Double-Precision	I
0x32	110010-----AAAAABBBBB-0000011100	lf.sflt.d	Set Flag if Less Than Floating-Point Double-Precision	I

OPC	Instruction	Mnemonic	Function	Class
0x32	110010-----AAAAABBBBB-0000011101	If.sfle.d	Set Flag if Less or Equal Than Floating-Point Double-Precision	I
0x32	110010-----AAAAABBBBB---00101000	If.sfueq.s	Set Flag if Unordered or Equal Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBB---00101001	If.sfune.s	Set Flag if Unordered or Not Equal Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBB---00101010	If.sfugt.s	Set Flag if Unordered or Greater Than Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBB---00101011	If.sfuge.s	Set Flag if Unordered or Greater Than or Equal Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBB---00101100	If.sfult.s	Set Flag if Unordered or Less Than Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBB---00101101	If.sfule.s	Set Flag if Unordered or Less Than or Equal Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBB---00101110	If.sfun.s	Set Flag if Unordered Floating-Point Single-Precision	II
0x32	110010-----AAAAABBBBBO--00110100	If.stod.d	Convert Single-precision Floating-Point Number To Double-precision	II
0x32	110010-----AAAAABBBBB-O-00110101	If.dtos.d	Convert Double-precision Floating-Point Number to Single-precision	II
0x32	110010-----AAAAABBBBB-0000111000	If.sfueq.d	Set Flag if Unordered or Equal Floating-Point Double-Precision	II

OPC	Instruction	Mnemonic	Function	Class
0x32	110010-----AAAAABBBBB-0000111001	If.sfune.d	Set Flag if Unordered or Not Equal Floating-Point Double-Precision	II
0x32	110010-----AAAAABBBBB-0000111010	If.sfugt.d	Set Flag if Unordered or Greater Than Floating-Point Double-Precision	II
0x32	110010-----AAAAABBBBB-0000111011	If.sfuge.d	Set Flag if Unordered or Greater Than or Equal Floating-Point Double-Precision	II
0x32	110010-----AAAAABBBBB-0000111100	If.sfult.d	Set Flag if Unordered or Less Than Floating-Point Double-Precision	II
0x32	110010-----AAAAABBBBB-0000111101	If.sfule.d	Set Flag if Unordered or Less Than or Equal Floating-Point Double-Precision	II
0x32	110010-----AAAAABBBBB-0000111110	If.sfun.d	Set Flag if Unordered Floating-Point Double-Precision	II
0x32	110010-----AAAAABBBBB0001101----	If.cust1.s	Reserved for ORFPX32 Custom Instructions	II
0x32	110010-----AAAAABBBBB0001110----	If.cust1.d	Reserved for ORFPX64 Custom Instructions	II
0x32	110010DDDDDAAAAA00000---00000100	If.itof.s	Integer To Floating-Point Single-Precision	I
0x32	110010DDDDDAAAAA00000---00000101	If.ftoi.s	Floating-Point Single-Precision To Integer	I
0x32	110010DDDDDAAAAA0000000-00010100	If.itof.d	Integer To Floating-Point Double-Precision	I
0x32	110010DDDDDAAAAA0000000-00010101	If.ftoi.d	Floating-Point Double-Precision To Integer	I
0x32	110010DDDDDAAAAAABBBBB---00000000	If.add.s	Add Floating-Point Single-Precision	I



OPC	Instruction	Mnemonic	Function	Class
0x32	110010DDDDDAAAAABBBBB---00000001	lf.sub.s	Subtract Floating-Point Single-Precision	I
0x32	110010DDDDDAAAAABBBBB---00000010	lf.mul.s	Multiply Floating-Point Single-Precision	I
0x32	110010DDDDDAAAAABBBBB---00000011	lf.div.s	Divide Floating-Point Single-Precision	II
0x32	110010DDDDDAAAAABBBBB---00000110	lf.rem.s	Remainder Floating-Point Single-Precision	II
0x32	110010DDDDDAAAAABBBBB---00000111	lf.madd.s	Multiply and Add Floating-Point Single-Precision	II
0x32	110010DDDDDAAAAABBBBBBOOO00010000	lf.add.d	Add Floating-Point Double-Precision	I
0x32	110010DDDDDAAAAABBBBBBOOO00010001	lf.sub.d	Subtract Floating-Point Double-Precision	I
0x32	110010DDDDDAAAAABBBBBBOOO00010010	lf.mul.d	Multiply Floating-Point Double-Precision	II
0x32	110010DDDDDAAAAABBBBBBOOO00010011	lf.div.d	Divide Floating-Point Double-Precision	II
0x32	110010DDDDDAAAAABBBBBBOOO00010110	lf.rem.d	Remainder Floating-Point Double-Precision	II
0x32	110010DDDDDAAAAABBBBBBOOO00010111	lf.madd.d	Multiply and Add Floating-Point Double-Precision	II
0x33	110011IIIIIAAAAABBBBBIIIIIIIIII	l.swa	Store Single Word Atomic	II
0x35	110101IIIIIAAAAABBBBBIIIIIIIIII	l.sw	Store Single Word	I
0x36	110110IIIIIAAAAABBBBBIIIIIIIIII	l.sb	Store Byte	I
0x37	110111IIIIIAAAAABBBBBIIIIIIIIII	l.sh	Store Half Word	I
0x38	111000DDDDDAAAA-----0000--1100	l.exths	Extend Half Word with Sign	II

OPC	Instruction	Mnemonic	Function	Class
0x38	111000DDDDAAAAA-----0000--1101	I.extws	Extend Word with Sign	II
0x38	111000DDDDAAAAA-----0001--1100	I.extbs	Extend Byte with Sign	II
0x38	111000DDDDAAAAA-----0001--1101	I.extwz	Extend Word with Zero	II
0x38	111000DDDDAAAAA-----0010--1100	I.exthz	Extend Half Word with Zero	II
0x38	111000DDDDAAAAA-----0011--1100	I.extbz	Extend Byte with Zero	II
0x38	111000DDDDAAAAABBBBB-00----0000	I.add	Add Signed	I
0x38	111000DDDDAAAAABBBBB-00----0001	I.addc	Add Signed and Carry	I
0x38	111000DDDDAAAAABBBBB-00----0010	I.sub	Subtract Signed	I
0x38	111000DDDDAAAAABBBBB-00----0011	I.and	And	I
0x38	111000DDDDAAAAABBBBB-00----0100	I.or	Or	I
0x38	111000DDDDAAAAABBBBB-00----0101	I.xor	Exclusive Or	I
0x38	111000DDDDAAAAABBBBB-00----1110	I.cmov	Conditional Move	II
0x38	111000DDDDAAAAA-----00----1111	I.ff1	Find First 1	II
0x38	111000DDDDAAAAABBBBB-0000--1000	I.sll	Shift Left Logical	I
0x38	111000DDDDAAAAABBBBB-0001--1000	I.srl	Shift Right Logical	I
0x38	111000DDDDAAAAABBBBB-0010--1000	I.sra	Shift Right Arithmetic	I
0x38	111000DDDDAAAAABBBBB-0011--1000	I.ror	Rotate Right	II
0x38	111000DDDDAAAAA-----01----1111	I.fl1	Find Last 1	II
0x38	111000DDDDAAAAABBBBB-11----0110	I.mul	Multiply Signed	II
0x38	111000-----AAAAABBBBB-11----0111	I.muld	Multiply Signed to Double	II
0x38	111000DDDDAAAAABBBBB-11----1001	I.div	Divide Signed	II

OPC	Instruction	Mnemonic	Function	Class
0x38	111000DDDDAAAAABBBBB-11----1010	I.divu	Divide Unsigned	II
0x38	111000DDDDAAAAABBBBB-11----1011	I.mulu	Multiply Unsigned	II
0x38	111000-----AAAAABBBBB-11----1100	I.muldu	Multiply Unsigned to Double	II
0x39	11100100000AAAAABBBBB-----	I.sfeq	Set Flag if Equal	I
0x39	11100100001AAAAABBBBB-----	I.sfne	Set Flag if Not Equal	I
0x39	11100100010AAAAABBBBB-----	I.sfgtu	Set Flag if Greater Than Unsigned	I
0x39	11100100011AAAAABBBBB-----	I.sfgeu	Set Flag if Greater or Equal Than Unsigned	I
0x39	11100100100AAAAABBBBB-----	I.sfltu	Set Flag if Less Than Unsigned	I
0x39	11100100101AAAAABBBBB-----	I.sfleu	Set Flag if Less or Equal Than Unsigned	I
0x39	11100101010AAAAABBBBB-----	I.sfgts	Set Flag if Greater Than Signed	I
0x39	11100101011AAAAABBBBB-----	I.sfges	Set Flag if Greater or Equal Than Signed	I
0x39	11100101100AAAAABBBBB-----	I.sflts	Set Flag if Less Than Signed	I
0x39	11100101101AAAAABBBBB-----	I.sfls	Set Flag if Less or Equal Than Signed	I
0x3C	111100DDDDAAAAABBBBBLLLLLLKKKKK	I.cust5	Reserved for ORBIS32/64 Custom Instructions	II
0x3D	111101-----	I.cust6	Reserved for ORBIS32/64 Custom Instructions	II
0x3E	111110-----	I.cust7	Reserved for ORBIS32/64 Custom Instructions	II

OPC	Instruction	Mnemonic	Function	Class
0x3F	111111-----	I.cust8	Reserved for ORBIS32/64 Custom Instructions	II

# 19 Index

## Instruction mnemonics

l.add.....37	l.lhz.....75	l.sflts.....116
l.addc.....38	l.lwa.....76	l.sfltsi.....116
l.addi.....39	l.lws.....77	l.sfltui.....118
l.addic.....40	l.lwz.....78	l.sfltui.....118
l.adrp.....41	l.mac.....79	l.sfne.....120
l.and.....41	l.maci.....79	l.sfnei.....120
l.andi.....43	l.macrc.....81	l.sh.....122
l.bf.....44	l.mfspr.....83	l.sll.....123
l.bnf.....45	l.movhi.....84	l.slli.....124
l.cmov.....45	l.msb.....84	l.sra.....125
l.csync.....46	l.msync.....87	l.srai.....126
l.cust1.....48	l.mtspr.....88	l.srl.....127
l.cust2.....49	l.mul.....88	l.srli.....128
l.cust3.....50	l.muli.....92	l.sub.....129
l.cust4.....51	l.mulu.....93	l.sw.....130
l.cust5.....52	l.nop.....94	l.swa.....131
l.cust6.....53	l.or.....95	l.sys.....132
l.cust7.....54	l.ori.....96	l.trap.....133
l.cust8.....55	l.psync.....96	l.xor.....134
l.div.....56	l.rfe.....97	l.xori.....135
l.divu.....57	l.ror.....98	lf.add.d.....136
l.extbs.....57	l.rori.....99	lf.add.s.....136
l.extbz.....59	l.sb.....100	lf.cust1.d.....137
l.exths.....60	l.sd.....100	lf.cust1.s.....138
l.exthz.....61	l.sfeq.....102	lf.div.d.....138
l.extws.....62	l.sfeqi.....102	lf.div.s.....140
l.extwz.....63	l.sfges.....104	lf.dtos.d.....140
l.ffl.....64	l.sfgesi.....104	lf.ftoi.d.....142
l.fll.....65	l.sfgeu.....106	lf.ftoi.s.....142
l.j.....66	l.sfgeui.....106	lf.itof.d.....144
l.jal.....67	l.sfgts.....108	lf.itof.s.....144
l.jalr.....68	l.sfgtsi.....108	lf.madd.d.....145
l.jr.....69	l.sfgtu.....110	lf.madd.s.....147
l.lbs.....70	l.sfgtui.....110	lf.mul.d.....148
l.lbz.....71	l.sfles.....112	lf.mul.s.....148
l.ld.....72	l.sflesi.....112	lf.rem.d.....149
l.lf.....72	l.sfleu.....114	lf.rem.s.....150
l.lhs.....73	l.sfleui.....114	lf.sfeq.d.....150

lf.sfeq.s.....151	lv.all_gt.h.....189	lv.max.b.....228
lf.sfge.d.....152	lv.all_le.b.....190	lv.max.h.....229
lf.sfge.s.....153	lv.all_le.h.....191	lv.merge.b.....230
lf.sfgt.d.....153	lv.all_lt.b.....192	lv.merge.h.....231
lf.sfgt.s.....154	lv.all_lt.h.....193	lv.min.b.....232
lf.sfle.d.....155	lv.all_ne.b.....194	lv.min.h.....233
lf.sfle.s.....155	lv.all_ne.h.....195	lv.msubs.h.....234
lf.sflt.d.....156	lv.and.....196	lv.muls.h.....235
lf.sflt.s.....157	lv.any_eq.b.....197	lv.nand.....236
lf.sfne.d.....157	lv.any_eq.h.....198	lv.nor.....237
lf.sfne.s.....159	lv.any_ge.b.....199	lv.or.....238
lf.sfueq.d.....159	lv.any_ge.h.....200	lv.pack.b.....238
lf.sfueq.s.....161	lv.any_gt.b.....201	lv.pack.h.....240
lf.sfuge.d.....162	lv.any_gt.h.....202	lv.packs.b.....241
lf.sfuge.s.....163	lv.any_le.b.....203	lv.packs.h.....242
lf.sfugt.d.....164	lv.any_le.h.....204	lv.packus.b.....243
lf.sfugt.s.....165	lv.any_lt.b.....205	lv.packus.h.....244
lf.sfule.d.....166	lv.any_lt.h.....206	lv.perm.n.....245
lf.sfuge.s.....167	lv.any_ne.b.....207	lv.rl.b.....246
lf.sfult.d.....168	lv.any_ne.h.....208	lv.rl.h.....247
lf.sfult.s.....168	lv.avg.b.....209	lv.sll.....248
lf.sfun.d.....170	lv.avg.h.....210	lv.sll.b.....249
lf.sfun.s.....170	lv.cmp_eq.b.....211	lv.sll.h.....250
lf.stod.d.....172	lv.cmp_eq.h.....212	lv.sra.b.....251
lf.sub.d.....173	lv.cmp_ge.b.....213	lv.sra.h.....252
lf.sub.s.....173	lv.cmp_ge.h.....214	lv.srl.....253
lv.add.b.....176	lv.cmp_gt.b.....215	lv.srl.b.....254
lv.add.h.....177	lv.cmp_gt.h.....216	lv.srl.h.....255
lv.adds.b.....178	lv.cmp_le.b.....217	lv.sub.b.....256
lv.adds.h.....179	lv.cmp_le.h.....218	lv.sub.h.....257
lv.addu.b.....180	lv.cmp_lt.b.....219	lv.subs.b.....258
lv.addu.h.....181	lv.cmp_lt.h.....220	lv.subs.h.....259
lv.addus.b.....182	lv.cmp_ne.b.....221	lv.subu.b.....260
lv.addus.h.....183	lv.cmp_ne.h.....222	lv.subu.h.....261
lv.all_eq.b.....184	lv.cust1.....223	lv.subus.b.....262
lv.all_eq.h.....185	lv.cust2.....224	lv.subus.h.....263
lv.all_ge.b.....186	lv.cust3.....225	lv.unpack.b.....264
lv.all_ge.h.....187	lv.cust4.....226	lv.unpack.h.....265
lv.all_gt.b.....188	lv.madds.h.....227	lv.xor.....266