# OpenRISC 1000 Architecture Manual[1]

April 4, 2006

Copyright (C) 2000, 2001, 2002, 2003, 2004 OPENCORES.ORG 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

**1 ABOUT THIS MANUAL ........ 10**
1.1 INTRODUCTION ........ 10
1.2 AUTHORS ........ 10
1.3 REVISION HISTORY ........ 11
1.4 WORK IN PROGRESS ........ 12
1.5 FONTS IN THIS MANUAL ........ 12
1.6 CONVENTIONS ........ 13
1.7 NUMBERING ........ 13
**2 ARCHITECTURE OVERVIEW ........ 14**
2.1 FEATURES ........ 14
2.2 INTRODUCTION ........ 14
**3 ADDRESSING MODES AND OPERAND CONVENTIONS ........ 16**
3.1 MEMORY ADDRESSING MODES ........ 16
**3.1.1 Register Indirect with Displacement ........ 16**
**3.1.2 PC Relative ........ 17**
3.2 MEMORY OPERAND CONVENTIONS ........ 17
**3.2.1 Bit and Byte Ordering ........ 18**
**3.2.2 Aligned and Misaligned Accesses ........ 19**
**4 REGISTER SET ........ 20**
4.1 FEATURES ........ 20
4.2 OVERVIEW ........ 20
4.3 SPECIAL-PURPOSE REGISTERS ........ 20
4.4 GENERAL-PURPOSE REGISTERS (GPRS) ........ 24
4.5 SUPPORT FOR CUSTOM NUMBER OF GPRS ........ 25
4.6 SUPERVISION REGISTER (SR) ........ 25
4.7 EXCEPTION PROGRAM COUNTER REGISTERS (EPCR0 - EPCR15) ........ 27
4.8 EXCEPTION EFFECTIVE ADDRESS REGISTERS (EEAR0-EEAR15) ........ 27
4.9 EXCEPTION SUPERVISION REGISTERS (ESR0-ESR15) ........ 28
4.10 NEXT AND PREVIOUS PROGRAM COUNTER (NPC AND PPC) ........ 28
4.11 FLOATING POINT CONTROL STATUS REGISTER (FPCSR) ........ 28
**5 INSTRUCTION SET ........ 31**
5.1 FEATURES ........ 31
5.2 OVERVIEW ........ 31
5.3 ORBIS32/64 ........ 33
**6 EXCEPTION MODEL ........ 252**
6.1 INTRODUCTION ........ 252
6.2 EXCEPTION CLASSES ........ 252

6.3 EXCEPTION PROCESSING....254
6.4 FAST CONTEXT SWITCHING (OPTIONAL)....255
**6.4.1 Changing Context in Supervisor Mode....255**
**6.4.2 Context Switch Caused by Exception....256**
**6.4.3 Accessing Other Contexts' Registers....257**
**7 MEMORY MODEL....258**
7.1 MEMORY....258
7.2 MEMORY ACCESS ORDERING....258
**7.2.1 Memory Synchronize Instruction....258**
**7.2.2 Pages Designated as Weakly-Ordered-Memory....258**
**8 MEMORY MANAGEMENT....260**
8.1 MMU FEATURES....260
8.2 MMU OVERVIEW....260
8.3 MMU EXCEPTIONS....262
8.4 MMU SPECIAL-PURPOSE REGISTERS....262
**8.4.1 Data MMU Control Register (DMMUCR)....264**
**8.4.2 Data MMU Protection Register (DMMUPR)....264**
**8.4.3 Instruction MMU Control Register (IMMUCR)....265**
**8.4.4 Instruction MMU Protection Register (IMMUPR)....266**
**8.4.5 Instruction/Data TLB Entry Invalidate Registers (xTLBEIR)....267**
**8.4.6 Instruction/Data Translation Lookaside Buffer Way y Match Registers (xTLBWyMR0-xTLBWyMR127)....268**
**8.4.7 Data Translation Lookaside Buffer Way y Translate Registers (DTLBWyTR0-DTLBWyTR127)....270**
**8.4.8 Instruction Translation Lookaside Buffer Way y Translate Registers (ITLBWyTR0-ITLBWyTR127)....271**
**8.4.9 Instruction/Data Area Translation Buffer Match Registers (xATBMR0-xATBMR3)....272**
**8.4.10 Data Area Translation Buffer Translate Registers (DATBTR0-DATBTR3) .274**
**8.4.11 Instruction Area Translation Buffer Translate Registers (IATBTR0-IATBTR3) 275**
8.5 ADDRESS TRANSLATION MECHANISM IN 32-BIT IMPLEMENTATIONS....276
8.6 ADDRESS TRANSLATION MECHANISM IN 64-BIT IMPLEMENTATIONS....280
8.7 MEMORY PROTECTION MECHANISM....283
8.8 PAGE TABLE ENTRY DEFINITION....284
8.9 PAGE TABLE SEARCH OPERATION....286
8.10 PAGE HISTORY RECORDING....286
8.11 PAGE TABLE UPDATES....286
**9 CACHE MODEL & CACHE COHERENCY....288**
9.1 CACHE SPECIAL-PURPOSE REGISTERS....288
**9.1.1 Data Cache Control Register....289**
**9.1.2 Instruction Cache Control Register....289**
9.2 CACHE MANAGEMENT....290
**9.2.1 Data Cache Block Prefetch (Optional)....290**
**9.2.2 Data Cache Block Flush....291**
**9.2.3 Data Cache Block Invalidate....292**

**9.2.4 Data Cache Block Write-Back ....293**
**9.2.5 Data Cache Block Lock (Optional) ....293**
**9.2.6 Instruction Cache Block Prefetch (Optional) ....294**
**9.2.7 Instruction Cache Block Invalidate ....294**
**9.2.8 Instruction Cache Block Lock (Optional) ....295**
9.3 CACHE/MEMORY COHERENCY ....296
**9.3.1 Pages Designated as Cache Coherent Pages ....296**
**9.3.2 Pages Designated as Caching-Inhibited Pages ....296**
**9.3.3 Pages Designated as Write-Back Cache Pages ....297**
**10 DEBUG UNIT (OPTIONAL) ....298**
10.1 FEATURES ....298
10.2 DEBUG VALUE REGISTERS (DVR0-DVR7) ....299
10.3 DEBUG CONTROL REGISTERS (DCR0-DCR7) ....300
10.4 DEBUG MODE REGISTER 1 (DMR1) ....301
10.5 DEBUG MODE REGISTER 2(DMR2) ....303
10.6 DEBUG WATCHPOINT COUNTER REGISTER (DWCR0-DWCR1) ....304
10.7 DEBUG STOP REGISTER (DSR) ....304
10.8 DEBUG REASON REGISTER (DRR) ....306
**11 PERFORMANCE COUNTERS UNIT (OPTIONAL) ....308**
11.1 FEATURES ....308
11.2 PERFORMANCE COUNTERS COUNT REGISTERS (PCCR0-PCCR7) ....308
11.3 PERFORMANCE COUNTERS MODE REGISTERS (PCMR0-PCMR7) ....309
**12 POWER MANAGEMENT (OPTIONAL) ....311**
12.1 FEATURES ....311
12.2 POWER MANAGEMENT REGISTER (PMR) ....312
**13 PROGRAMMABLE INTERRUPT CONTROLLER (OPTIONAL) ....313**
13.1 FEATURES ....313
13.2 PIC MASK REGISTER (PICMR) ....313
13.3 PIC STATUS REGISTER (PICSR) ....314
**14 TICK TIMER FACILITY (OPTIONAL) ....316**
14.1 FEATURES ....316
14.2 TIMER INTERRUPTS ....317
14.3 TIMER MODES ....317
**14.3.1 Disabled timer ....317**
**14.3.2 Auto-restart timer ....317**
**14.3.3 One-shot timer ....317**
**14.3.4 Continuous timer ....318**
14.4 TICK TIMER MODE REGISTER (TTMR) ....318
14.5 TICK TIMER COUNT REGISTER (TTCR) ....319
**15 OPENRISC 1000 IMPLEMENTATIONS ....320**
15.1 OVERVIEW ....320
15.2 VERSION REGISTER (VR) ....320
15.3 UNIT PRESENT REGISTER (UPR) ....321
15.4 CPU CONFIGURATION REGISTER (CPUCFGR) ....322

15.5 DMMU CONFIGURATION REGISTER (DMMUCFGR) ....................................323
15.6 IMMU CONFIGURATION REGISTER (IMMUCFGR) ....................................325
15.7 DC CONFIGURATION REGISTER (DCCFGR) ....................................326
15.8 IC CONFIGURATION REGISTER (ICCFGR) ....................................327
15.9 DEBUG CONFIGURATION REGISTER (DCFGR) ....................................328
15.10 PERFORMANCE COUNTERS CONFIGURATION REGISTER (PCCFGR).........329
**16 APPLICATION BINARY INTERFACE ....................................330**
16.1 DATA REPRESENTATION....................................330
**16.1.1 Fundamental Types....................................330**
**16.1.2 Aggregates and Unions....................................331**
**16.1.3 Bit-fields....................................332**
16.2 FUNCTION CALLING SEQUENCE....................................333
**16.2.1 Register Usage ....................................333**
**16.2.2 The Stack Frame....................................335**
**16.2.3 Parameter Passing ....................................335**
**16.2.4 Functions Returning Scalars or No Value ....................................336**
**16.2.5 Functions Returning Structures or Unions ....................................336**
16.3 OPERATING SYSTEM INTERFACE....................................336
**16.3.1 Exception Interface....................................336**
**16.3.2 Virtual Address Space....................................337**
**16.3.3 Page Size ....................................337**
**16.3.4 Virtual Address Assignments ....................................337**
**16.3.5 Stack ....................................338**
**16.3.6 Processor Execution Modes ....................................338**
16.4 POSITION-INDEPENDENT CODE ....................................338
16.5 ELF....................................338
**16.5.1 Header Convention....................................338**
**16.5.2 Sections ....................................339**
**16.5.3 Relocation....................................339**
16.6 COFF ....................................340
**16.6.1 Sections ....................................340**
**16.6.2 Relocation....................................340**

# Table Of Figures

Figure 3-1. Register Indirect with Displacement Addressing ....................................................16
Figure 3-2. PC Relative Addressing ..........................................................................................17
Figure 5-1. Instruction Set..........................................................................................................31
Figure 8-1. Translation of Effective to Physical Address – Simplified block diagram for 32-bit processor implementations.....................................................................................................261
Figure 8-2. Memory Divided Into L1 and L2 pages ................................................................277
Figure 8-3. Address Translation Mechanism using Two-Level Page Table ...............................278
Figure 8-4. Address Translation Mechanism using only L1 Page Table ...................................279
Figure 8-5. Memory Divided Into L0, L1 and L2 pages ..........................................................280
Figure 8-6. Address Translation Mechanism using Three-Level Page Table............................281
Figure 8-7. Address Translation Mechanism using Two-Level Page Table ...............................282
Figure 8-8. Selection of Page Protection Attributes for Data Accesses ....................................284
Figure 8-9. Selection of Page Protection Attributes for Instruction Fetch Accesses ..................284
Figure 8-10. Page Table Entry Format .....................................................................................285
Figure 10-1. Block Diagram of Debug Support ........................................................................299
Figure 13-1. Programmable Interrupt Controller Block Diagram...............................................313
Figure 14-1. Tick Timer Block Diagram ...................................................................................316
Figure 16-1. Byte aligned, sizeof is 1 .......................................................................................331
Figure 16-2. No padding, sizeof is 8.........................................................................................331
Figure 16-3. Padding, sizeof is 18............................................................................................332
Figure 16-4. Storage unit sharingand alignment padding, sizeof is 12 .....................................333

# Table Of Tables

Table 1-1. Acronyms and Abbreviations ........ 9
Table 1-1. Authors of this Manual ........ 10
Table 1-2. Revision History ........ 12
Table 1-3. Conventions ........ 13
Table 3-1. Memory Operands and their sizes ........ 18
Table 3-2. Default Bit and Byte Ordering in Halfwords ........ 18
Table 3-3. Default Bit and Byte Ordering in Singlewords and Single Precision Floats ........ 18
Table 3-4. Default Bit and Byte Ordering in Doublewords, Double Precision Floats and all Vector Types ........ 19
Table 3-5. Memory Operand Alignment ........ 19
Table 4-1. Groups of SPRs ........ 21
Table 4-2. List of All Special-Purpose Registers ........ 24
Table 4-3. General-Purpose Registers ........ 24
Table 4-4. SR Field Descriptions ........ 27
Table 4-5. EPCR Field Descriptions ........ 27
Table 4-6. EEAR Field Descriptions ........ 28
Table 4-7. ESR Field Descriptions ........ 28
Table 4-8. FPCSR Field Descriptions ........ 30
Table 5-1. OpenRISC 1000 Instruction Classes ........ 32
Table 6-1. Exception Classes ........ 252
Table 6-2. Exception Types and Causal Conditions ........ 253
Table 6-3. Values of EPCR and EEAR After Exception ........ 255
Table 8-1. MMU Exceptions ........ 262
Table 8-2. List of MMU Special-Purpose Registers ........ 263
Table 8-3. DMMUCR Field Descriptions ........ 264
Table 8-4. DMMUPR Field Descriptions ........ 265
Table 8-5. IMMUCR Field Descriptions ........ 266
Table 8-6. IMMUPR Field Descriptions ........ 267
Table 8-7. xTLBEIR Field Descriptions ........ 267
Table 8-8. xTLBMR Field Descriptions ........ 269
Table 8-9. DTLBTR Field Descriptions ........ 271
Table 8-10. ITLBWyTR Field Descriptions ........ 272
Table 8-11. xATBMR Field Descriptions ........ 273
Table 8-12. DATBTR Field Descriptions ........ 275
Table 8-13. IATBTR Field Descriptions ........ 276
Table 8-14. Protection Attributes ........ 283
Table 8-15. PTE Field Descriptions ........ 285
Table 9-1. Cache Registers ........ 289
Table 9-2. DCCR Field Descriptions ........ 289
Table 9-3. ICCR Field Descriptions ........ 290

Table 9-4. DCBPR Field Descriptions ....................291
Table 9-5. DCBFR Field Descriptions ....................292
Table 9-6. DCBIR Field Descriptions....................292
Table 9-7. DCBWR Field Descriptions ....................293
Table 9-8. DCBLR Field Descriptions....................294
Table 9-9. ICBPR Field Descriptions....................294
Table 9-10. ICBIR Field Descriptions ....................295
Table 9-11. ICBLR Field Descriptions ....................295
Table 10-1. DVR Field Descriptions ....................299
Table 10-2. DCR Field Descriptions....................300
Table 10-3. DMR1 Field Descriptions....................302
Table 10-4. DMR2 Field Descriptions....................304
Table 10-5. DWCR Field Descriptions....................304
Table 10-6. DSR Field Descriptions ....................306
Table 10-7. DRR Field Descriptions....................307
Table 11-1. PCCR0 Field Descriptions....................309
Table 11-2. PCMR Field Descriptions ....................310
Table 12-1. PMR Field Descriptions....................312
Table 13-1. PICMR Field Descriptions ....................314
Table 13-2. PICSR Field Descriptions....................315
Table 14-1. TTMR Field Descriptions....................318
Table 14-2. TTCR Field Descriptions ....................319
Table 15-1. VR Field Descriptions....................321
Table 15-2. UPR Field Descriptions ....................322
Table 15-3. CPUCFGR Field Descriptions ....................323
Table 15-4. DMMUCFGR Field Descriptions....................324
Table 15-5. IMMUCFGR Field Descriptions ....................326
Table 15-6. DCCFGR Field Descriptions....................327
Table 15-7. ICCFGR Field Descriptions ....................328
Table 15-8. DCFGR Field Descriptions ....................329
Table 15-9. PCCFGR Field Descriptions....................329
Table 16-1. Scalar Types....................330
Table 16-2. Vector Types....................331
Table 16-3. Bit-Field Types and Ranges ....................332
Table 16-4. General-Purpose Registers ....................334
Table 16-5. Stack Frame ....................335
Table 16-6. Hardware Exceptions and Signals....................336
Table 16-7. Virtual Address Configuration....................338
Table 16-8. *e_ident* Field Values....................339
Table 16-9. *e_flags* Field Values....................339

# 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 | Supervison Register |
| TLB | Translation Lookaside Buffer |

**Table 1-1. 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 | damjanl@opencores.org | Initial document |
| Chen-Min Chen | jimmy@ee.nctu.edu.tw | Some notes |
| Marko Mlinar | markom@opencores.org | Fast context switches |
| Johan Rydberg | jrydberg@opencores.org | ELF section |
| Matan Ziv-Av | matan@svgalib.org | Several suggestions |
| Chris Ziomkowski | chris@opencores.org | Several suggestions |
| Greg McGary | greg@mcgary.org | l.cmov, trap exception |
| Bob Gardner | | Native Speaker Check |
| Rohit Mathur | rohitmathurs@opencores.org | Technical review and corrections |
| Maria Bolado | mbolado@teisa.unican.es | Technical review and corrections |

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

## 1.3 Revision History

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

| Revision Date | By | Modifications |
|---|---|---|
| 15/Mar/2000 | Damjan Lampret | Initial document |
| 7/Apr/2001 | Damjan Lampret | First public release |
| 22/Apr/2001 | Damjan Lampret | Incorporated changes from Johan and Matan |
| 16/May/2001 | Damjan Lampret | Changed SR, Debug, Exceptions, TT, PM. Added l.cmov, l.ff1, etc. |
| 23/May/2001 | Damjan Lampret | Added SR[SUMRA], configuration registerc etc. |
| 24/May/2001 | Damjan Lampret | Changed virtually almost all chapters in some way – major change is addition of configuration registers. |
| 28/May/2001 | Damjan Lampret | Changed addresses of some SPRs, removed group SPR group 11, added DCR[CT]=7. |
| 24/Jan/2002 | Marko Mlinar | Major check and update |
| 9/Apr/2002 | Marko Mlinar | PICPR register removed; l.sys convention added; mtspr/mfspr now use bitwise OR instead of sum |
| 28/July/2002 | Jeanne Wiegelmann | First overall review & layout adjustment |
| 20/Spetember/2002 | Rohit Mathur | Second overall review |
| 12/January/2003 | Damjan Lampret | Synchronization with or1ksim and OR1200 RTL. Not all chapters have been checked. |
| 26/January/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. |

| Revision Date | By | Modifications |
|---|---|---|
| 26/June/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). |
| 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. |
| 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. |

**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 OPENCORES CVS. See details about how to get it on www.opencores.org.

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

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

**Table 1-3. Conventions**

## 1.7 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 (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- 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
- 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.opencores.org 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.

Opencores.org encourages third parties to design and market their own implementations of the OpenRISC 1000 architecture and to participate in further development of the architecture.

# 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.

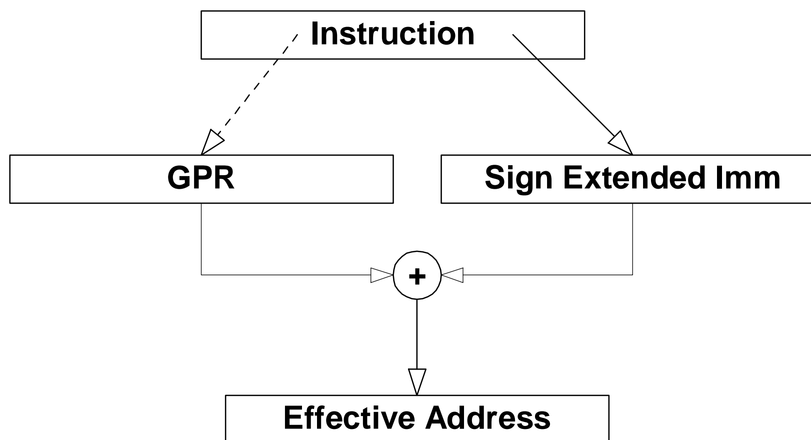



**Figure 3-1. Register Indirect with Displacement Addressing**

Figure 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.

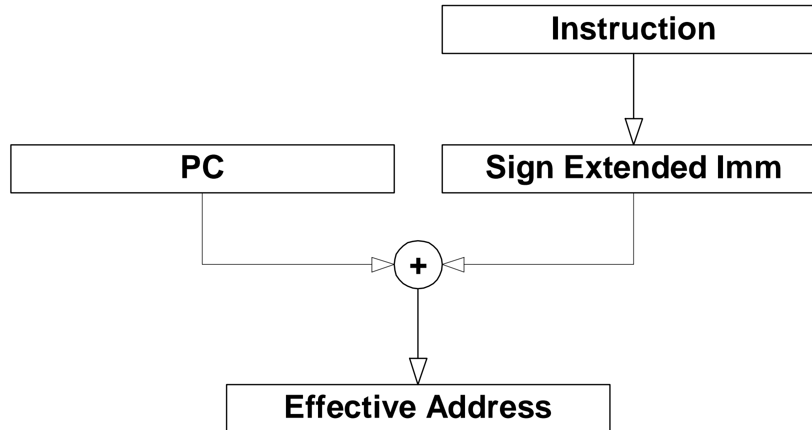



**Figure 3-2. PC Relative Addressing**

Figure 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.

Table 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**

Table 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**

Table 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 **Error! Reference source not found.** on page **Error! Bookmark not defined.** for information on alignment exception.

| 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.

| 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 |

| GROUP # | UNIT DESCRIPTION |
|---|---|
| | 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 |
| 0 | 8 | PCCFGR | – | R | Performance Counters Configuration register |
| 0 | 16 | NPC | – | R/W | PC mapped to SPR space (next PC) |
| 0 | 17 | SR | – | R/W | Supervision register |

| Grp # | Reg # | Reg Name | USER MODE | SUPV MODE | Description |
|---|---|---|---|---|---|
| 0 | 18 | PPC | – | R/W | PC mapped to SPR space (previous PC) |
| 0 | 20 | FPCSR | R* | R/W | FP Control Status register |
| 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 | 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 |
| 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- | ITLBW0MR0- | – | R/W | Instruction TLB Match registers |

| Grp # | Reg # | Reg Name | USER MODE | SUPV MODE | Description |
|---|---|---|---|---|---|
| | 639 | ITLBW0MR127 | | | 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 |
| 5 | 2 | MACHI | R/W | R/W | 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 |

| Grp # | Reg # | Reg Name | USER MODE | SUPV MODE | Description |
|---|---|---|---|---|---|
| 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.

## 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, floting-point data, vectors or memory pointers. Table 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 330 for information on floating-point data types.

| 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 | r8 | r7 | r6 |
| Register | R5 | r4 | r3 | r2 | r1 | r0 |

**Table 4-3. General-Purpose Registers**

R0 is used as a constant zero. Whether or not R0 is actually hardwired to zero is implementation dependent. **R0 should never be used as a destination register**. Functions of other registers are explained in chapter Application Binary Interfaceon page 330.

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.

## 4.6 Supervision Register (SR)

The Supervison 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.

| **Bit** | 31-28 | 27-17 | 16 |
|---|---|---|---|
| **Identifier** | CID | Reserved | SUMRA |
| **Reset** | 0 | 0 | 0 |
| **R/W** | R/W | Read Only | R/W |

| **Bit** | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
|---|---|---|---|---|---|---|---|---|
| **Identifier** | FO | EPH | DSX | OVE | OV | CY | F | CE |
| **Reset** | 1 | 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** | LEE | IME | DME | ICE | DCE | IEE | TEE | SM |
| **Reset** | 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 |

| | |
|---|---|
| SM | Supervisor Mode<br>0 Processor is in User Mode<br>1 Processor is in Supervisor Mode |
| TEE | Tick Timer Exception Enabled<br>0 Tick Timer Exceptions are not recognized<br>1 Tick Timer Exceptions are recognized |
| IEE | Interrupt Exception Enabled<br>0 Interrupts are not recognized |

| | |
|---|---|
| | 1 Interrupts are recognized |
| DCE | Data Cache Enable<br>0 Data Cache is not enabled<br>1 Data Cache is enabled |
| ICE | Instruction Cache Enable<br>0 Instruction Cache is not enabled<br>1 Instruction Cache is enabled |
| DME | Data MMU Enable<br>0 Data MMU is not enabled<br>1 Data MMU is enabled |
| IME | Instruction MMU Enable<br>0 Instruction MMU is not enabled<br>1 Instruction MMU is enabled |
| LEE | Little Endian Enable<br>0 Little Endian (LSB) byte ordering is not enabled<br>1 Little Endian (LSB) byte ordering is enabled |
| CE | CID Enable<br>0 CID disabled and shadow registers disabled<br>1 CID automatic increment and shadow registers enabled |
| F | Flag<br>0 Conditional branch flag was cleared by sfXX instructions<br>1 Conditional branch flag was set by sfXX instructions |
| CY | Carry flag<br>0 No carry out produced by last arithmetic operation<br>1 Carry out was produced by last arithmetic operation |
| OV | Overflow flag<br>0 No overflow occured during last arithmetic operation<br>1 Overflow occured during last arithmetic operation |
| OVE | Overflow flag Exception<br>0 Overflow flag does not cause an exception<br>1 Overflow flag causes range exception |
| DSX | Delay Slot Exception<br>0 EPCR points to instruction not in the delay slot<br>1 EPCR points to instruction in delay slot |
| EPH | Exception Prefix High<br>0 Exceptions vectors are located in memory area starting at 0x0<br>1 Exception vectors are located in memory area starting at 0xF0000000 |
| FO | Fixed One<br>This bit is always set |
| SUMRA | SPRs User Mode Read Access<br>0 All SPRs are inaccessible in user mode<br>1 Certain SPRs can be read in user mode |
| CID | Context ID *(optional)*<br>0-15 Current Processor Context |

**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, 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 |

| EEA | Exception SR |
|---|---|

**Table 4-7. ESR Field Descriptions**

## 4.10 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.

## 4.11 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<br>0 FP Exception is disabled<br>1 FP Exception is enabled |
| RM | Rounding Mode<br>0 Round to nearest<br>1 Round to zero<br>2 Round to infinity+<br>3 Round to infinity- |
| OVF | OVerflow Flag<br>0 No overflow<br>1 Result overflowed |
| UNF | UNderflow Flag<br>0 No underflow<br>1 Result underflowed |
| SNF | SNAN Flag<br>0 Result not SNAN<br>1 Result SNAN |
| QNF | QNAN Flag<br>0 Result not QNAN<br>1 Result QNAN |
| ZF | Zero Flag<br>0 Result not zero<br>1 Result zero |
| IXF | IneXact Flag<br>0 Result precise<br>1 Result inexact |
| IVF | InaValid Flag |

| | |
|---|---|
| | 0 Result valid<br>1 Result invalid |
| INF | INfinity Flag<br>0 Result finite<br>1 Result infinite |
| DZF | Divide by Zero Flag<br>0 Proper divide<br>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.

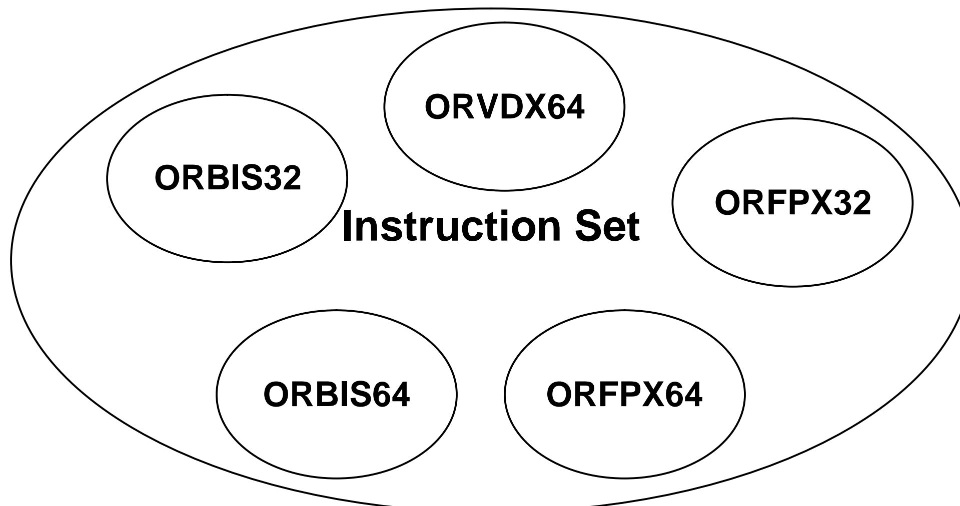



**Figure 5-1. Instruction Set**

## 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
- 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**

# l.add Add Signed l.add

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

## 5.3 ORBIS32/64

### Format:

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

### Description:

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

### 32-bit Implementation:

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

### 64-bit Implementation:

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

### Exceptions:

```
Range Exception
```

Instruction Class
ORBIS32 I

# l.addc Add Signed and Carry l.addc

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

## Format:

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

## Description:

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

## 32-bit Implementation:

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

## 64-bit Implementation:

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

## Exceptions:

```
Range Exception
```

Instruction Class
ORBIS32 I

# l.addi Add Immediate Signed l.addi

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . . . . . . . . . . . . 0 |
|---|---|---|---|
| opcode 0x27 | D | A | I |
| 6 bits | 5 bits | 5 bits | 16bits |

## Format:

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

## Description:

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

## 32-bit Implementation:

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

## 64-bit Implementation:

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

## Exceptions:

```
Range Exception
```

Instruction Class
ORBIS32 I

# l.addic Add Immediate Signed and Carry l.addic

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x28 | | | | | | D | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

### Format:

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

### Description:

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

### 32-bit Implementation:

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

### 64-bit Implementation:

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

### Exceptions:

```
Range Exception
```

Instruction Class
ORBIS32 I

# 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 bits | 2 bits | 4 bits | 4bits |

## 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] < - rA[31:0] AND rB[31:0]
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] AND rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x29 | | | | | | D | | | | | A | | | | | K | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## 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
```

Instruction Class
ORBIS32 I

# l.bf Branch if Flag l.bf

| 31 . . . . 26 | 25 . . . . . . . . . . . . . . . . . . . . . . . . 0 |
|---|---|
| opcode 0x4 | N |
| 6 bits | 26bits |

## Format:

```
l.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 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
```

Instruction Class
ORBIS32 I

# l.bnf Branch if No Flag l.bnf

| 31 | . | . | . | . | 26 | 25 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x3 | | | | | | N | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 26bits | | | | | | | | | | | | | | | | | | | | | | | | | | |

## Format:

```
l.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 with a delay of one instruction.

## 32-bit Implementation:

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

## 64-bit Implementation:

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

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

# 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 bits | 2 bits | 4 bits | 4bits |

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

## 64-bit Implementation:

```
rD[63:0] < - SR[F] ? rA[63:0] : rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 II

# l.csync Context Syncronization l.csync

| 31 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x23000000 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 32bits | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

### 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
```

Instruction Class

ORBIS32 II

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x1c | | | | | | reserved | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 26bits | | | | | | | | | | | | | | | | | | | | | | | | | | |

## Format:

```
l.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
```

Instruction Class
ORBIS32 II

# l.cust2 Reserved for ORBIS32/64 Custom Instructions l.cust2

<table>
<tr><td>31</td><td>.</td><td>.</td><td>.</td><td>.</td><td>26</td><td>25</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>0</td></tr>
<tr><td colspan="6">opcode 0x1d</td><td colspan="26">reserved</td></tr>
<tr><td colspan="6">6 bits</td><td colspan="26">26bits</td></tr>
</table>

### Format:

```
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.

### 32-bit Implementation:

```
N/A
```

### 64-bit Implementation:

```
N/A
```

### Exceptions:

```
N/A
```

Instruction Class
ORBIS32 II

# l.cust3 Reserved for ORBIS32/64 Custom Instructions l.cust3

| 31 | . | . | . | . | 26 | 25 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x1e | | | | | | reserved | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 26bits | | | | | | | | | | | | | | | | | | | | | | | | | | |

## 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
```

Instruction Class
ORBIS32 II

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x1f | | | | | | reserved | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 26bits | | | | | | | | | | | | | | | | | | | | | | | | | | |

### Format:

```
l.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
```

Instruction Class
ORBIS32 II

# l.cust5 Reserved for ORBIS32/64 Custom Instructions 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 | | | | | | 5bits | | | | |

## 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
```

Instruction Class
ORBIS32 II

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x3d | | | | | | reserved | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 26bits | | | | | | | | | | | | | | | | | | | | | | | | | |

### Format:

```
l.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
```

Instruction Class
ORBIS32 II

# l.cust7 Reserved for ORBIS32/64 Custom Instructions l.cust7

| 31 | . | . | . | . | 26 | 25 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x3e | | | | | | reserved | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 26bits | | | | | | | | | | | | | | | | | | | | | | | | | | | |

## Format:

```
l.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
```

Instruction Class
ORBIS32 II

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x3f | | | | | | reserved | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 26bits | | | | | | | | | | | | | | | | | | | | | | | | | |

## Format:

```
l.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
```

Instruction Class
ORBIS32 II

# 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 bits | 2 bits | 4 bits | 4bits |

## 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. A carry flag is set when the divisor is zero (if carry SR[CY] is implemented).

## 32-bit Implementation:

```
rD[31:0] < - rA[31:0] / rB[31:0]
SR[OV] < - overflow
SR[CY] < - carry
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] / rB[63:0]
SR[OV] < - overflow
SR[CY] < - carry
```

## Exceptions:

```
Range Exception
```

Instruction Class
ORBIS32 II

# l.divu Divide Unsigned l.divu

| 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 0xa | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 1 bits | 2 bits | | 4 bits | | | | 4bits | | | |

## 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. A carry flag is set when the divisor is zero (if carry SR[CY] is implemented).

## 32-bit Implementation:

```
rD[31:0] < - rA[31:0] / rB[31:0]
SR[OV] < - overflow
SR[CY] < - carry
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] / rB[63:0]
SR[OV] < - overflow
SR[CY] < - carry
```

## Exceptions:

```
Range Exception
```

Instruction Class
ORBIS32 II

# l.extbs Extend Byte with Sign l.extbs

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

## 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
```

Instruction Class
ORBIS32 II

# l.extbz Extend Byte with Zero l.extbz

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | 10 | 9 | . | . | 6 | 5 | 4 | 3 | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x38 | | | | | | D | | | | | A | | | | | reserved | | | | | | opcode 0x3 | | | | reserved | | opcode 0xc | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 6 bits | | | | | | 4 bits | | | | 2 bits | | 4bits | | | |

## 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

Instruction Class
ORBIS32 II

# l.exths Extend Half Word with Sign l.exths

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

## 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

Instruction Class
ORBIS32 II

# 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 | 4bits |

## 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
```

Instruction Class
ORBIS32 II

# 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 | 4bits |

## 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

Instruction Class
ORBIS64 II

# 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 | | 4bits | | | |

## 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
```

Instruction Class
ORBIS64 II

# l.ff1 Find First 1 l.ff1

| 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 0xf |
| 6 bits | 5 bits | 5 bits | 5 bits | 1 bits | 2 bits | 4 bits | 4bits |

## Format:

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

## Description:

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

## 32-bit Implementation:

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

## 64-bit Implementation:

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

## Exceptions:

None

Instruction Class
ORBIS32 II

# l.fl1 Find Last 1 l.fl1

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

## Format:

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

## Description:

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

## 32-bit Implementation:

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

## 64-bit Implementation:

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

## Exceptions:

None

Instruction Class
ORBIS32 II

# l.j Jump l.j

| 31 . . . . 26 | 25 . . . . . . . . . . . . . . . . . . . . . . . . 0 |
|---|---|
| opcode 0x0 | N |
| 6 bits | 26bits |

## Format:

```
l.j N
```

## Description:

The immediate value is shifted left two bits, sign-extended to program counter width, and then added to the address of the jump instruction. The result is the effective address of the jump. The program unconditionally jumps to EA with a delay of one instruction.

## 32-bit Implementation:

```
PC < - exts(Immediate < < 2) + JumpInsnAddr
```

## 64-bit Implementation:

```
PC < - exts(Immediate < < 2) + JumpInsnAddr
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

# l.jal Jump and Link l.jal

| 31 | . | . | . | . | 26 | 25 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x1 | | | | | | N | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 26bits | | | | | | | | | | | | | | | | | | | | | | | | | | |

## Format:

```
l.jal N
```

## Description:

The immediate value is shifted left two bits, sign-extended to program counter width, and then added to the address of the jump instruction. The result is the effective address of the jump. The program unconditionally jumps to EA with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register.

## 32-bit Implementation:

```
PC < - exts(Immediate < < 2) + JumpInsnAddr
LR < - DelayInsnAddr + 4
```

## 64-bit Implementation:

```
PC < - exts(Immediate < < 2) + JumpInsnAddr
LR < - DelayInsnAddr + 4
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

# l.jalr Jump and Link Register l.jalr

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

## Format:

```
l.jalr rB
```

## Description:

The contents of general-purpose register rB is the effective address of the jump. The program unconditionally jumps to EA with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register. It is not allowed to specify link register as rB.

## 32-bit Implementation:

```
PC < - rB
LR < - DelayInsnAddr + 4
```

## 64-bit Implementation:

```
PC < - rB
LR < - DelayInsnAddr + 4
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

# l.jr Jump Register l.jr

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

## Format:

```
l.jr rB
```

## Description:

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

## 32-bit Implementation:

```
PC < - rB
```

## 64-bit Implementation:

```
PC < - rB
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

# l.lbs Load Byte and Extend with Sign l.lbs

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x24 | | | | | | D | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## Format:

```
l.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
```

Instruction Class
ORBIS32 I

# l.lbz Load Byte and Extend with Zero l.lbz

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . . . . . . . . . . . . 0 |
|---|---|---|---|
| opcode 0x23 | D | A | I |
| 6 bits | 5 bits | 5 bits | 16bits |

## Format:

```
l.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
```

Instruction Class
ORBIS32 I

# l.ld Load Double Word l.ld

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x20 | | | | | | D | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## 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
```

Instruction Class

ORBIS64 I

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x26 | | | | | | D | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## 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
```

Instruction Class

ORBIS32 I

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x25 | | | | | | D | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## 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
```

Instruction Class
ORBIS32 I

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x22 | | | | | | D | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## 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
```

Instruction Class
ORBIS32 I

# l.lwz Load Single Word and Extend with Zero l.lwz

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

## Format:

```
l.lwz 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.

## 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] < - 0
```

## Exceptions:

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

Instruction Class

ORBIS32 I

# l.mac Multiply Signed and Accumulate 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 | | | | | | | 4bits | | | |

## 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 result is truncated to 32 bits and added to the special-purpose registers MACHI and MACLO. All operands are treated as signed integers.

## 32-bit Implementation:

```
temp[31:0] < - rA[31:0] * rB[31:0]
MACHI[31:0]MACLO[31:0] < - temp[31:0] +
MACHI[31:0]MACLO[31:0]
```

## 64-bit Implementation:

```
temp[31:0] < - rA[63:0] * rB[63:0]
MACHI[31:0]MACLO[31:0] < - temp[31:0] +
MACHI[31:0]MACLO[31:0]
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 II

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

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . . . . . . . . . 0 |
|---|---|---|---|---|
| opcode 0x13 | I | reserved | B | I |
| 6 bits | 5 bits | 5 bits | 5 bits | 11bits |

## Format:

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

## Description:

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

## 32-bit Implementation:

```
temp[31:0] < - rA[31:0] * exts(Immediate)
MACHI[31:0]MACLO[31:0] < - temp[31:0] +
MACHI[31:0]MACLO[31:0]
```

## 64-bit Implementation:

```
temp[31:0] < - rA[63:0] * exts(Immediate)
MACHI[31:0]MACLO[31:0] < - temp[31:0] +
MACHI[31:0]MACLO[31:0]
```

## Exceptions:

```
None
```

Instruction Class

ORBIS32 II

# l.macrc MAC Read and Clear l.macrc

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | 17 | 16 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x6 | | | | | | D | | | | | reserved | | | | opcode 0x10000 | | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 4 bits | | | | 17bits | | | | | | | | | | | | | | | | |

## Format:

```
l.macrc rD
```

## Description:

Once all instructions in MAC pipeline are completed, the contents of MAC is placed into general-purpose register rD and MAC accumulator is cleared.

## 32-bit Implementation:

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

## 64-bit Implementation:

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

## Exceptions:

None

Instruction Class
ORBIS32 II

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x2d | | | | | | D | | | | | A | | | | | K | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

### 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

Instruction Class
ORBIS32 I

# l.movhi Move Immediate High l.movhi

| 31 . . . . 26 | 25 . . . 21 | 20 . . 17 | 16 | 15 . . . . . . . . . . . . . . 0 |
|---|---|---|---|---|
| opcode 0x6 | D | reserved | opcode 0x0 | K |
| 6 bits | 5 bits | 4 bits | 1 bits | 16bits |

## 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] < - extz(Immediate) < < 16
```

## 64-bit Implementation:

```
rD[63:0] < - extz(Immediate) < < 16
```

## Exceptions:

None

Instruction Class
ORBIS32 I

# l.msb Multiply Signed and Subtract 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 | 4bits |

## 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 result is truncated to 32 bits and 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.

## 32-bit Implementation:

```
temp[31:0] < - rA[31:0] * rB[31:0]
MACHI[31:0]MACLO[31:0] < - MACHI[31:0]MACLO[31:0]
- temp[31:0]
```

## 64-bit Implementation:

```
temp[31:0] < - rA[63:0] * rB[63:0]
MACHI[31:0]MACLO[31:0] < - MACHI[31:0]MACLO[31:0]
- temp[31:0]
```

## Exceptions:

None

Instruction Class
ORBIS32 II

# l.msync Memory Syncronization l.msync

| 31 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x22000000 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 32bits | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

## Format:

```
l.msync
```

## Description:

Execution of the memory synchronization instruction results in completion of all load/store operations before the RISC core continues.

## 32-bit Implementation:

```
memory-synchronization
```

## 64-bit Implementation:

```
memory-synchronization
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 II

# 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 | | | | | 11bits | | | | | | | | | | |

## 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
```

Instruction Class
ORBIS32 I

# 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 bits | 2 bits | 4 bits | 4bits |

## Format:

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

## Description:

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

## 32-bit Implementation:

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

## 64-bit Implementation:

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

## Exceptions:

```
Range Exception
```

Instruction Class
ORBIS32 I

# l.muli Multiply Immediate Signed l.muli

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

## 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.

## 32-bit Implementation:

```
rD[31:0] < - rA[31:0] * Immediate
SR[OV] < - overflow
SR[CY] < - carry
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] * Immediate
SR[OV] < - overflow
SR[CY] < - carry
```

## Exceptions:

```
Range Exception
```

Instruction Class
ORBIS32 I

# 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 bits | 2 bits | | 4 bits | | | | 4bits | | | |

## Format:

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

## Description:

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

## 32-bit Implementation:

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

## 64-bit Implementation:

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

## Exceptions:

```
Range Exception
```

Instruction Class
ORBIS32 I

# l.nop No Operation l.nop

| 31 | . | . | . | . | . | . | 24 | 23 | . | . | . | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x15 | | | | | | | | reserved | | | | | | | | K | | | | | | | | | | | | | | | |
| 8 bits | | | | | | | | 8 bits | | | | | | | | 16bits | | | | | | | | | | | | | | | |

## 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

| 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 0x4 | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 1 bits | 2 bits | | 4 bits | | | | 4bits | | | |

## 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] <- rA[31:0] OR rB[31:0]
```

## 64-bit Implementation:

```
rD[63:0] <- rA[63:0] OR rB[63:0]
```

## Exceptions:

None

Instruction Class

ORBIS32 I

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x2a | | | | | | D | | | | | A | | | | | K | | | | | | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## 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] < - rA[31:0] OR extz(Immediate)
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] OR extz(Immediate)
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

# l.psync Pipeline Syncronization l.psync

| 31 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x22800000 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 32bits | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

### 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
```

Instruction Class
ORBIS32 II

# l.rfe Return From Exception l.rfe

| 31 . . . . 26 | 25 . . . . . . . . . . . . . . . . . . . . . . . . 0 |
|---|---|
| opcode 0x9 | reserved |
| 6 bits | 26bits |

## Format:

```
l.rfe
```

## Description:

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

## 32-bit Implementation:

```
PC < - EPCR
SR < - ESR
```

## 64-bit Implementation:

```
PC < - EPCR
SR < - ESR
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

# 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 bits | 4 bits | 2 bits | 4bits |

## 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. In 32-bit implementations bit 5 of rB is ignored.

## 32-bit Implementation:

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

## 64-bit Implementation:

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

## Exceptions:

None

Instruction Class
ORBIS32 II

# l.rori Rotate Right with Immediate l.rori

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . . . . 8 | 7 6 | 5 . . . . 0 |
|---|---|---|---|---|---|
| opcode 0x2e | D | A | reserved | opcode 0x3 | L |
| 6 bits | 5 bits | 5 bits | 8 bits | 2 bits | 6bits |

## 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
```

Instruction Class
ORBIS32 II

# l.sb Store Byte l.sb

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x36 | | | | | | I | | | | | A | | | | | B | | | | | I | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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
```

Instruction Class
ORBIS32 I

# l.sd Store Double Word l.sd

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x34 | | | | | | I | | | | | A | | | | | B | | | | | I | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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
```

Instruction Class
ORBIS64 I

# l.sfeq Set Flag if Equal l.sfeq

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x720 | | | | | | | | | | | A | | | | | B | | | | | reserved | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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] < - rA[31:0] == rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] == rB[63:0]
```

## Exceptions:

None

Instruction Class
ORBIS32 I

# l.sfeqi Set Flag if Equal Immediate l.sfeqi

| 31 . . . . . . . . . 21 | 20 . . . 16 | 15 . . . . . . . . . . . . . . 0 |
|---|---|---|
| opcode 0x5e0 | A | I |
| 11 bits | 5 bits | 16bits |

## Format:

```
l.sfeqi rA,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
```

Instruction Class
ORBIS32 II

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

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x72b | | | | | | | | | | | A | | | | | B | | | | | reserved | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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

Instruction Class
ORBIS32 I

# l.sfgesi Set Flag if Greater or Equal Than Immediate Signed l.sfgesi

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x5eb | | | | | | | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

### Format:

```
l.sfgesi 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 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
```

Instruction Class
ORBIS32 II

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

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x723 | | | | | | | | | | | A | | | | | B | | | | | reserved | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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] < - rA[31:0] >= rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] >= rB[63:0]
```

## Exceptions:

None

Instruction Class

ORBIS32 I

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

| 31 . . . . . . . . . 21 | 20 . . . 16 | 15 . . . . . . . . . . . . . . 0 |
|---|---|---|
| opcode 0x5e3 | A | I |
| 11 bits | 5 bits | 16bits |

### Format:

```
l.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

Instruction Class
ORBIS32 II

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

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x72a | | | | | | | | | | | A | | | | | B | | | | | reserved | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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] < - rA[31:0] > rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] > rB[63:0]
```

## Exceptions:

None

Instruction Class
ORBIS32 I

# l.sfgtsi Set Flag if Greater Than Immediate Signed l.sfgtsi

| 31 . . . . . . . . . 21 | 20 . . . 16 | 15 . . . . . . . . . . . . . . 0 |
|---|---|---|
| opcode 0x5ea | A | I |
| 11 bits | 5 bits | 16bits |

## Format:

```
l.sfgtsi 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 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

Instruction Class
ORBIS32 II

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

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x722 | | | | | | | | | | | A | | | | | B | | | | | reserved | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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] < - rA[31:0] > rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] > rB[63:0]
```

## Exceptions:

None

Instruction Class
ORBIS32 I

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

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x5e2 | | | | | | | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## Format:

```
l.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

Instruction Class
ORBIS32 II

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

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x72d | | | | | | | | | | | A | | | | | B | | | | | reserved | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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] <- rA[31:0] <= rB[31:0]
```

## 64-bit Implementation:

```
SR[F] <- rA[63:0] <= rB[63:0]
```

## Exceptions:

None

Instruction Class
ORBIS32 I

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

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x5ed | | | | | | | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## Format:

```
l.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

Instruction Class
ORBIS32 II

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

<table>
<tr><td>31</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>21</td><td>20</td><td>.</td><td>.</td><td>.</td><td>16</td><td>15</td><td>.</td><td>.</td><td>.</td><td>11</td><td>10</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>0</td></tr>
<tr><td colspan="11">opcode 0x725</td><td colspan="5">A</td><td colspan="5">B</td><td colspan="11">reserved</td></tr>
<tr><td colspan="11">11 bits</td><td colspan="5">5 bits</td><td colspan="5">5 bits</td><td colspan="11">11bits</td></tr>
</table>

## 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] < - rA[31:0] < = rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] < = rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

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

| 31 . . . . . . . . . 21 | 20 . . . 16 | 15 . . . . . . . . . . . . . . 0 |
|---|---|---|
| opcode 0x5e5 | A | I |
| 11 bits | 5 bits | 16bits |

## Format:

```
l.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
```

Instruction Class
ORBIS32 II

# l.sflts Set Flag if Less Than Signed l.sflts

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x72c | | | | | | | | | | | A | | | | | B | | | | | reserved | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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] < - rA[31:0] < rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] < rB[63:0]
```

## Exceptions:

None

Instruction Class
ORBIS32 I

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

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x5ec | | | | | | | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## Format:

```
l.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
```

Instruction Class

ORBIS32 II

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

| 31 . . . . . . . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . . . . . . . . . 0 |
|---|---|---|---|
| opcode 0x724 | A | B | reserved |
| 11 bits | 5 bits | 5 bits | 11bits |

### Format:

```
l.sfltu 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] < - rA[31:0] < rB[31:0]
```

### 64-bit Implementation:

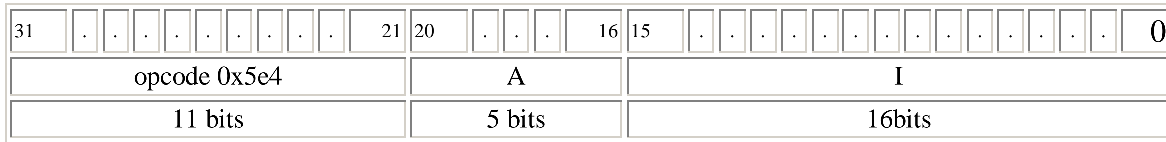
```
SR[F] < - rA[63:0] < rB[63:0]
```

### Exceptions:

None

Instruction Class
ORBIS32 I

# l.sfltui Set Flag if Less Than Immediate Unsigned l.sfltui



| 31 . . . . . . . . . 21 | 20 . . . 16 | 15 . . . . . . . . . . . . . . 0 |
|---|---|---|
| opcode 0x5e4 | A | I |
| 11 bits | 5 bits | 16bits |

## Format:

```
l.sfltui 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 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
```

Instruction Class
ORBIS32 II

# l.sfne Set Flag if Not Equal l.sfne

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x721 | | | | | | | | | | | A | | | | | B | | | | | reserved | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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] < - rA[31:0] != rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] != rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

# l.sfnei Set Flag if Not Equal Immediate l.sfnei

| 31 | . | . | . | . | . | . | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x5e1 | | | | | | | | | | | A | | | | | I | | | | | | | | | | | | | | | |
| 11 bits | | | | | | | | | | | 5 bits | | | | | 16bits | | | | | | | | | | | | | | | |

## Format:

```
l.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
```

Instruction Class
ORBIS32 II

# l.sh Store Half Word l.sh

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x37 | | | | | | I | | | | | A | | | | | B | | | | | I | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

## 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
```

Instruction Class
ORBIS32 I

# l.sll Shift Left Logical l.sll

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 | 9 . . 6 | 5 4 | 3 . . 0 |
|---|---|---|---|---|---|---|---|
| opcode 0x38 | D | A | B | reserved | opcode 0x0 | reserved | opcode 0x8 |
| 6 bits | 5 bits | 5 bits | 5 bits | 1 bits | 4 bits | 2 bits | 4bits |

## 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

Instruction Class
ORBIS32 I

# 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 | 6bits |

## 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

Instruction Class
ORBIS32 I

# l.sra Shift Right Arithmetic l.sra

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | 9 | . | . | 6 | 5 | 4 | 3 | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x38 | | | | | | D | | | | | A | | | | | B | | | | | reserved | opcode 0x2 | | | | reserved | | opcode 0x8 | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 1 bits | 4 bits | | | | 2 bits | | 4bits | | | |

## 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
```

Instruction Class
ORBIS32 I

# 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 | 6bits |

## 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

Instruction Class
ORBIS32 I

# l.srl Shift Right Logical l.srl

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | 9 | . | . | 6 | 5 | 4 | 3 | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x38 | | | | | | D | | | | | A | | | | | B | | | | | reserved | opcode 0x1 | | | | reserved | | opcode 0x8 | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 1 bits | 4 bits | | | | 2 bits | | 4bits | | | |

## 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
```

Instruction Class
ORBIS32 I

# 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 | 6bits |

## 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

Instruction Class
ORBIS32 I

# l.sub Subtract Signed 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 bits | 2 bits | 4 bits | 4bits |

## 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. This isntruction does not change carry SR[CY] flag.

## 32-bit Implementation:

```
rD[31:0] <- rA[31:0] - rB[31:0]
SR[CY] <- carry
SR[OV] <- overflow
```

## 64-bit Implementation:

```
rD[63:0] <- rA[63:0] - rB[63:0]
SR[CY] <- carry
SR[OV] <- overflow
```

## Exceptions:

```
Range Exception
```

Instruction Class
ORBIS32 I

# l.sw Store Single Word l.sw

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x35 | | | | | | I | | | | | A | | | | | B | | | | | I | | | | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 11bits | | | | | | | | | | |

### 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
```

Instruction Class
ORBIS32 I

# l.sys System Call l.sys

| 31 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x2000 | | | | | | | | | | | | | | | | K | | | | | | | | | | | | | | | |
| 16 bits | | | | | | | | | | | | | | | | 16bits | | | | | | | | | | | | | | | |

## Format:

```
l.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.

## 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

| 31 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 16 | 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x2100 | | | | | | | | | | | | | | | | K | | | | | | | | | | | | | | | |
| 16 bits | | | | | | | | | | | | | | | | 16bits | | | | | | | | | | | | | | | |

## Format:

```
l.trap K
```

## Description:

Execution of trap instruction results in the trap exception if specified bit in SR is set. 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:

```
if SR[K] = 1 then trap-exception()
```

## 64-bit Implementation:

```
if SR[K] = 1 then trap-exception()
```

## Exceptions:

```
Trap exception
```

Instruction Class
ORBIS32 II

# l.xor Exclusive Or l.xor

| 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 0x5 |
| 6 bits | 5 bits | 5 bits | 5 bits | 1 bits | 2 bits | 4 bits | 4bits |

## 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] < - rA[31:0] XOR rB[31:0]
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] XOR rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORBIS32 I

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

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . . . . . . . . . . . . 0 |
|---|---|---|---|
| opcode 0x2b | D | A | I |
| 6 bits | 5 bits | 5 bits | 16bits |

## 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
```

Instruction Class
ORBIS32 I

# 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 | | | | | A | | | | | B | | | | | reserved | | | opcode 0x10 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## Format:

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

## Description:

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

## 32-bit Implementation:

```
N/A
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] + rB[63:0]
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX64 I

# 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 | 8bits |

## 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.

## 32-bit Implementation:

```
rD[31:0] < - rA[31:0] + rB[31:0]
```

## 64-bit Implementation:

```
rD[31:0] < - rA[31:0] + rB[31:0]
rD[63:32] < - 0
```

## Exceptions:

```
Floating Point
```

Instruction Class

ORFPX32 I

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

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

## 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
```

Instruction Class
ORFPX64 II

# 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 | 4bits |

### 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
```

Instruction Class
ORFPX32 II

# 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 | | | | | A | | | | | B | | | | | reserved | | | opcode 0x13 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## Format:

```
lf.div.d 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.

## 32-bit Implementation:

```
N/A
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] / rB[63:0]
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX64 II

# 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 | | | 8bits | | | | | | | |

## 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.

## 32-bit Implementation:

```
rD[31:0] < - rA[31:0] / rB[31:0]
```

## 64-bit Implementation:

```
rD[31:0] < - rA[31:0] / rB[31:0]
rD[63:32] < - 0
```

## Exceptions:

```
Floating Point
```

Instruction Class

ORFPX32 II

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

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0x32 | D | A | opcode 0x0 | reserved | opcode 0x15 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## Format:

```
lf.ftoi.d rD,rA
```

## Description:

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

## 32-bit Implementation:

```
N/A
```

## 64-bit Implementation:

```
rD[63:0] < - ftoi(rA[63:0])
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX64 I

# 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 | | | 8bits | | | | | | | |

## 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.

## 32-bit Implementation:

```
rD[31:0] < - ftoi(rA[31:0])
```

## 64-bit Implementation:

```
rD[31:0] < - ftoi(rA[31:0])
rD[63:32] < - 0
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX32 I

# 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 | A | opcode 0x0 | reserved | opcode 0x14 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## Format:

```
lf.itof.d rD,rA
```

## Description:

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:

```
N/A
```

## 64-bit Implementation:

```
rD[63:0] < - itof(rA[63:0])
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX64 I

# 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 | 8bits |

## 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.

## 32-bit Implementation:

```
rD[31:0] < - itof(rA[31:0])
```

## 64-bit Implementation:

```
rD[31:0] < - itof(rA[31:0])
rD[63:32] < - 0
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX32 I

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

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

## Format:

```
lf.madd.d rD,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.

## 32-bit Implementation:

```
N/A
```

## 64-bit Implementation:

```
FPMADDHI[31:0]FPMADDLO[31:0] < - rA[63:0] *
rB[63:0] + FPMADDHI[31:0]FPMADDLO[31:0]
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX64 II

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

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

## Format:

```
lf.madd.s rD,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.

## 32-bit Implementation:

```
FPMADDHI[31:0]FPMADDLO[31:0] < - rA[31:0] *
rB[31:0] + FPMADDHI[31:0]FPMADDLO[31:0]
```

## 64-bit Implementation:

```
FPMADDHI[31:0]FPMADDLO[31:0] < - rA[31:0] *
rB[31:0] + FPMADDHI[31:0]FPMADDLO[31:0]
FPMADDHI < - 0
FPMADDLO < - 0
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX32 II

# 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 | A | B | reserved | opcode 0x12 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## Format:

```
lf.mul.d 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.

## 32-bit Implementation:

```
N/A
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] * rB[63:0]
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX64 I

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

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0x32 | D | A | B | reserved | opcode 0x2 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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.

## 32-bit Implementation:

```
rD[31:0] < - rA[31:0] * rB[31:0]
```

## 64-bit Implementation:

```
rD[31:0] < - rA[31:0] * rB[31:0]
rD[63:32] < - 0
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX32 I

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

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0x32 | D | A | B | reserved | opcode 0x16 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## Format:

```
lf.rem.d 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.

## 32-bit Implementation:

```
N/A
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] % rB[63:0]
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX64 II

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

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0x32 | D | A | B | reserved | opcode 0x6 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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.

## 32-bit Implementation:

```
rD[31:0] < - rA[31:0] % rB[31:0]
```

## 64-bit Implementation:

```
rD[31:0] < - rA[31:0] % rB[31:0]
rD[63:32] < - 0
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX32 II

# 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 | | | | | B | | | | | reserved | | | opcode 0x18 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## Format:

```
lf.sfeq.d 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:

```
N/A
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] == rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class

ORFPX64 I

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

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x32 | | | | | | reserved | | | | | A | | | | | B | | | | | reserved | | | opcode 0x8 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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] < - rA[31:0] == rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[31:0] == rB[31:0]
```

## Exceptions:

None

Instruction Class
ORFPX32 I

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

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

## Format:

```
lf.sfge.d 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:

```
N/A
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] >= rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORFPX64 I

# lf.sfge.s Set Flag if Greater or Equal Than Floating-Point Single-Precision lf.sfge.s

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

## 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] < - rA[31:0] >= rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[31:0] >= rB[31:0]
```

## Exceptions:

None

Instruction Class
ORFPX32 I

# 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 | B | reserved | opcode 0x1a |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## Format:

```
lf.sfgt.d 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:

```
N/A
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] > rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORFPX64 I

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

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

## 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] < - rA[31:0] > rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[31:0] > rB[31:0]
```

## Exceptions:

None

Instruction Class
ORFPX32 I

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

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

## Format:

```
lf.sfle.d 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:

N/A

## 64-bit Implementation:

```
SR[F] < - rA[363:0] < = rB[63:0]
```

## Exceptions:

None

Instruction Class
ORFPX64 I

# 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 | 8bits |

## 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] < - rA[31:0] < = rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[31:0] < = rB[31:0]
```

## Exceptions:

None

Instruction Class
ORFPX32 I

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

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

## Format:

```
lf.sflt.d 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:

```
N/A
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] < rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORFPX64 I

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

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

## 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] < - rA[31:0] < rB[31:0]
```

## 64-bit Implementation:

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

## Exceptions:

None

Instruction Class
ORFPX32 I

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

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

## Format:

```
lf.sfne.d 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:

```
N/A
```

## 64-bit Implementation:

```
SR[F] < - rA[63:0] != rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORFPX64 I

# 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 | 8bits |

## 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] < - rA[31:0] != rB[31:0]
```

## 64-bit Implementation:

```
SR[F] < - rA[31:0] != rB[31:0]
```

## Exceptions:

```
None
```

Instruction Class
ORFPX32 I

# 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 | A | B | reserved | opcode 0x11 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## Format:

```
lf.sub.d 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.

## 32-bit Implementation:

```
N/A
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] - rB[63:0]
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX64 I

# 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 | 8bits |

## 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.

## 32-bit Implementation:

```
rD[31:0] <- rA[31:0] - rB[31:0]
```

## 64-bit Implementation:

```
rD[31:0] <- rA[31:0] - rB[31:0]
rD[63:32] <- 0
```

## Exceptions:

```
Floating Point
```

Instruction Class
ORFPX32 I

# lv.add.b Vector Byte Elements Add Signed lv.add.b

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x30 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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
```

Instruction Class
ORVDX64 I

# lv.add.h Vector Half-Word Elements Add Signed lv.add.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x31 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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
```

Instruction Class
ORVDX64 I

# lv.adds.b Vector Byte Elements Add Signed Saturated lv.adds.b

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

### 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
```

Instruction Class
ORVDX64 I

# 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 | | | 8bits | | | | | | | |

## 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

Instruction Class
ORVDX64 I

# lv.addu.b Vector Byte Elements Add Unsigned lv.addu.b

| 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 | | | 8bits | | | | | | | |

## 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
```

Instruction Class
ORVDX64 I

# lv.addu.h Vector Half-Word Elements Add Unsigned lv.addu.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x35 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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
```

Instruction Class
ORVDX64 I

# lv.addus.b Vector Byte Elements Add Unsigned Saturated lv.addus.b

| 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 | 8bits |

### 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
```

Instruction Class
ORVDX64 I

# 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 | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.all_eq.b Vector Byte Elements All Equal lv.all_eq.b

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x10 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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
```

Instruction Class

ORVDX64 I

# lv.all_eq.h Vector Half-Word Elements All Equal lv.all_eq.h

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x11 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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
```

Instruction Class
ORVDX64 I

# lv.all_ge.b Vector Byte Elements All Greater Than or Equal To lv.all_ge.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x12 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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 | 8bits |

## 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 | 8bits |

## 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 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x15 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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
```

Instruction Class
ORVDX64 I

# 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 | | | 8bits | | | | | | | |

## 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 | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.all_lt.b Vector Byte Elements All Less Than lv.all_lt.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x18 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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
```

Instruction Class
ORVDX64 I

# lv.all_lt.h Vector Half-Word Elements All Less Than lv.all_lt.h

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x19 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

### 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 | 8bits |

### 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 lv.all_ne.h

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x1b | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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
```

Instruction Class
ORVDX64 I

# lv.and Vector And lv.and

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x38 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## Format:

```
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] < - rA[63:0] AND rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# 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 | 8bits |

### 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
```

Instruction Class
ORVDX64 I

# lv.any_eq.h Vector Half-Word Elements Any Equal lv.any_eq.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x21 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x22 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# 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 | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# 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 | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.any_gt.h Vector Half-Word Elements Any Greater Than lv.any_gt.h

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x25 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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
```

Instruction Class
ORVDX64 I

# 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 | | | 8bits | | | | | | | |

## 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
```

Instruction Class

ORVDX64 I

# lv.any_le.h Vector Half-Word Elements Any Less Than or Equal To lv.any_le.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x27 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# 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 | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.any_lt.h Vector Half-Word Elements Any Less Than lv.any_lt.h

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x29 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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
```

Instruction Class
ORVDX64 I

# 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 | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.any_ne.h Vector Half-Word Elements Any Not Equal lv.any_ne.h

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

## 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 | | | 8bits | | |

## 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
ORVDX64 I

# lv.avg.h Vector Half-Word Elements Average lv.avg.h

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x3a | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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
```

Instruction Class
ORVDX64 I

# lv.cmp_eq.b Vector Byte Elements Compare Equal lv.cmp_eq.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x40 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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 |
|---|
| ORVDX64 I |

# lv.cmp_eq.h Vector Half-Word Elements Compare Equal lv.cmp_eq.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x41 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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[7:0] == rB[7:0]
rD[31:16] < - repl(rA[23:16] == rB[23:16]
rD[47:32] < - repl(rA[39:32] == rB[39:32]
rD[63:48] < - repl(rA[55:48] == rB[55:48]
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# lv.cmp_ge.b Vector Byte Elements Compare Greater Than or Equal To lv.cmp_ge.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x42 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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

# lv.cmp_ge.h Vector Half-Word Elements Compare Greater Than or Equal To lv.cmp_ge.h

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x43 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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[7:0] >= rB[7:0]
rD[31:16] < - repl(rA[23:16] >= rB[23:16]
rD[47:32] < - repl(rA[39:32] >= rB[39:32]
rD[63:48] < - repl(rA[55:48] >= rB[55:48]
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 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 | 8bits |

## 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
```

Instruction Class

ORVDX64 I

# lv.cmp_gt.h Vector Half-Word Elements Compare Greater Than lv.cmp_gt.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x45 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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[7:0] > rB[7:0]
rD[31:16] < - repl(rA[23:16] > rB[23:16]
rD[47:32] < - repl(rA[39:32] > rB[39:32]
rD[63:48] < - repl(rA[55:48] > rB[55:48]
```

### Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# 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 | 8bits |

## 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
```

Instruction Class

ORVDX64 I

# Vector Half-Word Elements
# lv.cmp_le.h Compare Less Than or Equal To lv.cmp_le.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x47 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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[7:0] < = rB[7:0]
rD[31:16] < - repl(rA[23:16] < = rB[23:16]
rD[47:32] < - repl(rA[39:32] < = rB[39:32]
rD[63:48] < - repl(rA[55:48] < = rB[55:48]
```

### Exceptions:

```
None
```

Instruction Class
ORVDX64 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 | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.cmp_lt.h Vector Half-Word Elements Compare Less Than lv.cmp_lt.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x49 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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[7:0] < = rB[7:0]
rD[31:16] < - repl(rA[23:16] < = rB[23:16]
rD[47:32] < - repl(rA[39:32] < = rB[39:32]
rD[63:48] < - repl(rA[55:48] < = rB[55:48]
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# lv.cmp_ne.b Vector Byte Elements Compare Not Equal lv.cmp_ne.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x4a |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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
ORVDX64 I

# lv.cmp_ne.h Vector Half-Word Elements Compare Not Equal lv.cmp_ne.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x4b |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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[7:0] != rB[7:0])
rD[31:16] < - repl(rA[23:16] != rB[23:16])
rD[47:32] < - repl(rA[39:32] != rB[39:32])
rD[63:48] < - repl(rA[55:48] != rB[55:48])
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# lv.cust1 Reserved for Custom Vector Instructions lv.cust1

| 31 . . . . 26 | 25 . . . . . . . . . . . . . . . . . 8 | 7 . . 4 | 3 . . 0 |
|---|---|---|---|
| opcode 0xa | reserved | opcode 0xc | reserved |
| 6 bits | 18 bits | 4 bits | 4bits |

## 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
ORVDX64 II

# lv.cust2 Reserved for Custom Vector Instructions lv.cust2

| 31 . . . . 26 | 25 . . . . . . . . . . . . . . . . 8 | 7 . . 4 | 3 . . 0 |
|---|---|---|---|
| opcode 0xa | reserved | opcode 0xd | reserved |
| 6 bits | 18 bits | 4 bits | 4bits |

## 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
```

Instruction Class
ORVDX64 II

# lv.cust3 Reserved for Custom Vector Instructions lv.cust3

| 31 . . . . 26 | 25 . . . . . . . . . . . . . . . . 8 | 7 . . 4 | 3 . . 0 |
|---|---|---|---|
| opcode 0xa | reserved | opcode 0xe | reserved |
| 6 bits | 18 bits | 4 bits | 4bits |

## 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
ORVDX64 II

# lv.cust4 Reserved for Custom Vector Instructions lv.cust4

| 31 . . . . 26 | 25 . . . . . . . . . . . . . . . . 8 | 7 . . 4 | 3 . . 0 |
|---|---|---|---|
| opcode 0xa | reserved | opcode 0xf | reserved |
| 6 bits | 18 bits | 4 bits | 4bits |

## 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

ORVDX64 II

# lv.madds.h Vector Half-Word Elements Multiply Add Signed Saturated lv.madds.h

| 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 | 8bits |

## 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.

## 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

| 31 | . . . . | 26 | 25 | . . . | 21 | 20 | . . . | 16 | 15 | . . . | 11 | 10 | . | 8 | 7 | . . . . . . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | D | | | A | | | B | | | reserved | | | opcode 0x55 | | |
| 6 bits | | | 5 bits | | | 5 bits | | | 5 bits | | | 3 bits | | | 8bits | | |

## 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] :
vrfB[7:0]
rD[15:8] < - rA[15:8] > rB[15:8] ? rA[15:8] :
vrfB[15:8]
rD[23:16] < - rA[23:16] > rB[23:16] ? rA[23:16] :
vrfB[23:16]
rD[31:24] < - rA[31:24] > rB[31:24] ? rA[31:24] :
vrfB[31:24]
rD[39:32] < - rA[39:32] > rB[39:32] ? rA[39:32] :
vrfB[39:32]
rD[47:40] < - rA[47:40] > rB[47:40] ? rA[47:40] :
vrfB[47:40]
rD[55:48] < - rA[55:48] > rB[55:48] ? rA[55:48] :
vrfB[55:48]
rD[63:56] < - rA[63:56] > rB[63:56] ? rA[63:56] :
vrfB[63:56]
```

## Exceptions:

Instruction Class

ORVDX64 I

# lv.max.b Vector Byte Elements Maximum lv.max.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x55 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

None

Instruction Class

ORVDX64 I

# lv.max.h Vector Half-Word Elements Maximum lv.max.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x56 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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] :
vrfB[15:0]
rD[31:16] < - rA[31:16] > rB[31:16] ? rA[31:16] :
vrfB[31:16]
rD[47:32] < - rA[47:32] > rB[47:32] ? rA[47:32] :
vrfB[47:32]
rD[63:48] < - rA[63:48] > rB[63:48] ? rA[63:48] :
vrfB[63:48]
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# lv.merge.b Vector Byte Elements Merge lv.merge.b

| 31 | . . . . | 26 | 25 | . . . | 21 | 20 | . . . | 16 | 15 | . . . | 11 | 10 | . | 8 | 7 | . . . . . . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | D | | | A | | | B | | | reserved | | | opcode 0x57 | | |
| 6 bits | | | 5 bits | | | 5 bits | | | 5 bits | | | 3 bits | | | 8bits | | |

## 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

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x58 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# 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 | | | 8bits | | |

**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] :
vrfB[7:0]
rD[15:8] < - rA[15:8] < rB[15:8] ? rA[15:8] :
vrfB[15:8]
rD[23:16] < - rA[23:16] < rB[23:16] ? rA[23:16] :
vrfB[23:16]
rD[31:24] < - rA[31:24] < rB[31:24] ? rA[31:24] :
vrfB[31:24]
rD[39:32] < - rA[39:32] < rB[39:32] ? rA[39:32] :
vrfB[39:32]
rD[47:40] < - rA[47:40] < rB[47:40] ? rA[47:40] :
vrfB[47:40]
rD[55:48] < - rA[55:48] < rB[55:48] ? rA[55:48] :
vrfB[55:48]
rD[63:56] < - rA[63:56] < rB[63:56] ? rA[63:56] :
vrfB[63:56]
```

**Exceptions:**

Instruction Class
ORVDX64 I

# 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 | | | 8bits | | | | | | | |

`None`

Instruction Class
ORVDX64 I

# lv.min.h Vector Half-Word Elements Minimum lv.min.h

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x5a | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

### 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] :
vrfB[15:0]
rD[31:16] < - rA[31:16] < rB[31:16] ? rA[31:16] :
vrfB[31:16]
rD[47:32] < - rA[47:32] < rB[47:32] ? rA[47:32] :
vrfB[47:32]
rD[63:48] < - rA[63:48] < rB[63:48] ? rA[63:48] :
vrfB[63:48]
```

### Exceptions:

```
None
```

Instruction Class

ORVDX64 I

# lv.msubs.h Vector Half-Word Elements Multiply Subtract Signed Saturated lv.msubs.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x5b |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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.

## 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

ORVDX64 I

# lv.muls.h Vector Half-Word Elements Multiply Signed Saturated lv.muls.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x5c |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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] < - sat32s(rA[15:0] * rB[15:0])
rD[31:16] < - sat32s(rA[31:16] * rB[31:16])
rD[47:32] < - sat32s(rA[47:32] * rB[47:32])
rD[63:48] < - sat32s(rA[63:48] * rB[63:48])
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 II

# 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 | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.nor Vector Not Or lv.nor

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x5e | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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] < - rA[63:0] NOR rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# lv.or Vector Or lv.or

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x5f |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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] < - rA[63:0] OR rB[63:0]
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# lv.pack.b Vector Byte Elements Pack lv.pack.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x60 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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.

## 32-bit Implementation:

```
N/A
```

## 64-bit Implementation:

```
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]
```

Instruction Class
ORVDX64 I

# lv.pack.b Vector Byte Elements Pack lv.pack.b

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x60 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

**Exceptions:**

None

Instruction Class
ORVDX64 I

# lv.pack.h Vector Half-word Elements Pack lv.pack.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x61 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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[15:0]
rD[15:8] < - rB[31:16]
rD[23:16] < - rB[47:32]
rD[31:24] < - rB[63:48]
rD[39:32] < - rA[15:0]
rD[47:40] < - rA[31:16]
rD[55:48] < - rA[47:32]
rD[63:56] < - rA[63:48]
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# lv.packs.b Vector Byte Elements Pack Signed Saturated lv.packs.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x62 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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.

## 32-bit Implementation:

```
N/A
```

## 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]
```

Instruction Class

ORVDX64 I

# lv.packs.b Vector Byte Elements Pack Signed Saturated lv.packs.b

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x62 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

```
rD[59:56] < - sat4s(rA[55:48]
rD[63:60] < - sat4s(rA[63:56]
```

### Exceptions:

None

Instruction Class

ORVDX64 I

# lv.packs.h Vector Half-word Elements Pack Signed Saturated lv.packs.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x63 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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
```

Instruction Class
ORVDX64 I

# lv.packus.b Vector Byte Elements Pack Unsigned Saturated lv.packus.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x64 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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.

## 32-bit Implementation:

```
N/A
```

## 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]
```

Instruction Class
ORVDX64 I

# lv.packus.b Vector Byte Elements Pack Unsigned Saturated lv.packus.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x64 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

```
rD[59:56] < - sat4u(rA[55:48]
rD[63:60] < - sat4u(rA[63:56]
```

**Exceptions:**

None

Instruction Class
ORVDX64 I

# lv.packus.h — Vector Half-word Elements Pack Unsigned Saturated — lv.packus.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x65 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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
ORVDX64 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 | 8bits |

## 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.

## 32-bit Implementation:

```
N/A
```

## 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:

Instruction Class

ORVDX64 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 | 8bits |

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 | 8bits |

## 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

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x68 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.sll Vector Shift Left Logical lv.sll

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x6b |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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.

## 32-bit Implementation:

```
N/A
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] < < rB[2:0]
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# lv.sll.b Vector Byte Elements Shift Left Logical lv.sll.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x69 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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 lv.sll.h

| 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 | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.sra.b Vector Byte Elements Shift Right Arithmetic lv.sra.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x6e |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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
ORVDX64 I

# lv.sra.h Vector Half-Word Elements Shift Right Arithmetic lv.sra.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x6f |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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

Instruction Class
ORVDX64 I

# 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 | | | 8bits | | | | | | | |

## 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.

## 32-bit Implementation:

```
N/A
```

## 64-bit Implementation:

```
rD[63:0] < - rA[63:0] >> rB[2:0]
```

## Exceptions:

```
None
```

Instruction Class
ORVDX64 I

# lv.srl.b Vector Byte Elements Shift Right Logical lv.srl.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x6c |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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 lv.srl.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x6d |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.sub.b Vector Byte Elements Subtract Signed lv.sub.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x71 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x72 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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
```

Instruction Class
ORVDX64 I

# lv.subs.b Vector Byte Elements Subtract Signed Saturated lv.subs.b

| 31 | . | . | . | . | 26 | 25 | . | . | . | 21 | 20 | . | . | . | 16 | 15 | . | . | . | 11 | 10 | . | 8 | 7 | . | . | . | . | . | . | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0xa | | | | | | D | | | | | A | | | | | B | | | | | reserved | | | opcode 0x73 | | | | | | | |
| 6 bits | | | | | | 5 bits | | | | | 5 bits | | | | | 5 bits | | | | | 3 bits | | | 8bits | | | | | | | |

## 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
```

Instruction Class
ORVDX64 I

# lv.subs.h Vector Half-Word Elements Subtract Signed Saturated lv.subs.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x74 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.subu.b Vector Byte Elements Subtract Unsigned lv.subu.b

| 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 | | | 8bits | | | | | | | |

## 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
ORVDX64 I

# lv.subu.h Vector Half-Word Elements Subtract Unsigned lv.subu.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x76 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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
ORVDX64 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 | | | 8bits | | | | | | | |

## 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
ORVDX64 I

# lv.subus.h Vector Half-Word Elements Subtract Unsigned Saturated lv.subus.h

| 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 | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.unpack.b Vector Byte Elements Unpack lv.unpack.b

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x79 |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

### 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
```

Instruction Class
ORVDX64 I

# lv.unpack.h Vector Half-Word Elements Unpack lv.unpack.h

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x7a |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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
```

Instruction Class
ORVDX64 I

# lv.xor Vector Exclusive Or lv.xor

| 31 . . . . 26 | 25 . . . 21 | 20 . . . 16 | 15 . . . 11 | 10 . 8 | 7 . . . . . . 0 |
|---|---|---|---|---|---|
| opcode 0xa | D | A | B | reserved | opcode 0x7b |
| 6 bits | 5 bits | 5 bits | 5 bits | 3 bits | 8bits |

## 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] < - rA[63:0] 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 arcitecture 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 Table 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 except if the current instruction is in the delay slot. If the PC points to the delay slot instruction, PC-4 is saved to the current EPCR and SR[DSX] is set. Table 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.

| 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 | - |
| System Call | 7 | Address of next not | Address of just executed | - |

| Exception | Priority | EPCR (no delay slot) | EPCR (delay slot) | EEAR |
|---|---|---|---|---|
| | | executed instruction | 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.

## 6.4 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 invokation 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:

ü When an exception occurs, it holds the previous CID.

ü 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:

ü Switching between threads

ü Handling exceptions

ü 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 |
| 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 defered 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.msync** 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.

# 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.

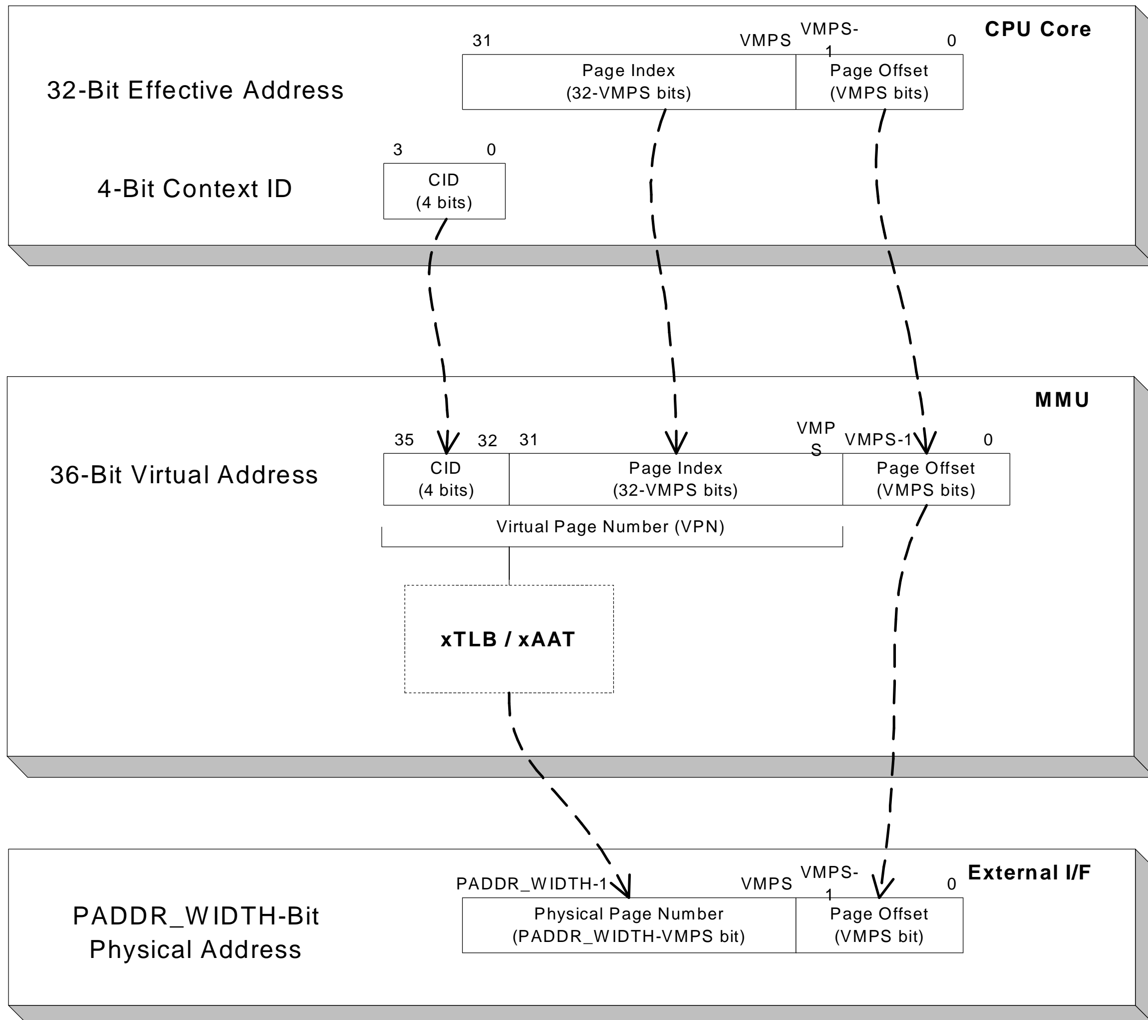



**Figure 8-1. Translation of Effective to Physical Address – Simplified block diagram for 32-bit processor implementations**

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 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 "**Error! Reference source not found.**" on page **Error! Bookmark not defined.** for a more detailed description of exception processing.

## 8.4 MMU Special-Purpose Registers

Table 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.

Table 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 |
| 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 |
| 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<br>0 DTLB ready for operation<br>1 DTLB flush request/status |
| PTBP | Page Table Base Pointer<br>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<br>0 Load operation in supervisor mode not permitted<br>1 Load operation in supervisor mode permitted |
| SWEx | Supervisor Write Enable x<br>0 Store operation in supervisor mode not permitted<br>1 Store operation in supervisor mode permitted |
| UREx | User Read Enable x<br>0 Load operation in user mode not permitted<br>1 Load operation in user mode permitted |
| UWEx | User Write Enable x<br>0 Store operation in user mode not permitted<br>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 DTLBWyTRx 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<br>0 ITLB ready for operation<br>1 ITLB flush request/status |
|---|---|
| PTBP | Page Table Base Pointer<br>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<br>0 Instruction fetch in supervisor mode not permitted<br>1 Instruction fetch in supervisor mode permitted |
| UXEx | User Execute Enable x<br>0 Instruction fetch in user mode not permitted<br>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 ITLBWyTRx 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<br>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 | 31-12 |
|---|---|
| Identifier | VPN |
| Reset | X |
| R/W | R/W |

| Bit | 11-8 | 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<br>0 TLB entry invalid<br>1 TLB entry valid |
| PL1 | Page Level 1<br>0 Page level is 2<br>1 Page level is 1 |
| CID | Context ID<br>0-15 TLB entry translates for CID |

| | |
|---|---|
| LRU | Last Recently used<br>0-3 Index in LRU queue (lower the number, more recent access) |
| VPN | Virtual Page Number<br>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 (DTLBWyTR0-DTLBWyTR127)

The DTLBWyTR registers are 32-bit special-purpose supervisor-level registers accessible with the l.mtspr/l.mfspr instructions in supervisor mode.

Together with the DTLBWyMR 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 DTLBWyMR registers and match logic they form a core of the DMMU.

| Bit | 31-12 | 11-10 | 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<br>0 Data cache coherency is not enforced for this page<br>1 Data cache coherency is enforced for this page |
| CI | Cache Inhibit<br>0 Cache is enabled for this page<br>1 Cache is disabled for this page |
| WBC | Write-Back Cache<br>0 Data cache uses write-through strategy for data from this page<br>1 Data cache uses write-back strategy for data from this page |
| WOM | Weakly-Ordered Memory<br>0 Strongly-ordered memory model for this page<br>1 Weakly-ordered memory model for this page |

| | |
|---|---|
| A | Accessed<br>0 Page was not accessed<br>1 Page was accessed |
| D | Dirty<br>0 Page was not modified<br>1 Page was modified |
| URE | User Read Enable x<br>0 Load operation in supervisor mode not permitted<br>1 Load operation in supervisor mode permitted |
| UWE | User Write Enable x<br>0 Store operation in supervisor mode not permitted<br>1 Store operation in supervisor mode permitted |
| SRE | Supervisor Read Enable x<br>0 Load operation in user mode not permitted<br>1 Load operation in user mode permitted |
| SWE | Supervisor Write Enable x<br>0 Store operation in user mode not permitted<br>1 Store operation in user mode permitted |
| PPN | Physical Page Number<br>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 | 31-12 | 11-8 | 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<br>0 Data cache coherency is not enforced for this page<br>1 Data cache coherency is enforced for this page |
| CI | Cache Inhibit<br>0 Cache is enabled for this page<br>1 Cache is disabled for this page |
| WBC | Write-Back Cache<br>0 Data cache uses write-through strategy for data from this page<br>1 Data cache uses write-back strategy for data from this page |
| WOM | Weakly-Ordered Memory<br>0 Strongly-ordered memory model for this page<br>1 Weakly-ordered memory model for this page |
| A | Accessed<br>0 Page was not accessed<br>1 Page was accessed |
| D | Dirty<br>0 Page was not modified<br>1 Page was modified |
| SXE | Supervisor Execute Enable x<br>0 Instruction fetch operation in supervisor mode not permitted<br>1 Instruction fetch operation in supervisor mode permitted |
| UXE | User Execute Enable x<br>0 Instruction fetch operation in user mode not permitted<br>1 Instruction fetch operation in user mode permitted |
| PPN | Physical Page Number<br>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<br>0 TLB entry invalid<br>1 TLB entry valid |
| CID | Context ID<br>0-15 TLB entry translates for CID |
| PS | Page Size<br>0 16 Mbyte page<br>1 32 Gbyte page |
| VPN | Virtual Page Number<br>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<br>0 Data cache coherency is not enforced for this page<br>1 Data cache coherency is enforced for this page |
| CI | Cache Inhibit<br>0 Cache is enabled for this page<br>1 Cache is disabled for this page |
| WBC | Write-Back Cache<br>0 Data cache uses write-through strategy for data from this page<br>1 Data cache uses write-back strategy for data from this page |
| WOM | Weakly-Ordered Memory<br>0 Strongly-ordered memory model for this page<br>1 Weakly-ordered memory model for this page |
| A | Accessed<br>0 Page was not accessed |

| | |
|---|---|
| | 1 Page was accessed |
| D | Dirty<br>0 Page was not modified<br>1 Page was modified |
| SRE | Supervisor Read Enable x<br>0 Load operation in supervisor mode not permitted<br>1 Load operation in supervisor mode permitted |
| SWE | Supervisor Write Enable x<br>0 Store operation in supervisor mode not permitted<br>1 Store operation in supervisor mode permitted |
| URE | User Read Enable x<br>0 Load operation in user mode not permitted<br>1 Load operation in user mode permitted |
| UWE | User Write Enable x<br>0 Store operation in user mode not permitted<br>1 Store operation in user mode permitted |
| PPN | Physical Page Number<br>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<br>0 Data cache coherency is not enforced for this page<br>1 Data cache coherency is enforced for this page |
| CI | Cache Inhibit<br>0 Cache is enabled for this page<br>1 Cache is disabled for this page |
| WBC | Write-Back Cache<br>0 Data cache uses write-through strategy for data from this page<br>1 Data cache uses write-back strategy for data from this page |
| WOM | Weakly-Ordered Memory<br>0 Strongly-ordered memory model for this page<br>1 Weakly-ordered memory model for this page |
| A | Accessed<br>0 Page was not accessed<br>1 Page was accessed |
| D | Dirty<br>0 Page was not modified<br>1 Page was modified |
| SXE | Supervisor Execute Enable x<br>0 Instruction fetch operation in supervisor mode not permitted<br>1 Instruction fetch operation in supervisor mode permitted |
| UXE | User Execute Enable x<br>0 Instruction fetch operation in user mode not permitted<br>1 Instruction fetch operation in user mode permitted |
| PPN | Physical Page Number<br>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.

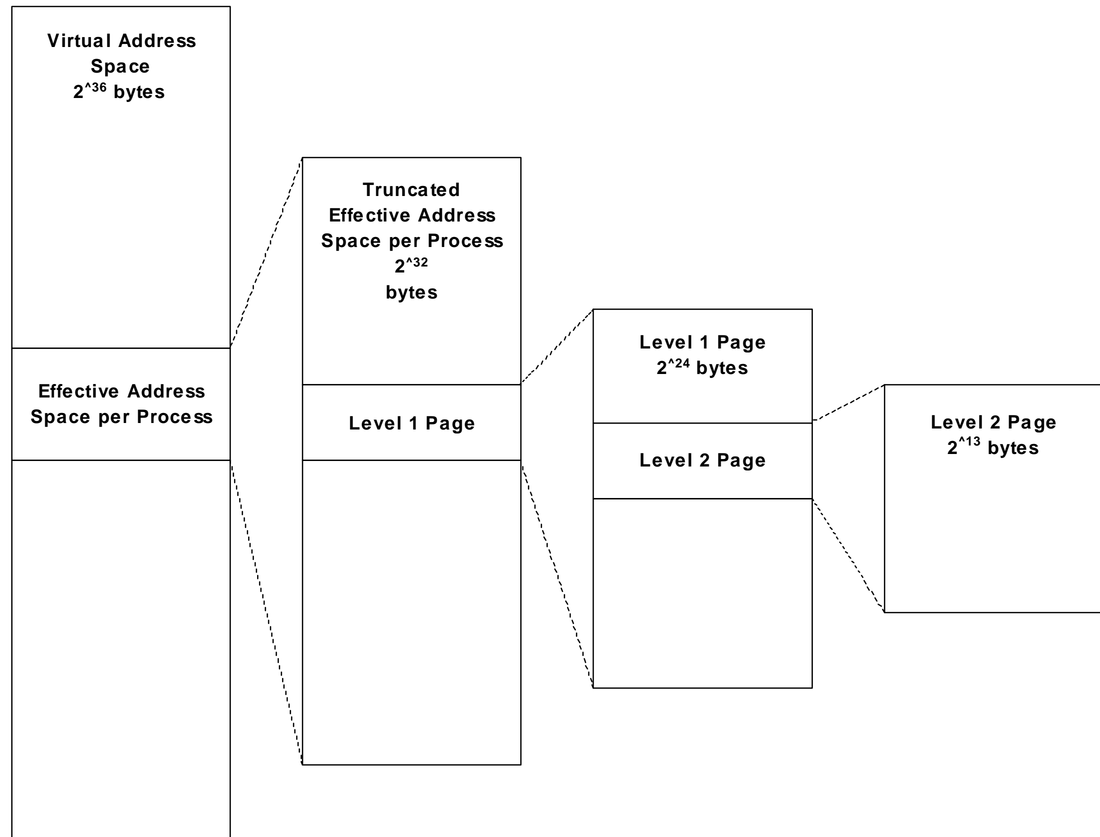



**Figure 8-2. Memory Divided Into L1 and L2 pages**

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.

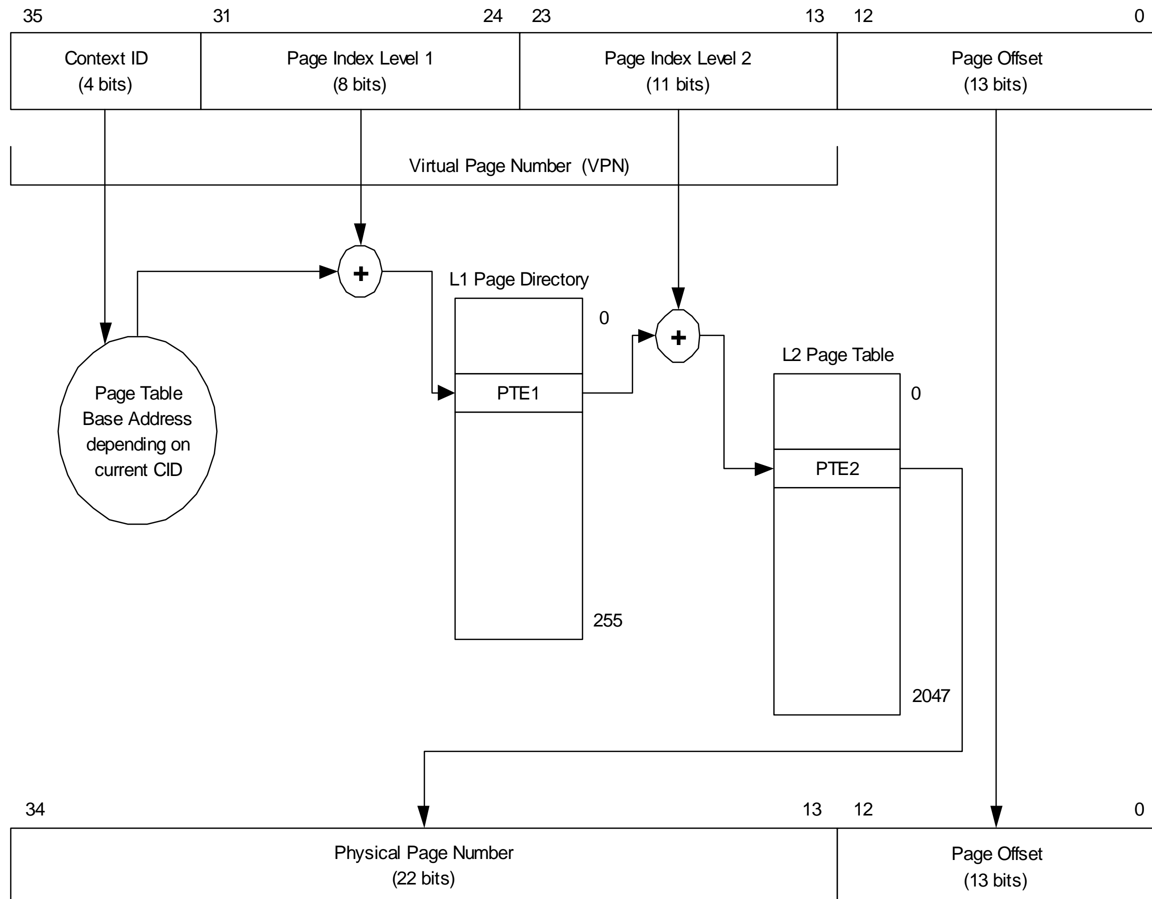



**Figure 8-3. Address Translation Mechanism using Two-Level Page Table**

Figure 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.

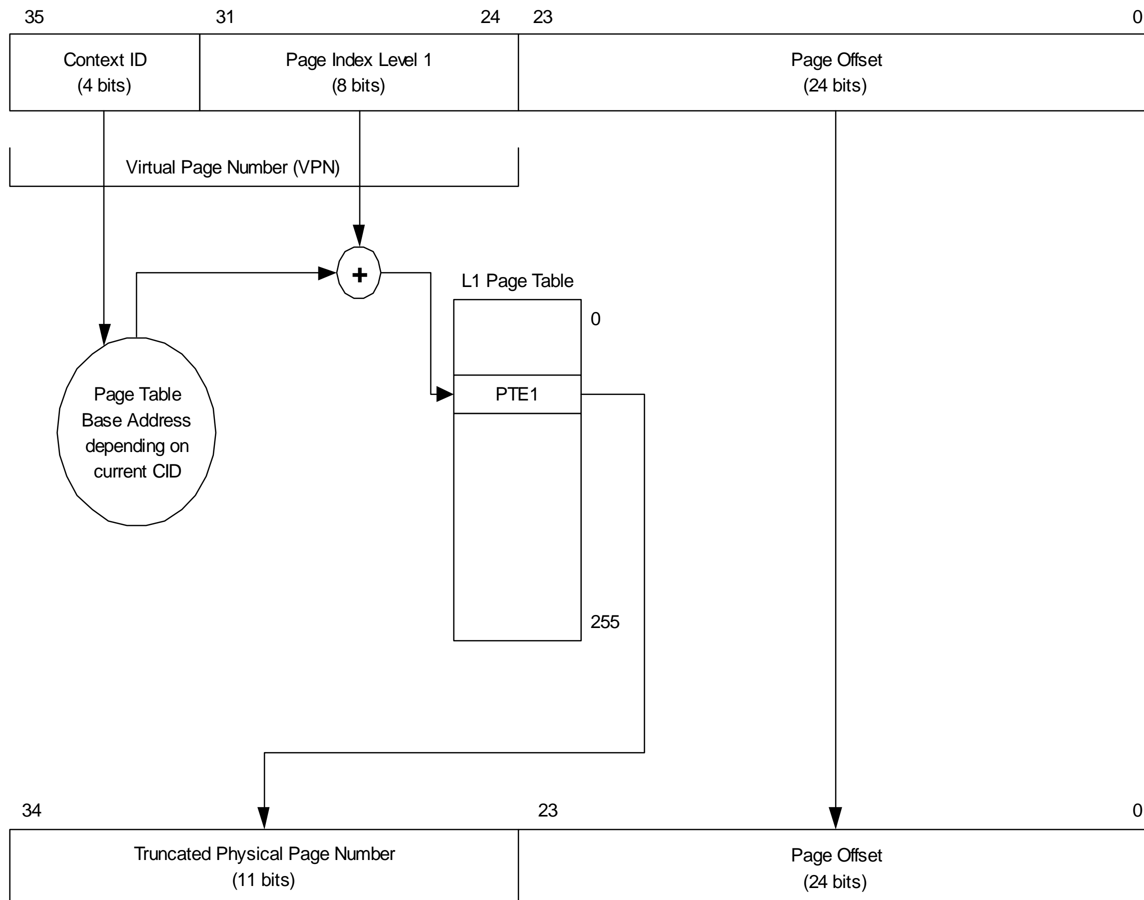



**Figure 8-4. Address Translation Mechanism using only L1 Page Table**

Figure 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.

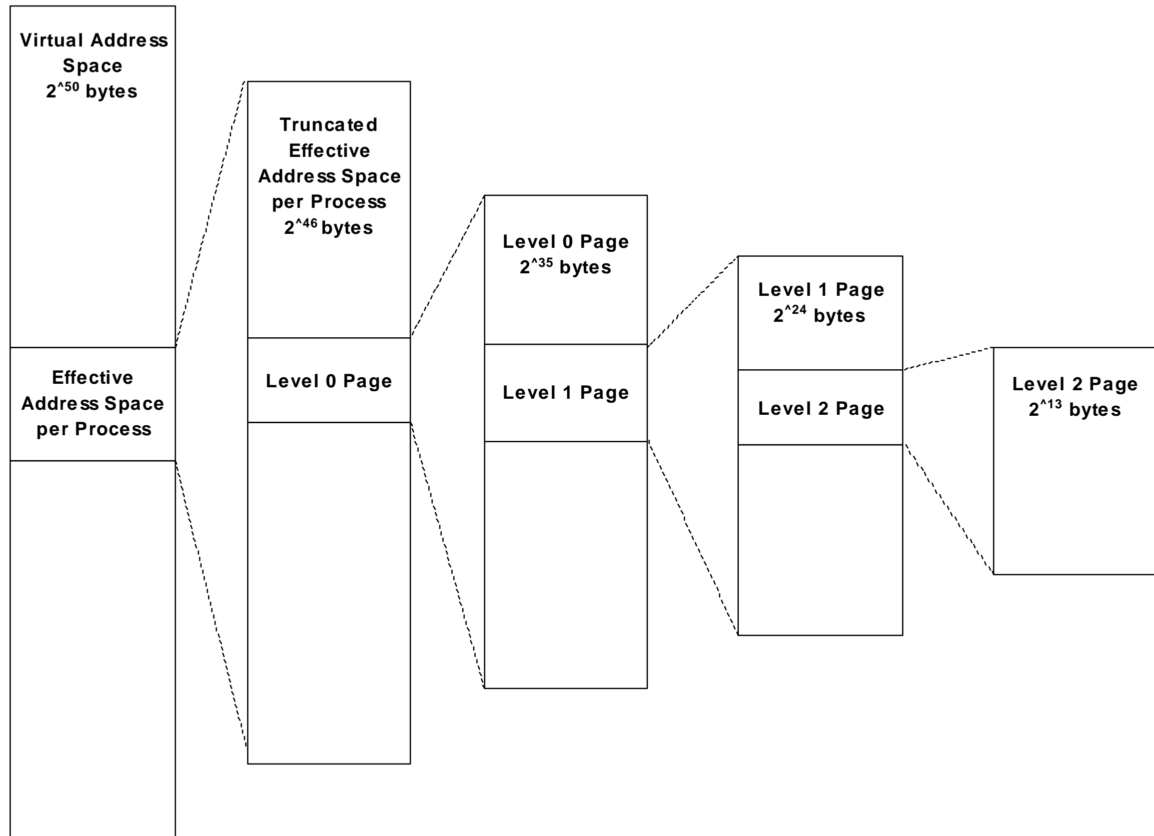



**Figure 8-5. Memory Divided Into L0, L1 and L2 pages**

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.

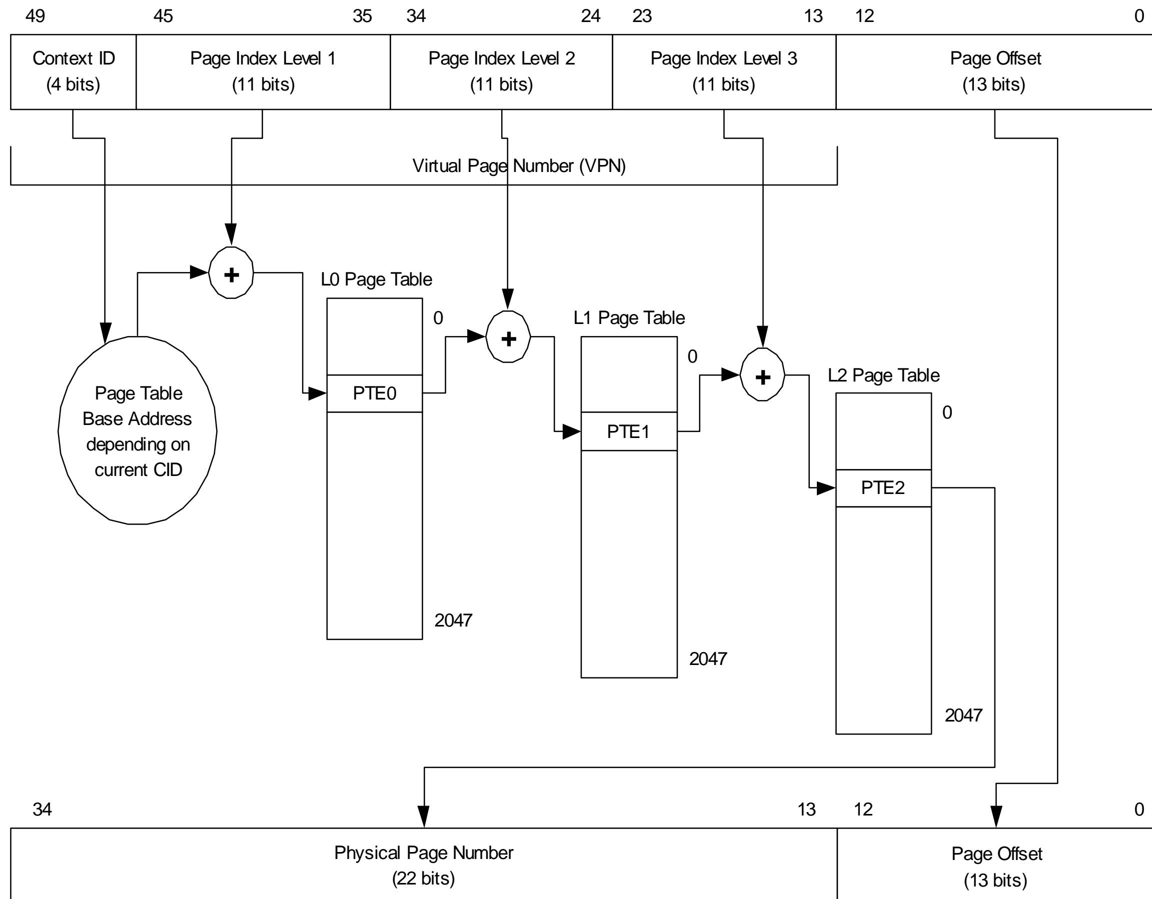



**Figure 8-6. Address Translation Mechanism using Three-Level Page Table**

Figure 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.

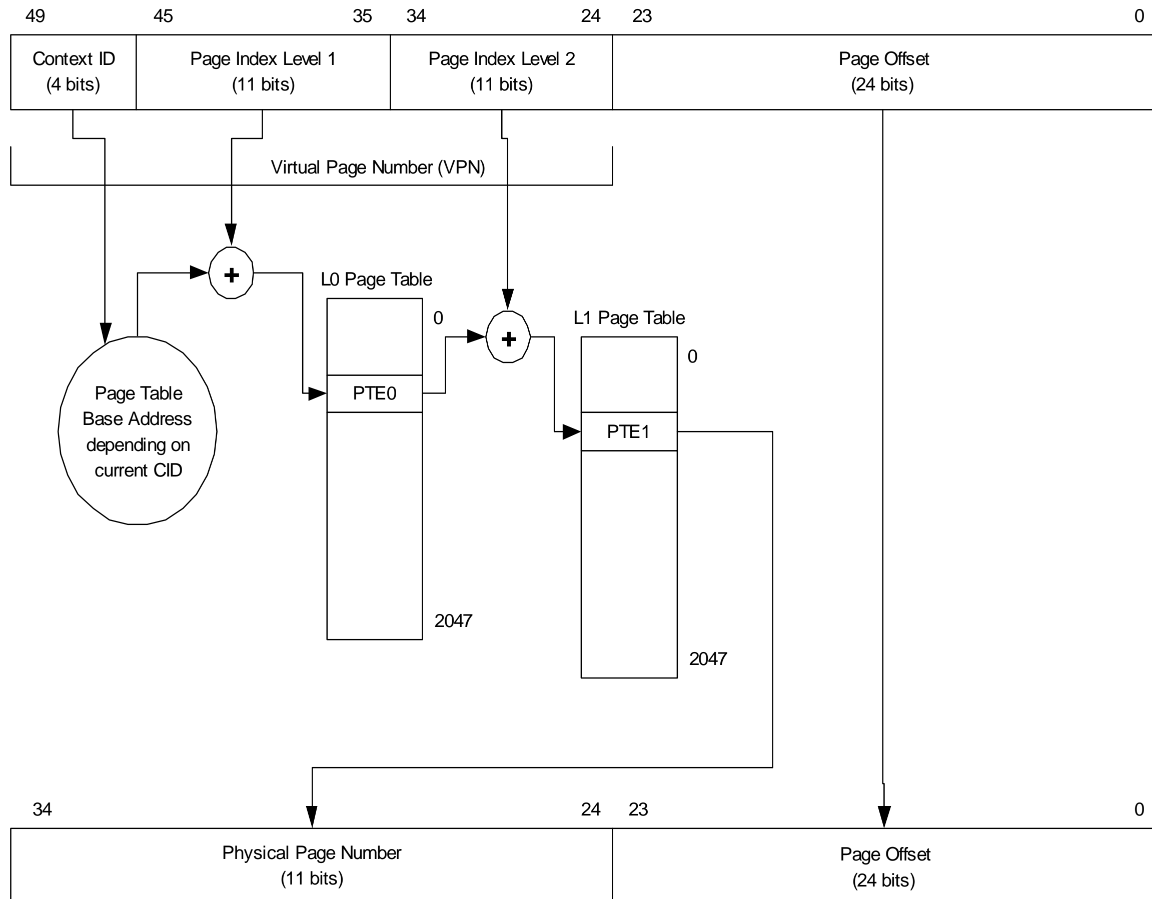



**Figure 8-7. Address Translation Mechanism using Two-Level Page Table**

Figure 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**

Table 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.

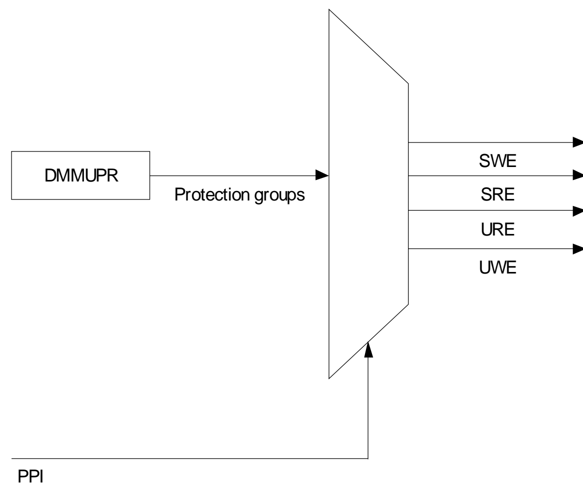



**Figure 8-8. Selection of Page Protection Attributes for Data Accesses**

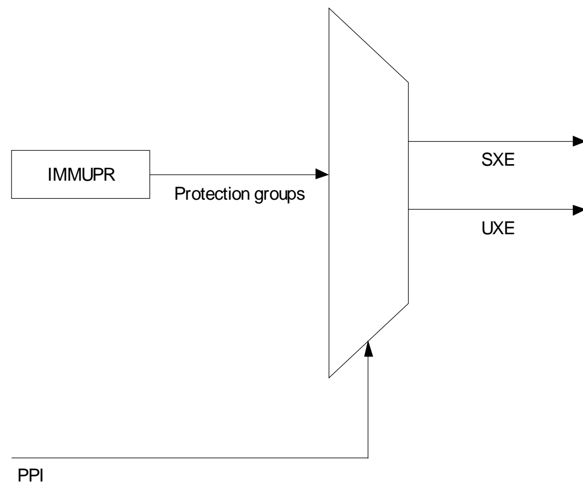



**Figure 8-9. Selection of Page Protection Attributes for Instruction Fetch Accesses**

## 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 containts PTEs for actual address translation.

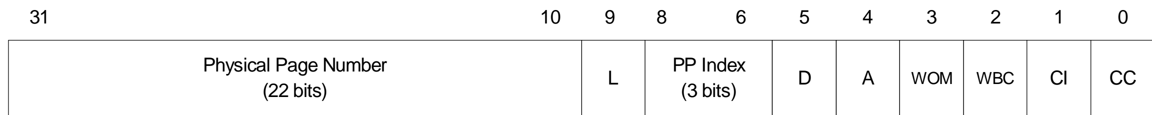



**Figure 8-10. Page Table Entry Format**

| | |
|---|---|
| CC | Cache Coherency<br>0 Data cache coherency is not enforced for this page<br>1 Data cache coherency is enforced for this page |
| CI | Cache Inhibit<br>0 Cache is enabled for this page<br>1 Cache is disabled for this page |
| WBC | Write-Back Cache<br>0 Data cache uses write-through strategy for data from this page<br>1 Data cache uses write-back strategy for data from this page |
| WOM | Weakly-Ordered Memory<br>0 Strongly-ordered memory model for this page<br>1 Weakly-ordered memory model for this page |
| A | Accessed<br>0 Page was not accessed<br>1 Page was accessed |
| D | Dirty<br>0 Page was not modified<br>1 Page was modified |
| PPI | Page Protection Index<br>0 PTE is invalid<br>1-7 Selects a group of six bits from a set of seven protection attribute groups in xMMUCR |
| L | Last<br>0 PTE from page directory pointing to next page directory/table<br>1 Last PTE in a linked form of PTEs (describing the actual page) |
| PPN | Physical Page Number<br>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 syncronization 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 accidently 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

Table 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 intruction 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 |

| GRP # | REG # | REG NAME | USER MODE | SUPV MODE | DESCRIPTION |
|---|---|---|---|---|---|
| | | | | | Register |
| 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<br>0000 0000 All ways disabled/locked<br>…<br>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<br>0000 0000 All ways disabled/locked<br>…<br>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<br>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 |

| Res<br>et | 0 |
|---|---|
| R/<br>W | Write only |

| EA | Effective Address<br>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 |
|---|---|
| Ide<br>ntifier | EA |
| Res<br>et | 0 |
| R/<br>W | Write Only |

| EA | Effective Address<br>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<br>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<br>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<br>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<br>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<br>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 gurantee cache coherency.

# 10 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.

## 10.1 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.

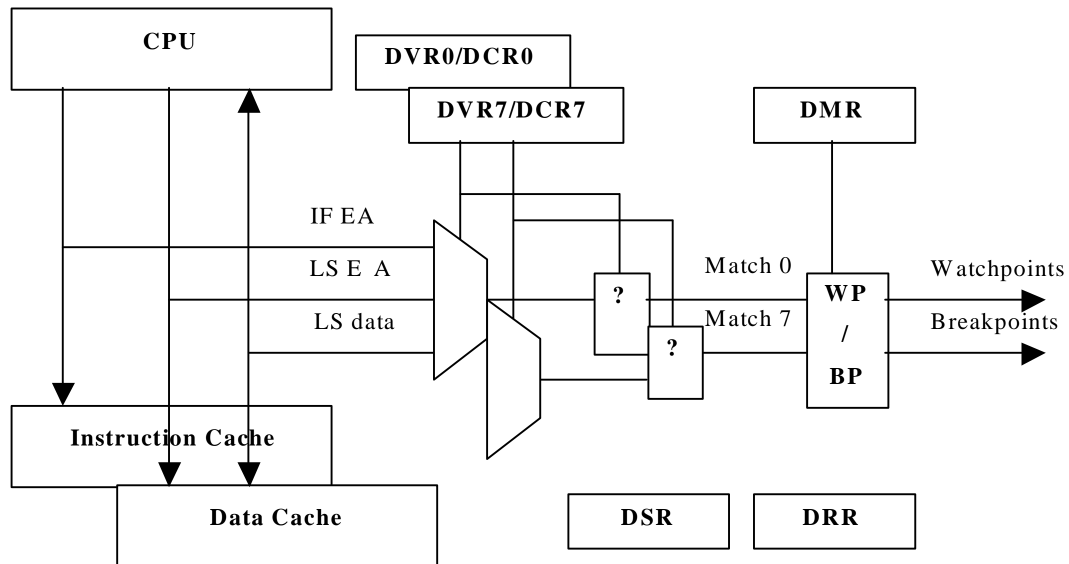



**Figure 10-1. Block Diagram of Debug Support**

# 10.2Debug 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 10-1. DVR Field Descriptions**

# 10.3 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<br>0 Corresponding DVR/DCR pair is not present<br>1 Corresponding DVR/DCR pair is present |
| CC | Compare Condition<br>000 Masked<br>001 Equal<br>010 Less than<br>011 Less than or equal<br>100 Greater than<br>101 Greater than or equal<br>110 Not equal<br>111 Reserved |
| SC | Signed Comparison<br>0 Compare using unsigned integers<br>1 Compare using signed integers |
| CT | Compare To<br>000 Comparison disabled<br>001 Instruction fetch EA<br>010 Load EA<br>011 Store EA<br>100 Load data<br>101 Store data<br>110 Load/Store EA<br>111 Load/Store data |

**Table 10-2. DCR Field Descriptions**

# 10.4 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<br>00 Watchpoint 0 = Match 0<br>01 Watchpoint 0 = Match 0 & External Watchpoint<br>10 Watchpoint 0 = Match 0 \| External Watchpoint<br>11 Reserved |
| CW1 | Chain Watchpoint 1<br>00 Watchpoint 1 = Match 1<br>01 Watchpoint 1 = Match 1 & Watchpoint 0<br>10 Watchpoint 1 = Match 1 \| Watchpoint 0<br>11 Reserved |
| CW2 | Chain Watchpoint 2<br>00 Watchpoint 2 = Match 2<br>01 Watchpoint 2 = Match 2 & Watchpoint 1<br>10 Watchpoint 2 = Match 2 \| Watchpoint 1<br>11 Reserved |
| CW3 | Chain Watchpoint 3 |

| | |
|---|---|
| | 00 Watchpoint 3 = Match 3<br>01 Watchpoint 3 = Match 3 & Watchpoint 2<br>10 Watchpoint 3 = Match 3 \| Watchpoint 2<br>11 Reserved |
| CW4 | Chain Watchpoint 4<br>00 Watchpoint 4 = Match 4<br>01 Watchpoint 4 = Match 4 & External Watchpoint<br>10 Watchpoint 4 = Match 4 \| External Watchpoint<br>11 Reserved |
| CW5 | Chain Watchpoint 5<br>00 Watchpoint 5 = Match 5<br>01 Watchpoint 5 = Match 5 & Watchpoint 4<br>10 Watchpoint 5 = Match 5 \| Watchpoint 4<br>11 Reserved |
| CW6 | Chain Watchpoint 6<br>00 Watchpoint 6 = Match 6<br>01 Watchpoint 6 = Match 6 & Watchpoint 5<br>10 Watchpoint 6 = Match 6 \| Watchpoint 5<br>11 Reserved |
| CW7 | Chain Watchpoint 7<br>00 Watchpoint 7 = Match 7<br>01 Watchpoint 7 = Match 7 & Watchpoint 6<br>10 Watchpoint 7 = Match 7 \| Watchpoint 6<br>11 Reserved |
| CW8 | Chain Watchpoint 8<br>00 Watchpoint 8 = Watchpoint counter 0 match<br>01 Watchpoint 8 = Watchpoint counter 0 match & Watchpoint 3<br>10 Watchpoint 8 = Watchpoint counter 0 match \| Watchpoint 3<br>11 Reserved |
| CW9 | Chain Watchpoint 9<br>00 Watchpoint 9 = Watchpoint counter 1 match<br>01 Watchpoint 9 = Watchpoint counter 1 match & Watchpoint 7<br>10 Watchpoint 9 = Watchpoint counter 1 match \| Watchpoint 7<br>11 Reserved |
| ST | Single-step Trace<br>0 Single-step trace disabled<br>1 Every executed instruction causes trap exception |
| BT | Branch Trace<br>0 Branch trace disabled<br>1 Every executed branch instruction causes trap exception |

**Table 10-3. DMR1 Field Descriptions**

# 10.5 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 ot 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<br>0 Counter 0 disabled<br>1 Counter 0 enabled |
| WCE1 | Watchpoint Counter Enable 1<br>0 Counter 1 disabled<br>1 Counter 1 enabled |
| AWTC | Assign Watchpoints to Counter<br>00 0000 0000 All Watchpoints increment counter 0<br>00 0000 0001 Watchpoint 0 increments counter 1<br>…<br>00 0000 1111 First four watchpoints increment counter 1, rest increment counter 0<br>…<br>11 1111 1111 All watchpoints increment counter 1 |
| WGB | Watchpoints Generating Breakpoint (trap exception)<br>00 0000 0000 Breakpoint disabled<br>00 0000 0001 Watchpoint 0 generates breakpoint<br>…<br>01 0000 0000 Watchpoint counter 0 generates breakpoint<br>…<br>11 1111 1111 All watchpoints generate breakpoint |
| WBS | Watchpoints Breakpoint Status<br>00 0000 0000 No watchpoint caused breakpoint<br>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 10-4. DMR2 Field Descriptions**

# 10.6 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<br>N 16-bit counter of generated watchpoints assigned to this counter |
|---|---|
| MATCH | N 16-bit value that when matched generates a watchpoint |

**Table 10-5. DWCR Field Descriptions**

# 10.7 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<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| BUSEE | Bus Error Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| DPFE | Data Page Fault Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| IPFE | Instruction Page Fault Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| TTE | Tick Timer Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| AE | Alignment Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| IIE | Illegal Instruction Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| INTE | Interrupt Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| DME | DTLB Miss Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |

| | |
|---|---|
| IME | ITLB Miss Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| RE | Range Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| SCE | System Call Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| FPE | Floating Point Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |
| TE | Trap Exception<br>0 This exception does not transfer control to the development I/F<br>1 This exception transfers control to the development interface |

**Table 10-6. DSR Field Descriptions**

## 10.8 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<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| BUSEE | Bus Error Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| DPFE | Data Page Fault Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| IPFE | Instruction Page Fault Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| TTE | Tick Timer Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| AE | Alignment Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| IIE | Illegal Instruction Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| INTE | Interrupt Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| DME | DTLB Miss Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| IME | ITLB Miss Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| RE | Range Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| SCE | System Call Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transfered control to the development interface |
| FPE | Floating Point Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transferred control to the development interface |
| TE | Trap Exception<br>0 This exception did not transfer control to the development I/F<br>1 This exception transferred control to the development interface |

**Table 10-7. DRR Field Descriptions**

# 11 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.

## 11.1 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.

## 11.2 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 11-1. PCCR0 Field Descriptions**

# 11.3 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<br>0 Counter not present<br>1 Counter present |
|---|---|
| CISM | Count in Supervisor Mode<br>0 Counter disabled in supervisor mode |

| | |
|---|---|
| | 1 Counter counts events in supervisor mode |
| CIUM | Count in User Mode<br>0 Counter disabled in user mode<br>1 Counter counts events in user mode |
| LA | Load Access event<br>0 Event ignored<br>1 Count load accesses |
| SA | Store Access event<br>0 Event ignored<br>1 Count store accesses |
| IF | Instruction Fetch event<br>0 Event ignored<br>1 Count instruction fetches |
| DCM | Data Cache Miss event<br>0 Event ignored<br>1 Count data cache missed |
| ICM | Instruction Cache Miss event<br>0 Event ignored<br>1 Count instruction cache misses |
| IFS | Instruction Fetch Stall event<br>0 Event ignored<br>1 Count instruction fetch stalls |
| LSUS | LSU Stall event<br>0 Event ignored<br>1 Count LSU stalls |
| BS | Branch Stalls event<br>0 Event ignored<br>1 Count branch stalls |
| DTLBM | DTLB Miss event<br>0 Event ignored<br>1 Count DTLB misses |
| ITLBM | ITLB Miss event<br>0 Event ignored<br>1 Count ITLB misses |
| DDS | Data Dependency Stalls event<br>0 Event ignored<br>1 Count data dependency stalls |
| WPE | Watchpoint Events<br>000 0000 0000 All watchpoint events ignored<br>000 0000 0001 Watchpoint 0 counted<br>…<br>111 1111 1111 All watchpoints counted |

**Table 11-2. PCMR Field Descriptions**

# 12 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.

## 12.1 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.

## 12.2 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<br>0 Full speed<br>1-15 Logarithmic clock frequency reduction |
| DME | Doze Mode Enable<br>0 Doze mode not enabled<br>1 Doze mode enabled |
| SME | Sleep Mode Enable<br>0 Sleep mode not enabled<br>1 Sleep mode enabled |
| DCGE | Dynamic Clock Gating Enable<br>0 Dynamic clock gating not enabled<br>1 Dynamic clock gating enabled |
| SUME | Suspend Mode Enable<br>0 Suspend mode not enabled<br>1 Suspend mode enabled |

**Table 12-1. PMR Field Descriptions**

# 13 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.

## 13.1 Features

The OpenRISC 1000 architecture defines an interrupt controller facility with up to 32 interrupt inputs:

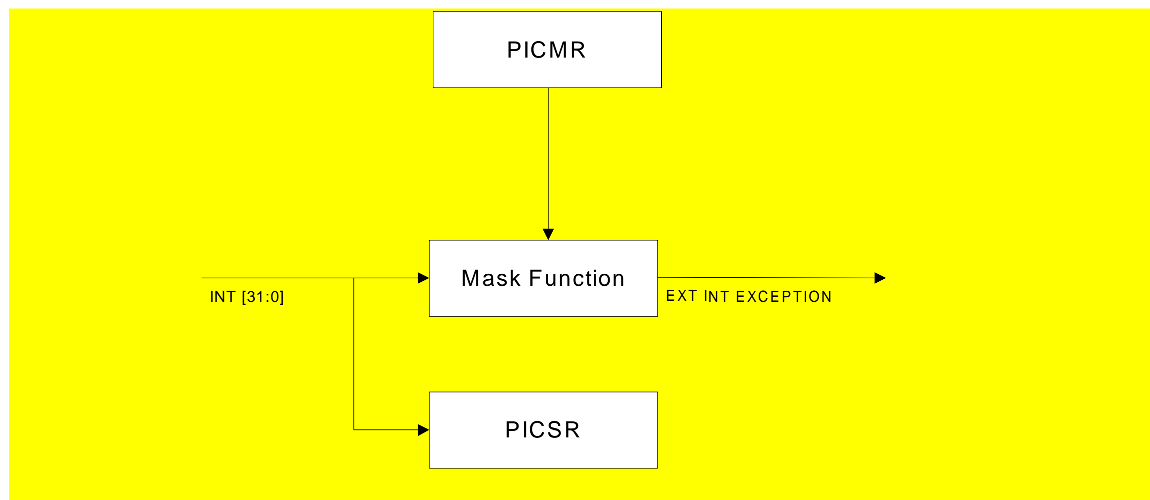



**Figure 13-1. Programmable Interrupt Controller Block Diagram**

## 13.2 PIC Mask Register (PICMR)

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

PICMR is used to mask or unmask 32 programmable interrupt sources.

| Bit | 31-0 |
|---|---|
| Identifier | IUM |
| Reset | 0 |
| R/W | R/W |

| IUM | Interrupt UnMask<br>0x00000000 All interrupts are masked<br>0x00000001 Interrupt input 0 is enabled, all others are masked<br>…<br>0xFFFFFFFF All interrupt inputs are enabled |
|---|---|

**Table 13-1. PICMR Field Descriptions**

## 13.3 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 | Interrupt Status<br>0x00000000 All interrupts are inactive<br>0x00000001 Interrupt input 0 is pending<br>…<br>0xFFFFFFFF All interrupts are pending |

**Table 13-2. PICSR Field Descriptions**

# 14 Tick Timer Facility (Optional)

This chapter describes the OpenRISC 1000 tick timer facility. It is optional and an implementation may chose 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 stoped 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.

## 14.1 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

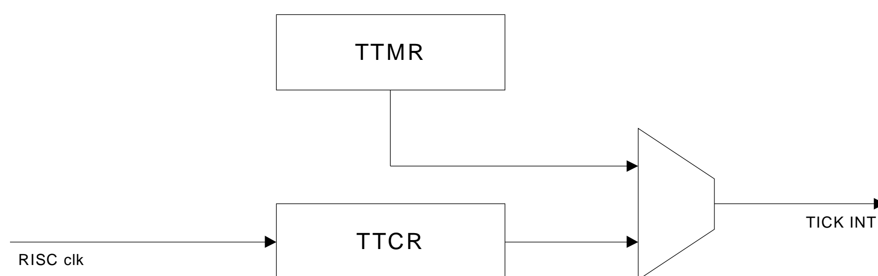



**Figure 14-1. Tick Timer Block Diagram**

# 14.2 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 occured 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.

# 14.3 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].

## 14.3.1 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.

## 14.3.2 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.

## 14.3.3 One-shot timer

In one-shot timeing 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.

### 14.3.4 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.

## 14.4Tick 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<br>0x0000000 Shortest comparison time period<br>…<br>0xFFFFFFF Longest comparison time period |
| IP | Interrupt Pending<br>0 Tick timer interrupt is not pending<br>1 Tick timer interrupt pending (write ‘0’ to clear it) |
| IE | Interrupt Enable<br>0 Tick timer does not generate tick timer interrupt<br>1 Tick timer generates tick timer interrupt when TTMR[TP] matches TTCR[27:0] |
| M | Mode<br>00 Tick timer is disabled<br>01 Timer is restarted when TTMR[TP] matches TTCR[27:0]<br>10 Timer stops when TTMR[TP] matches TTCR[27:0] (change TTCR to resume counting)<br>11 Timer does not stop when TTMR[TP] matches TTCR[27:0] |

**Table 14-1. TTMR Field Descriptions**

## 14.5 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<br>32-bit incrementing counter |
|---|---|

**Table 14-2. TTCR Field Descriptions**

# 15 OpenRISC 1000 Implementations

## 15.1 Overview

Implementations of the OpenRISC 1000 architecture come in different configurations and version releases.

Version and unit present registers both identify the version/release and its configuration. Detailed configuration for some units is available in configuration registers.

An operating system should read VR, UPR and the configuration registers, and adjust its own operation accordingly. Operating systems ported on a particular OpenRISC version should run on different configurations of this version without modifications.

## 15.2 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.

`

| Bit | 31-24 | 23-16 | 15-6 | 5-0 |
|---|---|---|---|---|
| Identifier | VER | CFG | Reserved | REV |
| Reset | - | - | X | - |
| R/W | R | R | R | R |

| | |
|---|---|
| REV | Revision<br>0..63 A 6-bit number that identifies various releases of a particular version. This number is changed for each revision of the device. |
| CFG | Configuration Template<br>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.<br>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. |

| VER | Version<br>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. |
|---|---|

**Table 15-1. VR Field Descriptions**

## 15.3 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 | UPR Present<br>0 UPR is not present<br>1 UPR is present |
|---|---|
| DCP | Data Cache Present<br>0 Unit is not present<br>1 Unit is present |
| ICP | Instruction Cache Present<br>0 Unit is not present<br>1 Unit is present |

| | |
|---|---|
| DMP | Data MMU Present<br>0 Unit is not present<br>1 Unit is present |
| IMP | Instruction MMU Present<br>0 Unit is not present<br>1 Unit is present |
| MP | MAC Present<br>0 Unit is not present<br>1 Unit is present |
| DUP | Debug Unit Present<br>0 Unit is not present<br>1 Unit is present |
| PCUP | Performance Counters Unit Present<br>0 Unit is not present<br>1 Unit is present |
| PMP | Power Management Present<br>0 Unit is not present<br>1 Unit is present |
| PICP | Programmable Interrupt Controller Present<br>0 Unit is not present<br>1 Unit is present |
| TTP | Tick Timer Present<br>0 Unit is not present<br>1 Unit is present |
| CUP | Custom Units Present |

**Table 15-2. UPR Field Descriptions**

# 15.4CPU 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 | 31-10 |
|---|---|
| Identifier | Reserved |
| Reset | - |
| R/W | 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<br>0 Zero shadow GPR files<br>15 Fifteen shadow GPR Files |
| CGF | Custom GPR File<br>0 GPR file has 32 registers<br>1 GPR file has less than 32 registers |
| OB32S | ORBIS32 Supported<br>0 Not supported<br>1 Supported |
| OB64S | ORBIS64 Supported<br>0 Not supported<br>1 Supported |
| OF32S | ORFPX32 Supported<br>0 Not supported<br>1 Supported |
| OF64S | ORFP64P Supported<br>0 Not supported<br>1 Supported |
| OV64S | ORVDX64 Supported<br>0 Not supported<br>1 Supported |

**Table 15-3. CPUCFGR Field Descriptions**

# 15.5 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<br>0 DTLB has one way<br>…<br>3 DTLB has four ways |
| NTS | Number of TLB Sets (entries per way)<br>0 DTLB has one set (entries per way)<br>…<br>7 DTLB has 128 sets (entries per way) |
| NAE | Number of ATB Entries<br>0 DATB does not exist<br>1 DATB has one entry<br>…<br>4 DATB has four entries<br>5..7 Invalid values |
| CRI | Control Register Implemented<br>0 DMMUCR not implemented<br>1 DMMUCR implemented |
| PRI | Protection Register Implemented<br>0 DMMUPR not implemented<br>1 DMMUPR implemented |
| TEIRI | TLB Entry Invalidate Register Implemented<br>0 DTLBEIR not implemented<br>1 DTLBEIR implemented |
| HTR | Hardware TLB Reload<br>0 TLB Entry reloaded in software<br>1 TLB Entry reloaded in hardware |

**Table 15-4. DMMUCFGR Field Descriptions**

# 15.6 IMMU 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<br>0 ITLB has one way<br>…<br>3 ITLB has four ways |
| NTS | Number of TLB Sets (entries per way)<br>0 ITLB has one set (entries per way)<br>…<br>7 ITLB has 128 sets (entries per way) |
| NAE | Number of ATB Entries<br>0 IATB does not exist<br>1 IATB has one entry<br>…<br>4 IATB has four entries<br>5..7 Invalid values |
| CRI | Control Register Implemented<br>0 IMMUCR not implemented<br>1 IMMUCR implemented |
| PRI | Protection Register Implemented |

| | |
|---|---|
| | 0 IMMUPR not implemented<br>1 IMMUPR implemented |
| TEIRI | TLB Entry Invalidate Register Implemented<br>0 ITLBEIR not implemented<br>1 ITLBEIR implemented |
| HTR | Hardware TLB Reload<br>0 ITLB Entry reloaded in software<br>1 ITLB Entry reloaded in hardware |

**Table 15-5. IMMUCFGR Field Descriptions**

# 15.7DC 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<br>0 DC has one way<br>…<br>5 DC has thirty-two ways |
| NCS | Number of Cache Sets (cache blocks per way)<br>0 DC has one set (cache blocks per way)<br>… |

| | |
|---|---|
| | 10 DC has 1024 sets (cache blocks per way) |
| BS | Cache Block Size<br>0 Cache block size 16 bytes<br>1 Cache block size 32 bytes |
| CWS | Cache Write Strategy<br>0 Cache write-through<br>1 Cache write-back |
| CCRI | Cache Control Register Implemented<br>0 Register is not implemented<br>1 Register is implemented |
| CBIRI | Cache Block Invalidate Register Implemented<br>0 Register is not implemented<br>1 Register is implemented |
| CBPRI | Cache Block Prefetch Register Implemented<br>0 Register is not implemented<br>1 Register is implemented |
| CBLRI | Cache Block Lock Register Implemented<br>0 Register is not implemented<br>1 Register is implemented |
| CBFRI | Cache Block Flush Register Implemented<br>0 Register is not implemented<br>1 Register is implemented |
| CBWBRI | Cache Block Write-Back Register Implemented<br>0 Register is not implemented<br>1 Register is implemented |

**Table 15-6. DCCFGR Field Descriptions**

## 15.8IC 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<br>0 IC has one way<br>…<br>5 IC has thirty-two ways |
| NCS | Number of Cache Sets (cache blocks per way)<br>0 IC has one set (cache blocks per way)<br>…<br>10 IC has 1024 sets (cache blocks per way) |
| BS | Cache Block Size<br>0 Cache block size 16 bytes<br>1 Cache block size 32 bytes |
| CCRI | Cache Control Register Implemented<br>0 Register is not implemented<br>1 Register is implemented |
| CBIRI | Cache Block Invalidate Register Implemented<br>0 Register is not implemented<br>1 Register is implemented |
| CBPRI | Cache Block Prefetch Register Implemented<br>0 Register is not implemented<br>1 Register is implemented |
| CBLRI | Cache Block Lock Register Implemented<br>0 Register is not implemented<br>1 Register is implemented |

**Table 15-7. ICCFGR Field Descriptions**

## 15.9 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<br>0 Debug unit has one DCR/DVR pair<br>…<br>7 Debug unit has eight DCR/DVR pairs |
| WPCI | Watchpoint Counters Implemented<br>0 Watchpoint counters not implemented<br>1 Watchpoint counters implemented |

**Table 15-8. DCFGR Field Descriptions**

## 15.10 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<br>0 One performance counter<br>…<br>7 Eight performance counters |

**Table 15-9. PCCFGR Field Descriptions**

# 16 Application Binary Interface

## 16.1 Data Representation

### 16.1.1 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 17. Similar relations between architecture and language types can be used for any other language.

| Type | C TYPE | SIZEOF | ALIGNMENT (BYTES) | OPENRISC EQUIVALENT |
|---|---|---|---|---|
| Integral | Char<br>Signed char | 1 | 1 | Signed byte |
| | Unsigned char | 1 | 1 | Unsigned byte |
| | Short<br>Signed short | 2 | 2 | Signed halfword |
| | Unsigned short | 2 | 2 | Unsigned halfword |
| | Int<br>Signed int<br>Long<br>Signed long<br>Enum | 4 | 4 | Signed singleword |
| | Unsigned int | 4 | 4 | Unsigned singleword |
| | Long long<br>Signed long long | 8 | 8 | Signed doubleword |
| | Unsigned long long | 8 | 8 | Unsigned doubleword |
| Pointer | Any-type *<br>Any-type (*) () | 4 | 4 | Unsigned singleword |
| Floating-point | Float | 4 | 4 | Single precision float |
| | Double | 8 | 8 | Double precision float |

**Table 16-1. Scalar Types**

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 Table 16-2. For vector assignments both side of assignment must be of the same vector type.

| VECTOR TYPE | SIZEOF | ALIGNMENT (BYTES) | OPENRISC EQUIVALENT |
|---|---|---|---|
| Vector char<br>Vector signed char | 8 | 8 | Vector of signed bytes |
| Vector unsigned char | 8 | 8 | Vector of unsigned bytes |
| Vector short<br>Vector signed short | 8 | 8 | Vector of signed halfwords |
| Vector unsigned short | 8 | 8 | Vector of unsigned halfwords |
| Vector int<br>Vector signed int<br>Vector long<br>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 16-2. Vector Types**

For alignment restrictions of all types see the chapter entitled “Addressing Modes and
Operand Conventions” on page 19.

### 16.1.2 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;
};
```

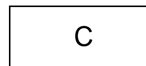



**Figure 16-1. Byte aligned, sizeof is 1**

```
struct {
  char   C;
  char   D;
  short  S;
  long   N;
```

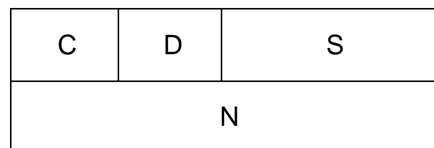



**Figure 16-2. No padding, sizeof is 8**

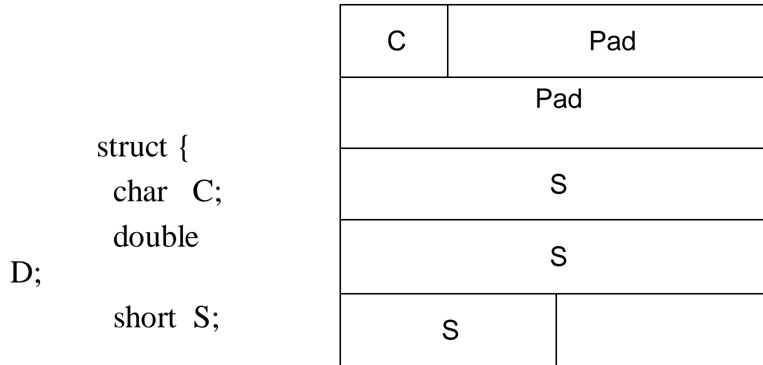



**Figure 16-3. Padding, sizeof is 18**

### 16.1.3 Bit-fields

C structure and union definitions can have elements defined by a specified number of bits. Table 16-3 describes valid bit-field types and their ranges.

| Bit-field Type | Width w [bits] | Range |
|---|---|---|
| Signed char<br>Char<br>Unsigned char | 1 to 8 | $-2^{w-1}$ to $2^{w-1}-1$<br>0 to $2^{w}-1$<br>0 to $2^{w}-1$ |
| Signed short<br>Short<br>Unsigned short | 1 to 16 | $-2^{w-1}$ to $2^{w-1}-1$<br>0 to $2^{w}-1$<br>0 to $2^{w}-1$ |
| Signed int<br>Int<br>Enum<br>Unsigned int<br>Signed long<br>Long<br>Unsigned long | 1 to 32 | $-2^{w-1}$ to $2^{w-1}-1$<br>0 to $2^{w}-1$<br>0 to $2^{w}-1$<br>0 to $2^{w}-1$<br>$-2^{w-1}$ to $2^{w-1}-1$<br>0 to $2^{w}-1$<br>0 to $2^{w}-1$ |

**Table 16-3. 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

```
struct {
  short   S:9;
  int     J:9;
  char    C;
  short   T:9;
  short   U:9;
  char    D;
};
```

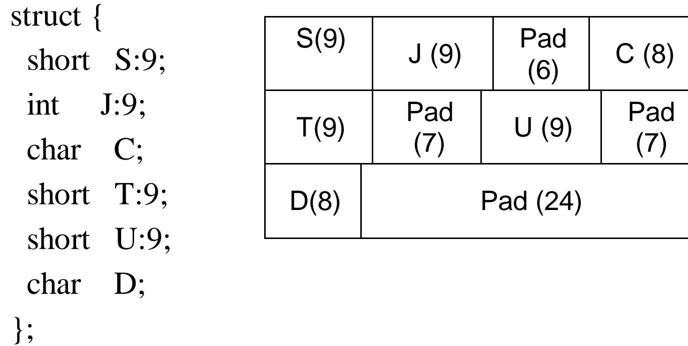



**Figure 16-4. Storage unit sharingand alignment padding, sizeof is 12**

# 16.2 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.

## 16.2.1 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 |
| 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 |

| Register | Preserved across function calls | Usage |
|---|---|---|
| R16 | Yes | Callee-saved register |
| R15 | No | Temporary register |
| R14 | Yes | Callee-saved register |
| R13 | No | Temporary register |
| R12 | No | Temporary register (RVH - Return value high 32 bits of 64-bit value on 32-bit system) |
| R11 | No | RV – Return value |
| R10 | Yes | Callee-saved register |
| R9 | Yes | LR – Link address register |
| R8 | No | Function parameter number 5 |
| R7 | No | Function parameter number 4 |
| R6 | No | Function parameter number 3 |
| R5 | No | Function parameter number 2 |
| R4 | No | Function parameter number 1 |
| R3 | No | Function parameter number 0 |
| R2 | Yes | FP - Frame pointer |
| R1 | Yes | SP - Stack pointer |
| R0 | - | Fixed to zero |

**Table 16-4. General-Purpose Registers**

Some registers have assigned roles:

| | |
|---|---|
| R0 [Zero] | Always fixed to zero. Even if it is writable in some embedded implementations, the software shouldn't modify it. |
| R1 [SP] | The **stack pointer** holds the limit of the current stack frame. The stack contents below the stack pointer are undefined. Stack pointer must be double 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. |
| R3 through R8 | **General-purpose parameters** use up to 6 general-purpose registers. Parameters beyond the sixth parameter appear on the stack. |
| R9 [LR] | **Link address** is the location of the function call instruction and is used to calculate where program execution should return after function completion. |
| R11 [RV] | **Return value** of the function. For *void* 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] | **Return value high** of the function. For functions returning 32-bit values this register can be considered temporary register. |

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.

### 16.2.2 The Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Table 16-5 shows the stack frame organization.

<table>
<tr><th>Position</th><th>Contents</th><th>Frame</th></tr>
<tr><td>FP + 4N<br>…<br>FP + 0</td><td>Parameter N<br>…<br>Parameter 0</td><td>Previous</td></tr>
<tr><td>FP – 4<br>FP – 8</td><td>Function variables</td><td rowspan="3">Current</td></tr>
<tr><td>SP + 4</td><td>Previous FP value</td></tr>
<tr><td>SP + 0</td><td>Return address</td></tr>
<tr><td>SP – 4<br>SP – 2096</td><td>For use by leaf functions w/o function prologue/epilogue</td><td rowspan="2">Future</td></tr>
<tr><td>SP – 2100<br>SP – 2536</td><td>For use by exception handlers</td></tr>
</table>

**Table 16-5. Stack Frame**

The stack pointer always points to the end of the latest allocated stack frame. All frames must be double word aligned. In code compiled for 32-bit implementations, upper halves of all double words are zero.

The first 2092 bytes below the current stack frame are reserved for leaf functions that do not need to modify their stack pointer. Exception handlers must guarantee that they will not use this area.

### 16.2.3 Parameter Passing

Functions receive their first 6 arguments in general-purpose parameter registers. If there are more than six arguments, 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 registers. 64-bit arguments are not aligned. For example *long long arg1, long arg2, long long arg3* are be passed in the following way: *arg1 in r3&r4, arg2 in r5, arg3 in r6&r7*.

### 16.2.4 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.

### 16.2.5 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.

## 16.3 Operating System Interface

### 16.3.1 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 and the machine state are saved
- The machine mode is changed to supervisor mode
- The execution resumes from a predefined exception vector address which is different for every exception

| Exception Type | Vector Offset | SIGNAL | Example |
|---|---|---|---|
| Reset | 0x100 | None | Reset |
| Bus Error | 0x200 | SIGBUS | Unexisting physical location, bus parity error. |
| Data Page Fault | 0x300 | SIGSEGV | Unammaped 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 16-6. Hardware Exceptions and Signals**

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. Table 16-6 shows how hardware exceptions can be mapped to signals if the operating system cannot complete the faulting exception.

### 16.3.2 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.

### 16.3.3 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.

### 16.3.4 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<br>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 16-7. 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*.

### 16.3.5 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.

### 16.3.6 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. System calls uses same software convention model as used with function calls, except additional register r11 specifies system call id.

## 16.4 Position-Independent Code

## 16.5 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.

### 16.5.1 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 Table 16-8.

| OR32 ELF e_ident Fields | | |
|---|---|---|
| e_ident[EI_CLASS] | ELFCLASS32 | For all 32-bit implementations |
| E_ident[EI_DATA] | ELFDATA2MSB | For all implementations |

**Table 16-8. *e_ident* Field Values**

The *e_flags* member of the ELF header contains values as shown in Table 16-9.

| 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 16-9. *e_flags* Field Values**

### 16.5.2 Sections

There are no OpenRISC section requirements beyond the base ELF standards.

### 16.5.3 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 refernced 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).

## 16.6COFF

### 16.6.1 Sections

### 16.6.2 Relocation