OpenRISC 1000

System Architecture Manual

22/Apr/2001, PRELIMINARY

1 Table of Contents

2 Table Of Figure

3 Table Of Tables

4 About this Manual

4.1 Brief Introduction

OpenRISC 1000 system architecture manual defines architecture for a family of open source, synthesizable RISC microprocessor cores. As architecture, OpenRISC 1000 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 architecture targets medium and high performance networking and embedded computer environments.

Architecture itself covers instruction set, register set, cache management and coherency, memory model, exception model, addressing modes, operands conventions and 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.

4.2 Authors

If you have contributed to this manual and your name isn't listed here, it is not meant as a slight. We just don't know about it. Send email to the maintainer(s), and we'll correct the situation.

NAME	E-MAIL	CONTRIBUTION
Damjan Lampret	lampret@opencores.org	Initial document
Chen-Min Chen	jimmy@ee.nctu.edu.tw	Appended 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

Figure 4-1. Authors of This Manual

4.3 Revision History

Revision history of this manual.

REVISION DATE	BY	MODIFICATIONS
15/Mar/2000	Damjan Lampret	Initial document
7/Apr/2001	Damjan Lampret	First public release
22/Apr/2001	Damjan Lampret	Incorporate changes from Johan and Matan

Figure 4-2. Revision History

4.4 Work in Progress

This document is *work in progress*. Anything in the manual can change until we will make our first silicon. Latest version is always available from OPENCORES CVS. See details how to get it on http://www.opencores.org/.

We are currently looking for people working on this document and for a maintainer of this document. If you would like to contribute send an email to one of the authors.

4.5 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 in this chapter
- Square brackets [ ] indicate an addressed field in a register or a numbered register in a register file

5 Architecture Overview

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

5.1 Features

OpenRISC 1000 architecture includes the following principal features:

- A completely open and free 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 bits wide instructions aligned on 32-bit boundaries in memory and operating on 32 bits and 64 bits data
  - OpenRISC Vector/DSP eXtension (ORVDX64) with 32 bits wide instructions aligned on 32-bit boundaries in memory and operating on 8, 16, 32 and 64 bits data
  - OpenRISC Floating-Point eXtension (ORFPX32/64) with 32 bits wide instructions aligned on 32-bit boundaries in memory and operating on 32 bits and 64 bits data
- Two simple memory addressing modes where memory address is calculated with:
  - Addition of register operand and signed 16-bit immediate
  - Addition of register operand and signed 16-bit immediate followed by update of register operand with calculated effective address
- Most instructions operate on two register operands (or one register and a constant), and 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 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 in either hardware or with assistance of implementation-specific software
- Low and high priority external exceptions (interrupts)
- Fast context switch support in register set, caches and MMUs

5.2 Introduction

OpenRISC 1000 architecture is completely open architecture. It defines architecture of a family of open source, RISC microprocessor cores. As architecture, OpenRISC 1000 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 fully 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. Architecture defines several features for networking and embedded computer environments. Most notable are several instruction extensions, configurable number of general-purpose registers, configurable cache and TLB sizes, dynamic power management support and space for user provided instructions. OpenRISC 1000 architecture is a predecessor of more powerful and richful next generation OpenRISC architectures.

Implementations of the OpenRISC 1000 architecture are available in full source from www.opencores.org and are supported with GNU software development tools and with a behavioral simulator. Most OpenRISC implementations are designed modular and vendor independent. They can be interfaced with other open source cores available from 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.

5.3 Acronyms and Abbreviations

ALU	Arithmetic logic unit

BAT	Block address translation
BIU	Bus interface unit
BTC	Branch target cache
CPU	Central processing unit
EA	Effective address
FPU	Floating-point unit
GPR	General purpose register
MMU	Memory management unit
PTE	Page table entry
R/W	Read/Write
RISC	Reduced instruction set computing
SMP	Symmetrical multi-processing
SMT	Simultaneous multi-threading
SPR	Special purpose register
TLB	Translation look aside buffer

Table 5-1. Acronyms and Abbreviations

5.4 Conventions

`l.mnemonic`	Identifies ORBIS32/64 instruction.
`lv.mnemonic`	Identifies ORVDX32/64 instruction.
`lf.mnemonic`	Identifies ORFPX32/64 instruction.
`0x`	Prefix 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 a one 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`	Actual processor implementing OpenRISC 1000 architecture.
`Module`	Sometimes referred to as a coprocessor. A unit in implementation usually with some special registers and controlling instructions. It can be defined by the architecture or it can be custom.
`Exception`	A vectored transfer of control to supervisor software through a 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[SUPV]=1).

Table 5-2. Conventions

5.5 Numbering

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

6 Addressing Modes and Operand Conventions

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

6.1 Memory Addressing Modes

The processor computes an effective address when executing memory access 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 is considered to wrap around from the maximum effective through effective address 0.

6.1.1 Register Indirect with Displacement

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

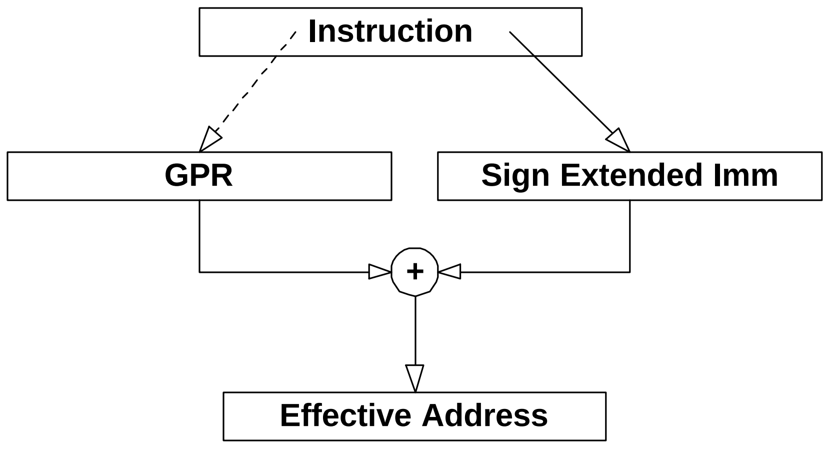


Figure 6-1. Register Indirect with Displacement Addressing

Figure 6-1 shows how an effective address is computed when using register indirect with displacement addressing mode.

6.1.2 Register Indirect with Displacement and Update

Load/store instructions using this address mode contain a signed 16-bit immediate which is sign extended and added to the contents of a general-purpose register specified in the instruction. Computed EA is then used to update the general-purpose register that was initially used to compute current EA.

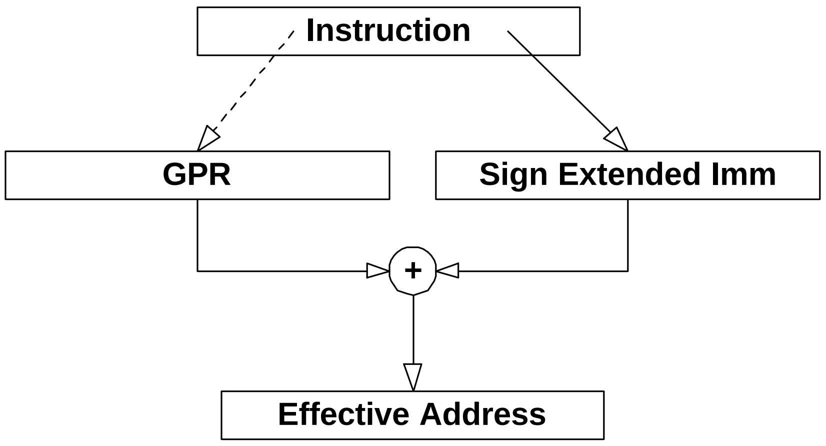


Figure 6-2. Register Indirect with Displacement and Update

Figure 6-2 shows how an effective address is computed when using register indirect with displacement and update addressing mode.

6.1.3 PC Relative

Branch instructions using this address mode contain a signed 26-bit immediate which is sign extended and added to the contents of a Program Counter register.

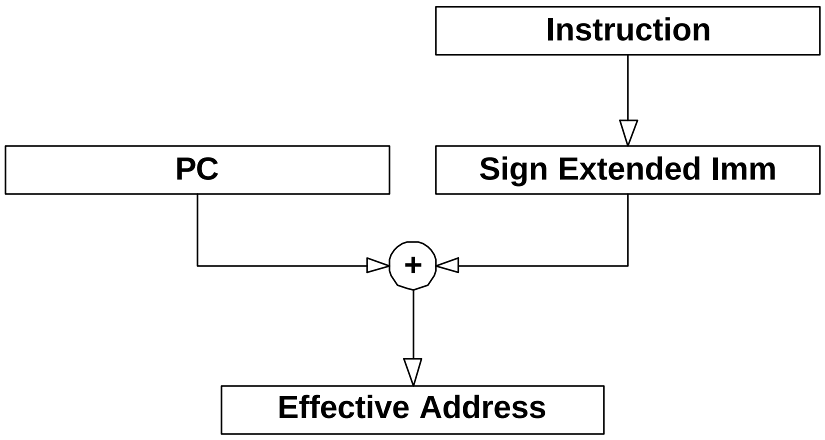


Figure 6-3. PC Relative Addressing

Figure 6-3 shows how an effective address is generated when using PC relative addressing mode.

6.2 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, 64-bit double precision float and 128-bit quad precision float. And it defines 64-bit vector of bytes, 64-bit vector of halfwords, 64-bit vector of singlewords and 64-bit vector of single precision floats.

OPENRISC TERM	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
Quad precision float	16	128
Vector of bytes	8	64
Vector of halfwords	8	64
Vector of singlewords	8	64
Vector of single precision floats	8	64

Table 6-1. Memory Operands

6.2.1 Bit and Byte Ordering

Byte ordering defines how the bytes those make up halfwords, singlewords and doublewords, are ordered in memory. To simplify OpenRISC implementations, architecture specifies as default byte ordering the most significant byte (MSB) ordering, or big endian as it is sometimes called. But implementation can support least significant byte (LSB) ordering if they implement byte reording 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 data and floating-point data, where the most significant byte of a floating-point value holds the sign and at least the start of the exponent.

Table 6-2 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 6-2. Default Bit and Byte Ordering in Halfwords

Table 6-3 shows how bits and bytes are ordered in a singleword.

Bit 31 Bit 24			Bit 7 Bit 0
MSB			LSB
Byte address 0	Byte address 1	Byte address 2	Byte address 3

Table 6-3. Default Bit and Byte Ordering in Singlewords and Single Precision Floats

Table 6-4 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 6-4. Default Bit and Byte Ordering in Doublewords, Double Precision Floats and all Vector Types

6.2.2 Alignment and Misaligned Accesses

A memory operand is naturally aligned if its address is integral multiple of the operand length. Implementation might support accessing unaligned memory operands but default behavioral is that accesses to unaligned operands result in alignment exception. See chapter “Exception Model” on page 271 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 6-5. Memory Operand Alignment

OR32 instructions are four bytes long and word-aligned.

7 Register Set

7.1 Features

OpenRISC 1000 register set includes the following principal features:

- Thirtytwo 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 possible thirty-two registers.
- Thirty-two 64-bit vector/floating-point/DSP registers.
- All other registers are special-purpose registers defined for each unit separately and accessible through l.mtspr/l.mfspr instruction pair

7.2 Overview

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

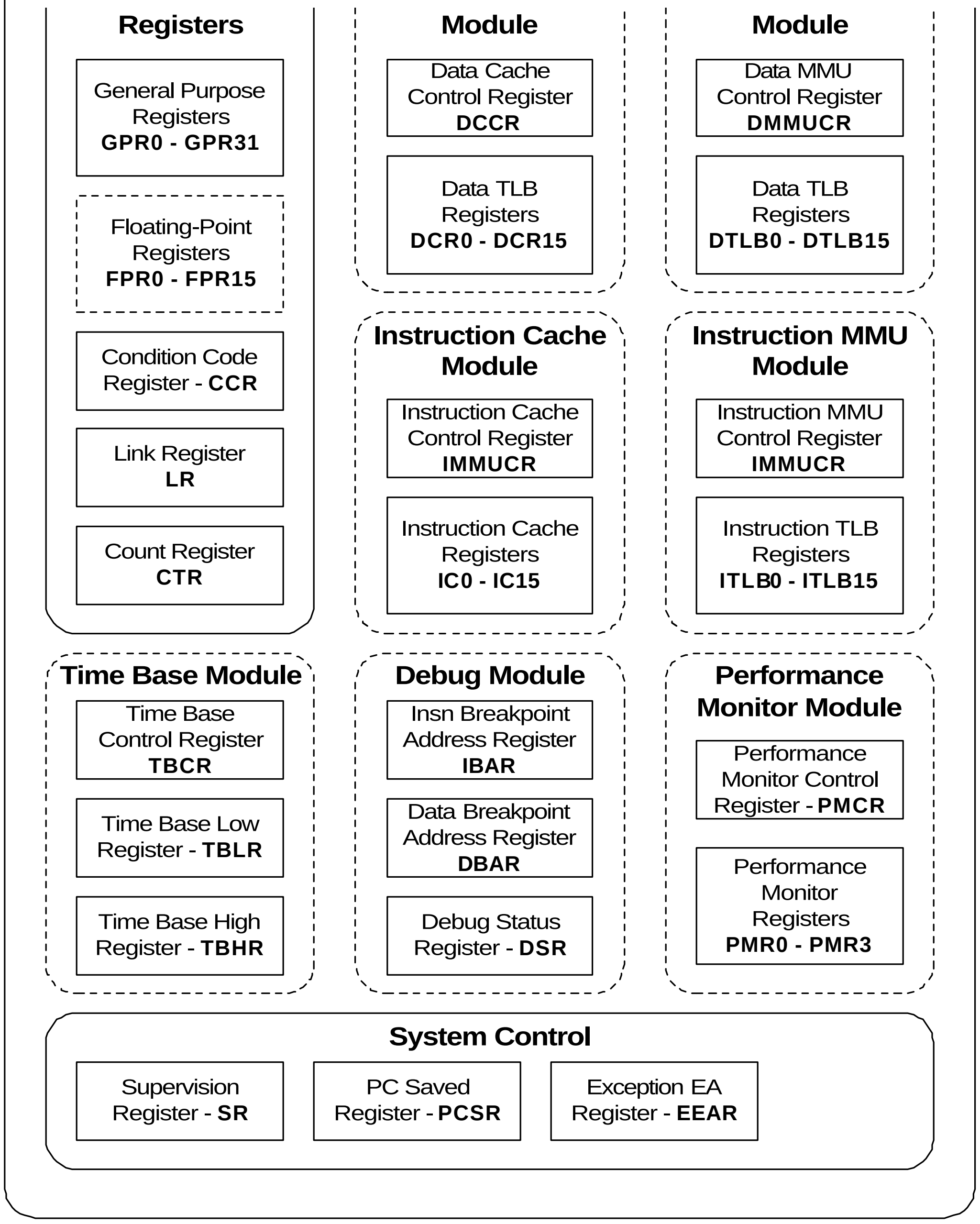
General-purpose and special-purpose user-level registers are accessible both in user mode and supervisor mode of operation. System control registers are accessible only in supervisor mode of operation (SR[SUPV]=1).

Unit dependent registers are usually accessible only in supervisor mode but not necessarily. Accessibility for architecture-defined units is defined in this manual. Accessibility for custom units not covered by this manual, is defined in implementation specific manuals.

7.3 Special-Purpose Registers

Special-purpose registers of all modules are grouped into thirtytwo groups. Each group can have different register address decoding depending on a maximum theorethical number of registers in that particular group. One group can contain registers from several different modules. In register address decoding it is also used SR[SUPV] bit since some registers are accessible only in supervisor mode. Instructions for reading and writing registers are l.mtspr and l.mfspr.

Figure 7-1. OpenRISC 1000 Programming Model – Registers (needs update !)


Registers
General Purpose Registers GPR0 - GPR31
Floating-Point Registers FPR0 - FPR15
Condition Code Register - CCR
Link Register LR
Count Register CTR
Module
Data Cache Control Register DCCR
Data TLB Registers DCR0 - DCR15
Module
Data MMU Control Register DMMUCR
Data TLB Registers DTLB0 - DTLB15
Instruction Cache Module
Instruction Cache Control Register IMMUCR
Instruction Cache Registers IC0 - IC15
Instruction MMU Module
Instruction MMU Control Register IMMUCR
Instruction TLB Registers ITLB0 - ITLB15
Time Base Module
Time Base Control Register TBCR
Time Base Low Register - TBLR
Time Base High Register - TBHR
Debug Module
Insn Breakpoint Address Register IBAR
Data Breakpoint Address Register DBAR
Debug Status Register - DSR
Performance Monitor Module
Performance Monitor Control Register - PMCR
Performance Monitor Registers PMR0 - PMR3
System Control
Supervision Register - SR
PC Saved Register - PCSR
Exception EA Register - EEAR

GROUP #	UNIT DESCRIPTION
0	System Control and Status Registers
1	Data MMU (in case of a single unified MMU groups 1 and 2 decode in a single set of registers)
2	Instruction MMU (in case of a single unified MMU groups 1 and 2 decode in a single set of registers)
3	Data Cache (in case of a single unified cache groups 3 and 4 decode in a single set of registers)
4	Instruction Cache (in case of a single unified cache groups 3 and 4 decode in 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
11-23	Reserved for future use
24-31	Custom units

Table 7-1. Groups of SPRs

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

GRP #	REG #	REG NAME	USER MODE	SUPV MODE	DESCRIPTION
0	1	VR	-	Read Only	Version Register
0	2	UPR	-	Read Only	Unit Present Register
0	3	SR	-	R/W	Supervision Register
0	16-31	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
1	0-255	DTLBMR0-DTLBMR255	-	Write Only	Data TLB Match Registers
1	256-511	DTLBTR0-DTLBTR255	-	Write Only	Data TLB Translate Registers
1	512	DMMUCR	-	R/W	Data MMU Control Register
1	513	DMMUPR	-	R/W	Data MMU Protection Register
1	514	DTLBEIR	-	W	Data TLB Entry Invalidate

					Register
2	0-255	ITLBMR0-ITLBMR255	-	R/W	Instruction TLB Match Registers
2	256-511	ITLBTR0-ITLBTR255	-	R/W	Instruction TLB Translate Registers
2	512	IMMUCR	-	R/W	Instruction MMU Control Register
2	513	IMMUPR	-	R/W	Instruction MMU Protection Register
2	514	ITLBEIR	-	R/W	Instruction TLB Entry Invalidate Register
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	3	ICBIR	W	W	IC Block Invalidate Register
4	5	ICBLR	W	W	IC Block Lock Register
5	0	MACLO	R/W	R/W	MAC Low
5	1	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
6	22	DIR	-	R/W	Debug Instruction Register
7	0-7	PCCR0-PCCR7	R/W*	R/W	Performance Counters Count Registers
7	8-15	PCMR0-PCRM7	-	R/W	Performance Counters Mode Registers
8	0	PMR	-	R/W	Power Management Register
9	0	PICMR	-	R/W	PIC Mask Register
9	1	PICPR	-	R/W	PIC Priority Register
9	2	PICSR	-	R/W	PIC Status Register
10	0	TTCR	-	R/W	Tick Timer Control Register

10	1	TTIR	R/W	R/W	Tick Timer Incrementing Register

Table 7-2. List of All Special-Purpose Registers

7.4 General-Purpose Registers (GPRs)

The thirtytwo 32-bit general-purpose registers are labeled R0-R31. They hold integer data or memory pointers used by instructions. Table 7-3 contains a list of general-purpose registers and functions for which they are used. The GPRs are accessed as source and destination registers in the instruction syntax.

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 7-3. Lower and Upper Parts of General-Purpose Registers

R0 is used as a constant zero. Whether is R0 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 Interface”on page 331.

Implementation may have several sets of GPRs and use them as shadow registers, switching between them whenever a new exception occurs. Current set is identified with SR[CID] value.

Implementation is not required to initialize GPRs to zero during reset procedure. It is a responsibility of a reset exception handler to initialize GPRs to zero if that is necessary.

7.5 Special Sixteen GPRs Support

Programs can be compiled with upper sixteen registers disabled (set as *fixed* registers). Such programs are also executable on normal implementation with thirty-two registers but not vice versa. This feature is quite useful since users will move from less powerful OpenRISC implementations with sixteen registers to more powerful thirtytwo register OpenRISC implementations.

7.6 Vector/Floating-Point Registers (VFRs)

The thirtytwo vector/floating-point registers are 32 bits wide in 32-bit implementations that do not support double-precision floating-point arithmetic nor vector arithmetic. They are 64 bits wide in implementations that support either double-precision floating-point arithmetic or vector arithmetic. They are labeled VFR0–VFR31. Table 7-4 contains a list of these floating-point registers. The VFRs are accessed as source and destination registers in vector and floating-point instructions. See chapter “Application Binary Interface” on page 331 for information on floating-point data types.

REGISTER			VFR31	VFR30	VFR29	VFR28
REGISTER	VFR27	VFR26	VFR25	VFR24	VFR23	VFR22
REGISTER	VFR21	VFR20	VFR19	VFR18	VFR17	VFR16
REGISTER	VFR15	VFR15	VFR15	VFR14	VFR13	VFR12
REGISTER	VFR11	VFR10	VFR9	VFR8	VFR7	VFR6
REGISTER	VFR5	VFR4	VFR3	VFR2	VFR1	VFR0

Table 7-4. Floating-Point Registers

7.6.1 Condition Code Register (CCR0-CCR15)

Condition code register is a 32-bit special-purpose user-level register accessible with l.mtspr/l.mfspr instruction pair.

Flag named FLAG is set by sfXX instructions as a result of a compare operation. Flag named CY is set by arithmetic operations as a result of a carry out and used with addic instruction. Flag named OVERFL is set by arithmetic operations when overflow occurs.

BIT	31-3	2	1	0
Identifier	Reserved	OVERFL	CARRY	FLAG
Reset	0	0	0	0
R/W	Read Only	R/W	R/W	R/W

FLAG	Conditional branch flag 0 FLAG flag was cleared by sfXX instructions 1 FLAG flag was set by sfXX instructions
CY	Carry flag 0 No carry out produced by last arithmetic operation 1 Carry out was produced by last arithmetic operation
OVERFL	Overflow flag

	0 No overflow occured during last arithmetic operation 1 Overflow occured during last arithmetic operation (might even result in a overflow exception)

Table 7-5. CCR Field Descriptions

7.7 Supervision Register (SR)

The supervison register is a 32-bit special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode.

It defines the state of the processor.

BIT	31-28	27-12	11	10	9	8	7
Identifier	CID	Reserved	OV	CY	F	CE	LEE
Reset	0	0	0	0	0	0	0
R/W	R/W	Read Only	R/W	R/W	R/W	R/W	R/W

BIT	6	5	4	3	2	1	0
Identifier	IME	DME	ICE	DCE	EIR	EXR	SUPV
Reset	0	0	0	0	0	0	1
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

SUPV	Supervisor Mode 0 Processor is in User Mode 1 Processor is in Supervisor Mode
EXR	Exception Recognition 0 Exceptions are not recognized 1 Exceptions are recognized
EIR	External Interrupt Recognition 0 External Interrupts are not recognized 1 External Interrupts are recognized
DCE	Data Cache Enable 0 Data Cache is not enabled 1 Data Cache is enabled
ICE	Instruction Cache Enable 0 Instruction Cache is not enabled 1 Instruction Cache is enabled
DME	Data MMU Enable 0 Data MMU is not enabled 1 Data MMU is enabled
IME	Instruction MMU Enable 0 Instruction MMU is not enabled

	1 Instruction MMU is enabled
LEE	Little Endian Enable 0 Little Endian (LSB) byte ordering is not enabled 1 Little Endian (LSB) byte ordering is enabled
CE	CID Enable 0 CID automatic increment and shadow registers disabled 1 CID automatic increment and shadow registers enabled
F	Flag 0 Conditional branch flag was cleared by sfXX instructions 1 Conditional branch flag was set by sfXX instructions
CY	Carry flag 0 No carry out produced by last arithmetic operation 1 Carry out was produced by last arithmetic operation
OV	Overflow flag 0 No overflow occured during last arithmetic operation 1 Overflow occured during last arithmetic operation (might even result in a range exception)
CID	Context ID 0-15 Current Processor Context

Table 7-6. SR Field Descriptions

7.8 Exception Program Counter Registers (EPCR0 - EPCR15)

The exception program counter registers are special-purpose supervisor-level registers accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. They are 32 bits wide registers in 32-bit implementations and can be wider than 32 bits in 64-bit implementations.

After an exception 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 SR.

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

EPC	Exception Program Counter Address

Table 7-7. EPCR Field Descriptions

7.9 Exception Effective Address Registers (EEAR0-EEAR15)

The exception effective address registers are special-purpose supervisor-level registers accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. They are 32 bits wide registers in 32-bit implementations and can be wider than 32 bits in 64-bit implementations.

After an exception 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 SR.

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

EEA	Exception Effective Address

Table 7-8. EEAR Field Descriptions

7.10 Exception Supervision Registers (ESR0-ESR15)

The exception supervision registers are special-purpose supervisor-level registers accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. They are 32 bits wide registers in 32-bit implementations and can be wider than 32 bits in 64-bit implementations.

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

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

EEA	Exception SR

Table 7-9. ESR Field Descriptions

8 Instruction Set

This chapter describes OpenRISC 1000 instruction set.

8.1 Features

OpenRISC 1000 instruction set includes the following principal features:

- Simple and uniform-length instruction formats featuring three Instruction Subsets
- OpenRISC Basic Instruction Set (ORBIS32/64) with 32 bits wide instructions aligned on 32-bit boundaries in memory and operating on 32 bits and 64 bits data
- OpenRISC Vector/DSP eXtension (ORVDX64) with 32 bits wide instructions aligned on 32-bit boundaries in memory and operating on 8, 16, 32 and 64 bits data
- OpenRISC Floating-Point eXtension (ORFPX32/64) with 32 bits wide instructions aligned on 32-bit boundaries in memory and operating on 32 bits and 64 bits data
- Reserved opcodes for custom instructions
- Instructions divided into instruction classes where only the basic classes are required to be implemented in OpenRISC 1000 implementation

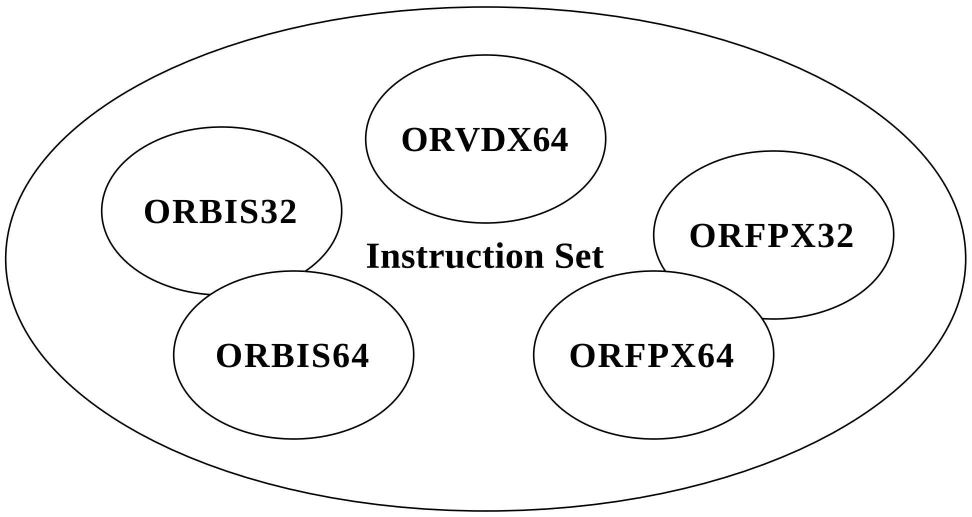


Figure 8-1. Instruction Set

8.2 Overview

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

- ORBIS32:
  - 32-bit integer 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:

- basic class
- advanced class

Class	Description
Basic	Instructions in basic class must always be implemented.
Advanced	Instructions from dvanced class are optional and implementation may choose to implement some or all instructions from this class based on requirements of the target application.

Table 8-1. OpenRISC 1000 Instruction Classes

8.3 ORBIS32/64

l.add Add Signed l.add

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

Format:

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

Description:

The contents of general-purpose register rA is 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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.addc Add Signed and Carry l.addc

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

Format:

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

Description:

The contents of general-purpose register rA is 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] <- 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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

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

Description:

Immediate is signed-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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.and And l.and

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

Format:

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

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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					16 bits															

Format:

```
l.andi rD,rA,K
```

Description:

Immediate is zero-extended and 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] <- rB[31:0] AND extz(Immediate)
```

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.bf Branch if Flag l.bf

31 26	25 0
opcode 0x4	N
6 bits	26 bits

Format:

```
l.bf N
```

Description:

The immediate is shifted left two bits, sign-extended to program counter width and then added to the address of the delay slot. The result is effective address of the branch. If the compare flag is set, then the program branches to EA with a delay of one instruction.

32-bit Implementation:

```
EA <- (Immediate || 00) + DelayInsnAddr
PC <- EA if flag set
```

64-bit Implementation:

```
EA <- (Immediate || 00) + DelayInsnAddr
PC <- EA if flag set
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.bnf Branch if No Flag l.bnf

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

Format:

```
l.bnf N
```

Description:

The immediate is shifted left two bits, sign-extended to program counter width and then added to the address of the delay slot. The result is effective address of the branch. If the compare flag is cleared, then the program branches to EA with a delay of one instruction.

32-bit Implementation:

```
EA <- (Immediate || 00) + DelayInsnAddr
PC <- EA if flag cleared
```

64-bit Implementation:

```
EA <- (Immediate || 00) + DelayInsnAddr
PC <- EA if flag cleared
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.brk Breakpoint l.brk

31 16	15 0
opcode 0x2100	K
16 bits	16 bits

Format:

```
l.brk K
```

Description:

Execution of the breakpoint instruction results in the breakpoint exception. Breakpoint exception is a request to the operating system and to the debug facility to execute certain debug services. Immediate is used by the debug to identify which breakpoint it is.

32-bit Implementation:

```
breakpoint-exception(K)
```

64-bit Implementation:

```
breakpoint-exception(K)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.csync Context Syncronization l.csync

31 0
opcode 0x23000000
32 bits

Format:

```
l.csync
```

Description:

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

32-bit Implementation:

```
context-synchronization
```

64-bit Implementation:

```
context-synchronization
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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

31 26	25 0
opcode 0x1c	reserved
6 bits	26 bits

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 instead by the implementation itself.

32-bit Implementation:

```
N/A
```

64-bit Implementation:

```
N/A
```

Exceptions:

```
N/A
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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

31	.	.	.	.	26	25	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0
opcode 0x1d						reserved																									
6 bits						26 bits																									

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 instead by the implementation itself.

32-bit Implementation:

```
N/A
```

64-bit Implementation:

```
N/A
```

Exceptions:

```
N/A
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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

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

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 instead by the implementation itself.

32-bit Implementation:

```
N/A
```

64-bit Implementation:

```
N/A
```

Exceptions:

```
N/A
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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

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

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 instead by the implementation itself.

32-bit Implementation:

```
N/A
```

64-bit Implementation:

```
N/A
```

Exceptions:

```
N/A
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.cust5 Reserved for ORBIS32/64 Custom Instructions l.cust5

31	.	.	.	.	26	25	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0
opcode 0x3c						reserved																									
6 bits						26 bits																									

Format:

```
l.cust5
```

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
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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

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

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 instead by the implementation itself.

32-bit Implementation:

```
N/A
```

64-bit Implementation:

```
N/A
```

Exceptions:

```
N/A
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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

31 26	25 0
opcode 0x3e	reserved
6 bits	26 bits

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 instead by the implementation itself.

32-bit Implementation:

```
N/A
```

64-bit Implementation:

```
N/A
```

Exceptions:

```
N/A
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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

31 26	25 0
opcode 0x3f	reserved
6 bits	26 bits

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 instead by the implementation itself.

32-bit Implementation:

```
N/A
```

64-bit Implementation:

```
N/A
```

Exceptions:

```
N/A
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.div Divide Signed l.div

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 6	5 4	3 . . 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x9
6 bits	5 bits	5 bits	5 bits	3 bits	2 bits	2 bits	4 bits

Format:

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

Description:

The contents of general-purpose register rA are divided by the contents of general-purpose register rB and the result is placed into general-purpose register rD. Both operands are treated as signed integers. A divide by zero flag is set when the divisor is zero.

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.divu Divide Unsigned l.divu

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

Format:

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

Description:

The contents of general-purpose register rA are divided by the contents of general-purpose register rA and the result is placed into general-purpose register rD. Both operands are treated as unsigned integers. A divide by zero flag is set when the divisor is zero.

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.extbs Extend Byte with Sign l.extbs

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

Format:

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

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 from 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
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.extbz Extend Byte with Zero l.extbz

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

Format:

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

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 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
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.exths Extend Half Word with Sign l.exths

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

Format:

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

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 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
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.exthz Extend Half Word with Zero l.exthz

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

Format:

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

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 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
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.extws Extend Word with Sign l.extws

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

Format:

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

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
```

Notes:

Instruction Class	Implementation
ORBIS64 II	Optional

l.extwz Extend Word with Zero l.extwz

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

Format:

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

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 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
```

Notes:

Instruction Class	Implementation
ORBIS64 II	Optional

l.j Jump l.j

31 26	25 0
opcode 0x0	N
6 bits	26 bits

Format:

```
l.j N
```

Description:

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

32-bit Implementation:

```
PC <- (Immediate || 00) + DelayInsnAddr
LR <- DelayInsnAddr + 4
```

64-bit Implementation:

```
PC <- (Immediate || 00) + DelayInsnAddr
LR <- DelayInsnAddr + 4
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.jal Jump and Link l.jal

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

Format:

```
l.jal N
```

Description:

The immediate is shifted left two bits, sign-extended to program counter width and then added to the address of the delay slot. The result is 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 <- (Immediate || 00) + DelayInsnAddr
LR <- DelayInsnAddr + 4
```

64-bit Implementation:

```
PC <- (Immediate || 00) + DelayInsnAddr
LR <- DelayInsnAddr + 4
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.jalr Jump and Link Register l.jalr

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

Format:

```
l.jalr rB
```

Description:

The contents of general-purpose register rB is 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 <- rB
LR <- DelayInsnAddr + 4
```

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.jr Jump Register l.jr

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

Format:

```
l.jr rB
```

Description:

The contents of general-purpose register rB is 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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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] <- rA[8]
```

64-bit Implementation:

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

Exceptions:

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

Instruction Class	Implementation
ORBIS32 I	Required

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					16 bits															

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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	Implementation
ORBIS32 I	Required

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	16 bits

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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					16 bits															

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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
```

Notes:

Instruction Class	Implementation
ORBIS64 I	Required

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	16 bits

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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] <- rA[15]
```

64-bit Implementation:

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

Exceptions:

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

Instruction Class	Implementation
ORBIS32 I	Required

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					16 bits															

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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
```

Instruction Class	Implementation
ORBIS32 I	Required

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					16 bits															

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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] <- rA[31]
```

Exceptions:

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

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.mac Multiply Signed and Accumulate l.mac

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 6	5 4	3 . . 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x7
6 bits	5 bits	5 bits	5 bits	3 bits	2 bits	2 bits	4 bits

Format:

```
l.mac 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 32 bits and added to the special-purpose registers MACHI and MACLO. All operands are treated as signed integers.

32-bit Implementation:

```
M[31:0] <- rA[31:0] * rB[31:0]
MACHI[31:0]MACLO[31:0] <- M[31:0] +
MACHI[31:0]MACLO[31:0]
SR[OV] <- overflow
```

64-bit Implementation:

```
M[31:0] <- rA[63:0] * rB[63:0]
MACHI[31:0]MACLO[31:0] <- M[31:0] +
MACHI[31:0]MACLO[31:0]
SR[OV] <- overflow
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORBIS32 II	Optional

l.mac Multiply Signed and Accumulate l.mac

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 6	5 4	3 . . 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x7
6 bits	5 bits	5 bits	5 bits	3 bits	2 bits	2 bits	4 bits

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.maci Multiply Immediate Signed and Accumulate l.maci

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

Format:

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

Description:

Immediate 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:

```
M[31:0] <- rA[31:0] * Immediate
MACHI[31:0]MACLO[31:0] <- M[31:0] +
MACHI[31:0]MACLO[31:0]
SR[OV] <- overflow
```

64-bit Implementation:

```
M[31:0] <- rA[63:0] * Immediate
MACHI[31:0]MACLO[31:0] <- M[31:0] +
MACHI[31:0]MACLO[31:0]
SR[OV] <- overflow
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORBIS32 II	Optional

l.maci Multiply Immediate Signed and Accumulate l.maci

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

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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

31 26	25 . . . 21	20 . . . 16	15 0
opcode 0x7	D	A	K
6 bits	5 bits	5 bits	16 bits

Format:

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

Description:

The contents of special register identified by the sum of general-purpose rA and zero-extended immediate are moved into general-purpose register rD.

32-bit Implementation:

```
rD[31:0] <- spr(rA+extz(Immediate))
```

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.movhi Move Immediate High l.movhi

31 26	25 . . . 21	20 . . . 16	15 0
opcode 0x6	D	reserved	K
6 bits	5 bits	5 bits	16 bits

Format:

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

Description:

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

32-bit Implementation:

```
rA[31:0] <- extz(Immediate) << 16
```

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.msync Memory Syncronization l.msync

31	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0
opcode 0x22000000																															
32 bits																															

Format:

```
l.msync
```

Description:

Execution of 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
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 0
opcode 0x10	K	A	B	K
6 bits	5 bits	5 bits	5 bits	11 bits

Format:

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

Description:

The contents of general-purpose register rB are moved into special register identified by the sum of general-purpose register rA and zero-extended immediate.

32-bit Implementation:

```
spr(rA+extz(Immediate)) <- rA[31:0]
```

64-bit Implementation:

```
spr(rA+extz(Immediate)) <- rA[31:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.mul Multiply Signed l.mul

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

Format:

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

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

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

Description:

Immediate 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
```

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.mulu Multiply Unsigned l.mulu

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

Format:

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

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.nop No Operation l.nop

31	.	.	.	.	.	.	24	23	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0
opcode 0x15								reserved																							
8 bits								24 bits																							

Format:

```
l.nop
```

Description:

This instruction does not do anything except it takes at least one clock cycle to complete. It is often used to fill delay slot gaps.

32-bit Implementation:

64-bit Implementation:

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.or Or l.or

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 6	5 4	3 . . 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x4
6 bits	5 bits	5 bits	5 bits	3 bits	2 bits	2 bits	4 bits

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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

```
l.ori rD,rA,K
```

Description:

Immediate is zero-extended and 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] <- rB[31:0] OR extz(Immediate)
```

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.psync Pipeline Syncronization l.psync

31	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0
opcode 0x22800000																															
32 bits																															

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
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.rfe Return From Exception l.rfe

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

Format:

```
l.rfe
```

Description:

Execution of this instruction restores the state of the processor prior to the exception.

32-bit Implementation:

```
state_restore()
```

64-bit Implementation:

```
state_restore()
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.ror Rotate Right l.ror

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

Format:

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

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

l.rori Rotate Right with Immediate l.rori

31 26	25 . . . 21	20 . . . 16	15 9	8 . 6	5 0
opcode 0x2e	D	A	reserved	opcode 0x4	L
6 bits	5 bits	5 bits	7 bits	3 bits	6 bits

Format:

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

Description:

6-bit immediate specifies the number of bit positions the contents of general-purpose register rA are rotated right. Result is written into general-purpose register rD.

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

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	11 bits

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	11 bits

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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
```

Notes:

Instruction Class	Implementation
ORBIS64 I	Required

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	11 bits

Format:

```
l.sfeq 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, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:

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

64-bit Implementation:

```
flag <- rA[63:0] == rB[63:0]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.sfeqi Set Flag if Equal Immediate l.sfeqi

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

Format:

```
l.sfeqi rA,I
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

```
flag <- rA[63:0] == rB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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	11 bits

Format:

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

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

```
l.sfgesi rA,I
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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	11 bits

Format:

```
l.sfgeu rA,rB
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

```
l.sfgeui rA,I
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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	11 bits

Format:

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

Description:

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

32-bit Implementation:

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

64-bit Implementation:

```
flag <- rA[63:0] > rB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

```
l.sfgtsi rA,I
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

```
flag <- rA[63:0] > rB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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	11 bits

Format:

```
l.sfgtu rA,rB
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

```
flag <- rA[63:0] > rB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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					16 bits															

Format:

```
l.sfgtui rA,I
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

```
flag <- rA[63:0] > rB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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	11 bits

Format:

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

Description:

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

32-bit Implementation:

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

64-bit Implementation:

```
flag <- rA[63:0] <= rB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

```
l.sflesi rA,I
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

```
flag <- rA[63:0] <= rB[63:0]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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

31 21	20 . . . 16	15 . . . 11	10 0
opcode 0x725	A	B	reserved
11 bits	5 bits	5 bits	11 bits

Format:

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

Description:

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

32-bit Implementation:

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

64-bit Implementation:

```
flag <- rA[63:0] <= rB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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					16 bits															

Format:

```
l.sfleui rA,I
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

```
flag <- rA[63:0] <= rB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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	11 bits

Format:

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

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

```
l.sfltsi rA,I
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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	11 bits

Format:

```
l.sfltu rA,rB
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

```
l.sfltui rA,I
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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	11 bits

Format:

```
l.sfne 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, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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

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

Format:

```
l.sfnei rA,I
```

Description:

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

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 II	Optional

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	11 bits

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.sll Shift Left Logical l.sll

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	.	0
opcode 0x38						D					A					B					reserved			opcode 0x8							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

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. Result is written into general-purpose rD.

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.slli Shift Left Logical with Immediate l.slli

31 26	25 . . . 21	20 . . . 16	15 9	8 . 6	5 0
opcode 0x2e	D	A	reserved	opcode 0x0	L
6 bits	5 bits	5 bits	7 bits	3 bits	6 bits

Format:

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

Description:

6-bit immediate specifies the number of bit positions the contents of general-purpose register rA are shifted left, inserting zeros into the low-order bits. Result is written into general-purpose rD.

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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.sra Shift Right Arithmetic l.sra

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0x38	D	A	B	reserved	opcode 0x28
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

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. Result is written into general-purpose register rD.

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

None

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.srai Shift Right Arithmetic with Immediate l.srai

31 26	25 . . . 21	20 . . . 16	15 9	8 . 6	5 0
opcode 0x2e	D	A	reserved	opcode 0x2	L
6 bits	5 bits	5 bits	7 bits	3 bits	6 bits

Format:

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

Description:

6-bit immediate specifies the number of bit positions the contents of general-purpose register rA are shifted right, sign-extending the high-order bits. Result is written into general-purpose register rD.

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

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.srl Shift Right Logical l.srl

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0x38	D	A	B	reserved	opcode 0x18
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

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. Result is written into general-purpose register rD.

32-bit Implementation:

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

64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.srli Shift Right Logical with Immediate l.srli

31 26	25 . . . 21	20 . . . 16	15 9	8 . 6	5 0
opcode 0x2e	D	A	reserved	opcode 0x1	L
6 bits	5 bits	5 bits	7 bits	3 bits	6 bits

Format:

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

Description:

6-bit Immediate specifies the number of bit positions the contents of general-purpose register rA are shifted right, inserting zeros into the high-order bits. Result is written into general-purpose register rD.

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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.sub Subtract Signed l.sub

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

Format:

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

Description:

The contents of general-purpose register rB is 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]
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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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					11 bits										

Format:

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

Description:

Offset is sign-extended and added to the contents of general-purpose register rA. Sum represents 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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.sys System Call l.sys

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

Format:

```
l.sys K
```

Description:

Execution of system call instruction results in the system call exception. System calls exception is a request to the operating system to provide operating system services. Immediate specifies which system service is required.

32-bit Implementation:

```
system-call-exception(K)
```

64-bit Implementation:

```
system-call-exception(K)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

l.xor Exclusive Or l.xor

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 6	5 4	3 . . 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x5
6 bits	5 bits	5 bits	5 bits	3 bits	2 bits	2 bits	4 bits

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
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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	16 bits

Format:

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

Description:

Immediate is zero-extended and 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] <- rB[31:0] XOR exts(Immediate)
```

64-bit Implementation:

```
rD[63:0] <- rB[63:0] XOR exts(Immediate)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORBIS32 I	Required

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

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

Format:

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

Description:

The contents of vector/floating-point register vfrA is added to the contents of vector/floating-point register vfrB to form the result. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] + vfrB[63:0]
```

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

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

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

Format:

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

Description:

The contents of vector/floating-point register vfrA is added to the contents of vector/floating-point register vfrB to form the result. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

```
vfrD[31:0] <- vfrA[31:0] + vfrB[31:0]
```

64-bit Implementation:

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

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

31 26	25 8	7 . . 4	3 . . 0
opcode 0xc	reserved	opcode 0xe	reserved
6 bits	18 bits	4 bits	4 bits

Format:

```
lf.cust1.d
```

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
```

Notes:

Instruction Class	Implementation
ORFPX64 II	Optional

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

31 26	25 8	7 . . 4	3 . . 0
opcode 0xb	reserved	opcode 0xe	reserved
6 bits	18 bits	4 bits	4 bits

Format:

```
lf.cust1.s
```

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
```

Notes:

Instruction Class	Implementation
ORFPX32 II	Optional

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

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xc	D	A	B	reserved	opcode 0x13
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

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

Description:

The contents of vector/floating-point register vfrA is divided by the contents of vector/floating-point register vfrB to form the result. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] / vfrB[63:0]
```

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX64 II	Optional

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

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xb	D	A	B	reserved	opcode 0x13
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

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

Description:

The contents of vector/floating-point register vfrA is divided by the contents of vector/floating-point register vfrB to form the result. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

```
vfrD[31:0] <- vfrA[31:0] / vfrB[31:0]
```

64-bit Implementation:

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX32 II	Optional

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 0xc	D	A	B	reserved	opcode 0x15
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

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

Description:

The contents of vector/floating-point register vfrA are converted to integer and stored into general-purpose register rD.

32-bit Implementation:
64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

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 0xb	D	A	B	reserved	opcode 0x15
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.ftoi.s rD,rA
```

Description:

The contents of vector/floating-point register vfrA are converted to integer and stored into general-purpose register rD.

32-bit Implementation:

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

64-bit Implementation:

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

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 0xc	D	A	B	reserved	opcode 0x14
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

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

Description:

The contents of general-purpose register rA are converted to Double-precision floating-point number and stored into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

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

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

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 0xb						D					A					B					reserved			opcode 0x14							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

Format:

```
lf.itof.s rD,rA
```

Description:

The contents of general-purpose register rA are converted to single-precision floating-point number and stored into vector/floating-point register vfrD.

32-bit Implementation:

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

64-bit Implementation:

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

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 0xc	D	A	B	reserved	opcode 0x17
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

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

Description:

The contents of vector/floating-point register vfrA is multiplied by the contents of vector/floating-point register vfrB and added to special-purpose register FPMADDLO/FPMADDHI.

32-bit Implementation:

64-bit Implementation:

```
FPMADDHI[31:0]FPMADDLO[31:0] <- vfrA[63:0] *
vfrB[63:0] + FPMADDHI[31:0]FPMADDLO[31:0]
```

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX64 II	Optional

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 0xb						D					A					B					reserved			opcode 0x17							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

Format:

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

Description:

The contents of vector/floating-point register vfrA is multiplied by the contents of vector/floating-point register vfrB and added to special-purpose register FPMADDLO/FPMADDHI.

32-bit Implementation:

```
FPMADDHI[31:0]FPMADDLO[31:0] <- vfrA[31:0] *
vfrB[31:0] + FPMADDHI[31:0]FPMADDLO[31:0]
```

64-bit Implementation:

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX32 II	Optional

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

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

Format:

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

Description:

The contents of vector/floating-point register vfrA is multiplied by the contents of vector/floating-point register vfrB to form the result. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] * vfrB[63:0]
```

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

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

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

Format:

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

Description:

The contents of vector/floating-point register vfrA is multiplied by the contents of vector/floating-point register vfrB to form the result. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

```
vfrD[31:0] <- vfrA[31:0] * vfrB[31:0]
```

64-bit Implementation:

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

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

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

Format:

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

Description:

The contents of vector/floating-point register vfrA is divided by the contents of vector/floating-point register vfrB and remainder is used as the result. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] % vfrB[63:0]
```

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX64 II	Optional

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

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

Format:

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

Description:

The contents of vector/floating-point register vfrA is divided by the contents of vector/floating-point register vfrB and remainder is used as the result. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

```
vfrD[31:0] <- vfrA[31:0] % vfrB[31:0]
```

64-bit Implementation:

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX32 II	Optional

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 0xc	reserved	A	B	reserved	opcode 0x18
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sfeq.d rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If the two registers are equal, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[63:0] == vfrB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

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 0xb						reserved					A					B					reserved			opcode 0x18							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

Format:

```
lf.sfeq.s rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If the two registers are equal, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:

```
flag <- vfrA[31:0] == vfrB[31:0]
```

64-bit Implementation:

Exceptions:

None

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

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 0xc	reserved	A	B	reserved	opcode 0x1b
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sfge.d rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If first register is greater or equal than the second register, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[63:0] >= vfrB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

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 0xb	reserved	A	B	reserved	opcode 0x1b
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sfge.s rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If first register is greater or equal than the second register, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:

```
flag <- vfrA[31:0] >= vfrB[31:0]
```

64-bit Implementation:

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

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 0xc	reserved	A	B	reserved	opcode 0x1a
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sfgt.d rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If first register is greater than second register, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[63:0] > vfrB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

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 0xb	reserved	A	B	reserved	opcode 0x1a
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sfgt.s rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If first register is greater than second register, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:

```
flag <- vfrA[31:0] > vfrB[31:0]
```

64-bit Implementation:

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

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 0xc	reserved	A	B	reserved	opcode 0x1d
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sfle.d rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If first register is less or equal than the second register, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[363:0] <= vfrB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

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 0xb	reserved	A	B	reserved	opcode 0x1d
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sfle.s rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If first register is less or equal than the second register, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:

```
flag <- vfrA[31:0] <= vfrB[31:0]
```

64-bit Implementation:

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

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 0xc	reserved	A	B	reserved	opcode 0x1c
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sflt.d rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If first register is less than second register, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[63:0] < vfrB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

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 0xb	reserved	A	B	reserved	opcode 0x1c
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sflt.s rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If first register is less than second register, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:

```
flag <- vfrA[31:0] < vfrB[31:0]
```

64-bit Implementation:

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

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 0xc	reserved	A	B	reserved	opcode 0x19
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sfne.d rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If the two registers are not equal, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[63:0] != vfrB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

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 0xb	reserved	A	B	reserved	opcode 0x19
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lf.sfne.s rA,rB
```

Description:

The contents of vector/floating-point register vfrA and the contents of vector/floating-point register vfrB are compared. If the two registers are not equal, then the compare flag is set; otherwise the compare flag is cleared.

32-bit Implementation:

```
flag <- vfrA[31:0] != vfrB[31:0]
```

64-bit Implementation:

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

lf.sub.d Subtract Floating-Point Double-Precision lf.sub.d

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

Format:

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

Description:

The contents of vector/floating-point register vfrB is subtracted from the contents of vector/floating-point register vfrA to form the result. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] - vfrB[63:0]
```

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

lf.sub.s Subtract Floating-Point Single-Precision lf.sub.s

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

Format:

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

Description:

The contents of vector/floating-point register vfrB is subtracted from the contents of vector/floating-point register vfrA to form the result. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

```
vfrD[31:0] <- vfrA[31:0] - vfrB[31:0]
```

64-bit Implementation:

Exceptions:

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

lvf.ld Load Vector/Floating-Point Double Word lvf.ld

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

Format:

```
lvf.ld rD,0(rA)
```

Description:

The contents of vector/floating-point register vfrA is used as effective address. The double word in memory addressed by EA is loaded into vector/floating-point register vfrD.

32-bit Implementation:

```
N/A
```

64-bit Implementation:

```
EA <- vfrA[63:0]
rD[63:0] <- (EA)[63:0]
```

Exceptions:

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

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

lvf.lw Load Vector/Floating-Point Single Word lvf.lw

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

Format:

```
lvf.lw rD,0(rA)
```

Description:

The contents of vector/floating-point register vfrA is used as effective address. The double word in memory addressed by EA is loaded into vector/floating-point register vfrD.

32-bit Implementation:

```
EA <- vfrA[31:0]
rD[31:0] <- (EA)[31:0]
```

64-bit Implementation:

```
EA <- vfrA[31:0]
rD[31:0] <- (EA)[31:0]
```

Exceptions:

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

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

lvf.sd Store Vector/Floating-Point Double Word lvf.sd

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

Format:

```
lvf.sd 0(rA),rB
```

Description:

The contents of vector/floating-point register vfrA is used as effective address. The double word in vector/floating-point register vrfB is stored to memory location addresses by EA.

32-bit Implementation:

```
N/A
```

64-bit Implementation:

```
EA <- vfrA[63:0]
rD[63:0] <- (EA)[63:0]
```

Exceptions:

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

Notes:

Instruction Class	Implementation
ORFPX64 I	Required

lvf.sw Store Vector/Floating-Point Single Word lvf.sw

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

Format:

```
lvf.sw 0(rA),rB
```

Description:

The contents of vector/floating-point register vfrA is used as effective address. The single word in vector/floating-point register vrfB is stored to memory location addresses by EA.

32-bit Implementation:

```
EA <- vfrA[31:0]
rD[31:0] <- (EA)[31:0]
```

64-bit Implementation:

```
EA <- vfrA[31:0]
rD[31:0] <- (EA)[31:0]
```

Exceptions:

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

Notes:

Instruction Class	Implementation
ORFPX32 I	Required

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	8 bits

Format:

```
lv.add.b rD,rA,rB
```

Description:

The byte elements of vector/floating-point register vfrA are added to the byte elements of vector/floating-point register vfrB to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[7:0] <- vfrA[7:0] + vfrB[7:0]
vfrD[15:8] <- vfrA[15:8] + vfrB[15:8]
vfrD[23:16] <- vfrA[23:16] + vfrB[23:16]
vfrD[31:24] <- vfrA[31:24] + vfrB[31:24]
vfrD[39:32] <- vfrA[39:32] + vfrB[39:32]
vfrD[47:40] <- vfrA[47:40] + vfrB[47:40]
vfrD[55:48] <- vfrA[55:48] + vfrB[55:48]
vfrD[63:56] <- vfrA[63:56] + vfrB[63:56]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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			8 bits							

Format:

```
lv.add.h rD,rA,rB
```

Description:

The half-word elements of vector/floating-point register vfrA are added to the half-word elements of vector/floating-point register vfrB to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrA[15:0] + vfrB[15:0]
vfrD[31:16] <- vfrA[31:16] + vfrB[31:16]
vfrD[47:32] <- vfrA[47:32] + vfrB[47:32]
vfrD[63:48] <- vfrA[63:48] + vfrB[63:48]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.adds.b rD,rA,rB
```

Description:

The byte elements of vector/floating-point register vfrA are added to the byte elements of vector/floating-point register vfrB to form the result elements. If the result exceeds min/max value for the destination data type, it is saturated to min/max value and placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- sat8s(vfrA[7:0] + vfrB[7:0])
vfrD[15:8] <- sat8s(vfrA[15:8] + vfrB[15:8])
vfrD[23:16] <- sat8s(vfrA[23:16] + vfrB[23:16])
vfrD[31:24] <- sat8s(vfrA[31:24] + vfrB[31:24])
vfrD[39:32] <- sat8s(vfrA[39:32] + vfrB[39:32])
vfrD[47:40] <- sat8s(vfrA[47:40] + vfrB[47:40])
vfrD[55:48] <- sat8s(vfrA[55:48] + vfrB[55:48])
vfrD[63:56] <- sat8s(vfrA[63:56] + vfrB[63:56])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.adds.h Vector Half-Word Elements Add Signed Saturated lv.adds.h

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x33
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.adds.h rD,rA,rB
```

Description:

The half-word elements of vector/floating-point register vfrA are added to the half-word elements of vector/floating-point register vfrB to form the result elements. If the result exceeds min/max value for the destination data type, it is saturated to min/max value and placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- sat16s(vfrA[15:0] + vfrB[15:0])
vfrD[31:16] <- sat16s(vfrA[31:16] + vfrB[31:16])
vfrD[47:32] <- sat16s(vfrA[47:32] + vfrB[47:32])
vfrD[63:48] <- sat16s(vfrA[63:48] + vfrB[63:48])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.addu.b rD,rA,rB
```

Description:

The unsigned byte elements of vector/floating-point register vfrA are added to the unsigned byte elements of vector/floating-point register vfrB to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- vfrA[7:0] + vfrB[7:0]
vfrD[15:8] <- vfrA[15:8] + vfrB[15:8]
vfrD[23:16] <- vfrA[23:16] + vfrB[23:16]
vfrD[31:24] <- vfrA[31:24] + vfrB[31:24]
vfrD[39:32] <- vfrA[39:32] + vfrB[39:32]
vfrD[47:40] <- vfrA[47:40] + vfrB[47:40]
vfrD[55:48] <- vfrA[55:48] + vfrB[55:48]
vfrD[63:56] <- vfrA[63:56] + vfrB[63:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.addu.h rD,rA,rB
```

Description:

The unsigned half-word elements of vector/floating-point register vfrA are added to the unsigned half-word elements of vector/floating-point register vfrB to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrA[15:0] + vfrB[15:0]
vfrD[31:16] <- vfrA[31:16] + vfrB[31:16]
vfrD[47:32] <- vfrA[47:32] + vfrB[47:32]
vfrD[63:48] <- vfrA[63:48] + vfrB[63:48]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.addus.b rD,rA,rB
```

Description:

The unsigned byte elements of vector/floating-point register vfrA are added to the unsigned byte elements of vector/floating-point register vfrB to form the result elements. If the result exceeds min/max value for the destination data type, it is saturated to min/max value and placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- sat8u(vfrA[7:0] + vfrB[7:0])
vfrD[15:8] <- sat8u(vfrA[15:8] + vfrB[15:8])
vfrD[23:16] <- sat8u(vfrA[23:16] + vfrB[23:16])
vfrD[31:24] <- sat8u(vfrA[31:24] + vfrB[31:24])
vfrD[39:32] <- sat8u(vfrA[39:32] + vfrB[39:32])
vfrD[47:40] <- sat8u(vfrA[47:40] + vfrB[47:40])
vfrD[55:48] <- sat8u(vfrA[55:48] + vfrB[55:48])
vfrD[63:56] <- sat8u(vfrA[63:56] + vfrB[63:56])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.addus.h Vector Half-Word Elements Add Unsigned Saturated lv.addus.h

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x37
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.addus.h rD,rA,rB
```

Description:

The unsigned half-word elements of vector/floating-point register vfrA are added to the unsigned half-word elements of vector/floating-point register vfrB to form the result elements. If the result exceeds min/max value for the destination data type, it is saturated to min/max value and placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- sat16s(vfrA[15:0] + vfrB[15:0])
vfrD[31:16] <- sat16s(vfrA[31:16] + vfrB[31:16])
vfrD[47:32] <- sat16s(vfrA[47:32] + vfrB[47:32])
vfrD[63:48] <- sat16s(vfrA[63:48] + vfrB[63:48])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.all_eq.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if all corresponding elements are equal; otherwise compare flag is cleared. Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[7:0] == vfrB[7:0]
vfrA[15:8] == vfrB[15:8]
vfrA[23:16] == vfrB[23:16]
vfrA[31:24] == vfrB[31:24]
vfrA[39:32] == vfrB[39:32]
vfrA[47:40] == vfrB[47:40]
vfrA[55:48] == vfrB[55:48]
vfrA[63:56] == vfrB[63:56]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.all_eq.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if all corresponding elements are equal; otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
flag <- vfrA[15:0] == vfrB[15:0]
vfrA[31:16] == vfrB[31:16]
vfrA[47:32] == vfrB[47:32]
vfrA[63:48] == vfrB[63:48]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.all_ge.b Vector Byte Elements All Greater or Equal Than 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	8 bits

Format:

```
lv.all_ge.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if all elements of vfrA are greater or equal than elements of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[7:0] >= vfrB[7:0]
vfrA[15:8] >= vfrB[15:8]
vfrA[23:16] >= vfrB[23:16]
vfrA[31:24] >= vfrB[31:24]
vfrA[39:32] >= vfrB[39:32]
vfrA[47:40] >= vfrB[47:40]
vfrA[55:48] >= vfrB[55:48]
vfrA[63:56] >= vfrB[63:56]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.all_ge.b Vector Byte Elements All Greater or Equal Than 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			8 bits							

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.all_ge.h Vector Half-Word Elements All Greater or Equal Than lv.all_ge.h

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x13
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.all_ge.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if all elements of vfrA are greater or equal than elements of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] >= vfrB[15:0]
vfrA[31:16] >= vfrB[31:16]
vfrA[47:32] >= vfrB[47:32]
vfrA[63:48] >= vfrB[63:48]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.all_gt.b Vector Byte Elements All Greater Than lv.all_gt.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x14
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.all_gt.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if all elements of vfrA are greater than elements of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[7:0] > vfrB[7:0]
vfrA[15:8] > vfrB[15:8]
vfrA[23:16] > vfrB[23:16]
vfrA[31:24] > vfrB[31:24]
vfrA[39:32] > vfrB[39:32]
vfrA[47:40] > vfrB[47:40]
vfrA[55:48] > vfrB[55:48]
vfrA[63:56] > vfrB[63:56]vfrD[63:0] <- repl(flag)
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.all_gt.b Vector Byte Elements All Greater Than lv.all_gt.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x14
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.all_gt.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if all elements of vfrA are greater than elements of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] > vfrB[15:0]
vfrA[31:16] > vfrB[31:16]
vfrA[47:32] > vfrB[47:32]
vfrA[63:48] > vfrB[63:48]vfrD[63:0] <- repl(flag)
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.all_le.b Vector Byte Elements All Less or Equal Than lv.all_le.b

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

Format:

```
lv.all_le.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if all elements of vfrA are less or equal than elements of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[7:0] <= vfrB[7:0]
vfrA[15:8] <= vfrB[15:8]
vfrA[23:16] <= vfrB[23:16]
vfrA[31:24] <= vfrB[31:24]
vfrA[39:32] <= vfrB[39:32]
vfrA[47:40] <= vfrB[47:40]
vfrA[55:48] <= vfrB[55:48]
vfrA[63:56] <= vfrB[63:56]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.all_le.b Vector Byte Elements All Less or Equal Than lv.all_le.b

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

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.all_le.h Vector Half-Word Elements All Less or Equal Than lv.all_le.h

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

Format:

```
lv.all_le.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if all elements of vfrA are less or equal than elements of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] ,= vfrB[15:0]
vfrA[31:16] <= vfrB[31:16]
vfrA[47:32] <= vfrB[47:32]
vfrA[63:48] <= vfrB[63:48]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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			8 bits							

Format:

```
lv.all_lt.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if all elements of vfrA are less than elements of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[7:0] < vfrB[7:0]
vfrA[15:8] < vfrB[15:8]
vfrA[23:16] < vfrB[23:16]
vfrA[31:24] < vfrB[31:24]
vfrA[39:32] < vfrB[39:32]
vfrA[47:40] < vfrB[47:40]
vfrA[55:48] < vfrB[55:48]
vfrA[63:56] < vfrB[63:56]vfrD[63:0] <- repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.all_lt.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if all elements of vfrA are less than elements of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] < vfrB[15:0]
vfrA[31:16] < vfrB[31:16]
vfrA[47:32] < vfrB[47:32]
vfrA[63:48] < vfrB[63:48]vfrD[63:0] <- repl(flag)
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.all_ne.b Vector Byte Elements All Not Equal lv.all_ne.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x1a
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.all_ne.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if all corresponding elements are not equal; otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[7:0] != vfrB[7:0]
vfrA[15:8] != vfrB[15:8]
vfrA[23:16] != vfrB[23:16]
vfrA[31:24] != vfrB[31:24]
vfrA[39:32] != vfrB[39:32]
vfrA[47:40] != vfrB[47:40]
vfrA[55:48] != vfrB[55:48]
vfrA[63:56] != vfrB[63:56]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.all_ne.b Vector Byte Elements All Not Equal lv.all_ne.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x1a
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.all_ne.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if all corresponding elements are not equal; otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] != vfrB[15:0]
vfrA[31:16] != vfrB[31:16]
vfrA[47:32] != vfrB[47:32]
vfrA[63:48] != vfrB[63:48]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.and rD,rA,rB
```

Description:

The contents of vector/floating-point register vfrA are combined with the contents of vector/floating-point register vfrB in a bit-wise logical AND operation. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] AND vfrB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_eq.b Vector Byte Elements Any Equal lv.any_eq.b

31 26	25 21	20 16	15 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x20
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.any_eq.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if any two corresponding elements are equal; otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[7:0] == vfrB[7:0] ||
vfrA[15:8] == vfrB[15:8] ||
vfrA[23:16] == vfrB[23:16] ||
vfrA[31:24] == vfrB[31:24] ||
vfrA[39:32] == vfrB[39:32] ||
vfrA[47:40] == vfrB[47:40] ||
vfrA[55:48] == vfrB[55:48] ||
vfrA[63:56] == vfrB[63:56]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_eq.b Vector Byte Elements Any Equal lv.any_eq.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x20
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.any_eq.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if any two corresponding elements are equal; otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] == vfrB[15:0] ||
vfrA[31:16] == vfrB[31:16] ||
vfrA[47:32] == vfrB[47:32] ||
vfrA[63:48] == vfrB[63:48]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_ge.b Vector Byte Elements Any Greater or Equal Than 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	8 bits

Format:

```
lv.any_ge.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if any element of vfrA is greater or equal than corresponding element of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
flag <- vfrA[7:0] >= vfrB[7:0] ||
vfrA[15:8] >= vfrB[15:8] ||
vfrA[23:16] >= vfrB[23:16] ||
vfrA[31:24] >= vfrB[31:24] ||
vfrA[39:32] >= vfrB[39:32] ||
vfrA[47:40] >= vfrB[47:40] ||
vfrA[55:48] >= vfrB[55:48] ||
vfrA[63:56] >= vfrB[63:56]vfrD[63:0] <-
repl(flag)
```

Exceptions:

None

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_ge.b Vector Byte Elements Any Greater or Equal Than 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	8 bits

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_ge.h Vector Half-Word Elements Any Greater or Equal Than lv.any_ge.h

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x23
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.any_ge.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if any element of vfrA is greater or equal than corresponding element of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] >= vfrB[15:0] ||
vfrA[31:16] >= vfrB[31:16] ||
vfrA[47:32] >= vfrB[47:32] ||
vfrA[63:48] >= vfrB[63:48]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_gt.b Vector Byte Elements Any Greater Than lv.any_gt.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x24
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.any_gt.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if any element of vfrA is greater than corresponding element of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[7:0] > vfrB[7:0] ||
vfrA[15:8] > vfrB[15:8] ||
vfrA[23:16] > vfrB[23:16] ||
vfrA[31:24] > vfrB[31:24] ||
vfrA[39:32] > vfrB[39:32] ||
vfrA[47:40] > vfrB[47:40] ||
vfrA[55:48] > vfrB[55:48] ||
vfrA[63:56] > vfrB[63:56]vfrD[63:0] <- repl(flag)
```

Exceptions:

None

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_gt.b Vector Byte Elements Any Greater Than lv.any_gt.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x24
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.any_gt.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if any element of vfrA is greater than corresponding element of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] > vfrB[15:0] ||
vfrA[31:16] > vfrB[31:16] ||
vfrA[47:32] > vfrB[47:32] ||
vfrA[63:48] > vfrB[63:48]vfrD[63:0] <- repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_le.b Vector Byte Elements Any Less or Equal Than lv.any_le.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x26
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.any_le.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if any element of vfrA is less or equal than corresponding element of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
flag <- vfrA[7:0] <= vfrB[7:0] ||
vfrA[15:8] <= vfrB[15:8] ||
vfrA[23:16] <= vfrB[23:16] ||
vfrA[31:24] <= vfrB[31:24] ||
vfrA[39:32] <= vfrB[39:32] ||
vfrA[47:40] <= vfrB[47:40] ||
vfrA[55:48] <= vfrB[55:48] ||
vfrA[63:56] <= vfrB[63:56]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_le.b Vector Byte Elements Any Less or Equal Than lv.any_le.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x26
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_le.h Vector Half-Word Elements Any Less or Equal Than 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	8 bits

Format:

```
lv.any_le.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if any element of vfrA is less or equal than corresponding element of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] ,= vfrB[15:0] ||
vfrA[31:16] <= vfrB[31:16] ||
vfrA[47:32] <= vfrB[47:32] ||
vfrA[63:48] <= vfrB[63:48]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_lt.b Vector Byte Elements Any Less Than lv.any_lt.b

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

Format:

```
lv.any_lt.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if any element of vfrA is less than corresponding element of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[7:0] < vfrB[7:0] ||
vfrA[15:8] < vfrB[15:8] ||
vfrA[23:16] < vfrB[23:16] ||
vfrA[31:24] < vfrB[31:24] ||
vfrA[39:32] < vfrB[39:32] ||
vfrA[47:40] < vfrB[47:40] ||
vfrA[55:48] < vfrB[55:48] ||
vfrA[63:56] < vfrB[63:56]vfrD[63:0] <- repl(flag)
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_lt.b Vector Byte Elements Any Less Than lv.any_lt.b

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

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.any_lt.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if any element of vfrA is less than corresponding element of vfrB;otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] < vfrB[15:0] ||
vfrA[31:16] < vfrB[31:16] ||
vfrA[47:32] < vfrB[47:32] ||
vfrA[63:48] < vfrB[63:48]vfrD[63:0] <- repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_ne.b Vector Byte Elements Any Not Equal lv.any_ne.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x2a
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.any_ne.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Compare flag is set if any two corresponding elements are not equal; otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[7:0] != vfrB[7:0] ||
vfrA[15:8] != vfrB[15:8] ||
vfrA[23:16] != vfrB[23:16] ||
vfrA[31:24] != vfrB[31:24] ||
vfrA[39:32] != vfrB[39:32] ||
vfrA[47:40] != vfrB[47:40] ||
vfrA[55:48] != vfrB[55:48] ||
vfrA[63:56] != vfrB[63:56]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.any_ne.b Vector Byte Elements Any Not Equal lv.any_ne.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x2a
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.any_ne.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Compare flag is set if any two corresponding elements are not equal; otherwise compare flag is cleared.
Compare flag is replicated into all bit positions of vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
flag <- vfrA[15:0] != vfrB[15:0] ||
vfrA[31:16] != vfrB[31:16] ||
vfrA[47:32] != vfrB[47:32] ||
vfrA[63:48] != vfrB[63:48]vfrD[63:0] <-
repl(flag)
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.avg.b Vector Byte Elements Average lv.avg.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x39
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.avg.b rD,rA,rB
```

Description:

The byte elements of vector/floating-point register vfrA are added to the byte elements of vector/floating-point register vfrB and the sum is shifted right by one to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[7:0] <- (vfrA[7:0] + vfrB[7:0]) >> 1
vfrD[15:8] <- (vfrA[15:8] + vfrB[15:8]) >> 1
vfrD[23:16] <- (vfrA[23:16] + vfrB[23:16]) >> 1
vfrD[31:24] <- (vfrA[31:24] + vfrB[31:24]) >> 1
vfrD[39:32] <- (vfrA[39:32] + vfrB[39:32]) >> 1
vfrD[47:40] <- (vfrA[47:40] + vfrB[47:40]) >> 1
vfrD[55:48] <- (vfrA[55:48] + vfrB[55:48]) >> 1
vfrD[63:56] <- (vfrA[63:56] + vfrB[63:56]) >> 1
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.avg.h rD,rA,rB
```

Description:

The half-word elements of vector/floating-point register vfrA are added to the half-word elements of vector/floating-point register vfrB and the sum is shifted right by one to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[15:0] <- (vfrA[15:0] + vfrB[15:0]) >> 1
vfrD[31:16] <- (vfrA[31:16] + vfrB[31:16]) >> 1
vfrD[47:32] <- (vfrA[47:32] + vfrB[47:32]) >> 1
vfrD[63:48] <- (vfrA[63:48] + vfrB[63:48]) >> 1
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.cmp_eq.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if two corresponding compared elements are equal; otherwise element bits are cleared.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- repl(vfrA[7:0] == vfrB[7:0]
vfrD[15:8] <- repl(vfrA[15:8] == vfrB[15:8]
vfrD[23:16] <- repl(vfrA[23:16] == vfrB[23:16]
vfrD[31:24] <- repl(vfrA[31:24] == vfrB[31:24]
vfrD[39:32] <- repl(vfrA[39:32] == vfrB[39:32]
vfrD[47:40] <- repl(vfrA[47:40] == vfrB[47:40]
vfrD[55:48] <- repl(vfrA[55:48] == vfrB[55:48]
vfrD[63:56] <- repl(vfrA[63:56] == vfrB[63:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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			8 bits							

Format:

```
lv.cmp_eq.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if two corresponding compared elements are equal; otherwise element bits are cleared.

32-bit Implementation:

64-bit Implementation:

```
vfrD[15:0] <- repl(vfrA[7:0] == vfrB[7:0]
vfrD[31:16] <- repl(vfrA[23:16] == vfrB[23:16]
vfrD[47:32] <- repl(vfrA[39:32] == vfrB[39:32]
vfrD[63:48] <- repl(vfrA[55:48] == vfrB[55:48]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.cmp_ge.b Vector Byte Elements Compare Greater Than or Equal 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	8 bits

Format:

```
lv.cmp_ge.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if element in vfrA is greater than or equal to element in vfrB; otherwise element bits are cleared.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- repl(vfrA[7:0] >= vfrB[7:0]
vfrD[15:8] <- repl(vfrA[15:8] >= vfrB[15:8]
vfrD[23:16] <- repl(vfrA[23:16] >= vfrB[23:16]
vfrD[31:24] <- repl(vfrA[31:24] >= vfrB[31:24]
vfrD[39:32] <- repl(vfrA[39:32] >= vfrB[39:32]
vfrD[47:40] <- repl(vfrA[47:40] >= vfrB[47:40]
vfrD[55:48] <- repl(vfrA[55:48] >= vfrB[55:48]
vfrD[63:56] <- repl(vfrA[63:56] >= vfrB[63:56]
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.cmp_ge.b Vector Byte Elements Compare Greater Than or Equal 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	8 bits

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.cmp_ge.h Vector Half-Word Elements Compare Greater Than or Equal 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	8 bits

Format:

```
lv.cmp_ge.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if element in vfrA is greater than or equal to element in vfrB; otherwise element bits are cleared.

32-bit Implementation:

64-bit Implementation:

```
vfrD[15:0] <- repl(vfrA[7:0] >= vfrB[7:0]
vfrD[31:16] <- repl(vfrA[23:16] >= vfrB[23:16]
vfrD[47:32] <- repl(vfrA[39:32] >= vfrB[39:32]
vfrD[63:48] <- repl(vfrA[55:48] >= vfrB[55:48]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.cmp_gt.b Vector Byte Elements Compare Greater Than lv.cmp_gt.b

31 26	25 21	20 16	15 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x44
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.cmp_gt.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if element in vfrA is greater than element in vfrB; otherwise element bits are cleared.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- repl(vfrA[7:0] > vfrB[7:0]
vfrD[15:8] <- repl(vfrA[15:8] > vfrB[15:8]
vfrD[23:16] <- repl(vfrA[23:16] > vfrB[23:16]
vfrD[31:24] <- repl(vfrA[31:24] > vfrB[31:24]
vfrD[39:32] <- repl(vfrA[39:32] > vfrB[39:32]
vfrD[47:40] <- repl(vfrA[47:40] > vfrB[47:40]
vfrD[55:48] <- repl(vfrA[55:48] > vfrB[55:48]
vfrD[63:56] <- repl(vfrA[63:56] > vfrB[63:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.cmp_gt.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if element in vfrA is greater than element in vfrB; otherwise element bits are cleared.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- repl(vfrA[7:0] > vfrB[7:0]
vfrD[31:16] <- repl(vfrA[23:16] > vfrB[23:16]
vfrD[47:32] <- repl(vfrA[39:32] > vfrB[39:32]
vfrD[63:48] <- repl(vfrA[55:48] > vfrB[55:48]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.cmp_le.b Vector Byte Elements Compare Less Than or Equal lv.cmp_le.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x46
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.cmp_le.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if element in vfrA is less than or equal to element in vfrB; otherwise element bits are cleared.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- repl(vfrA[7:0] <= vfrB[7:0]
vfrD[15:8] <- repl(vfrA[15:8] <= vfrB[15:8]
vfrD[23:16] <- repl(vfrA[23:16] <= vfrB[23:16]
vfrD[31:24] <- repl(vfrA[31:24] <= vfrB[31:24]
vfrD[39:32] <- repl(vfrA[39:32] <= vfrB[39:32]
vfrD[47:40] <- repl(vfrA[47:40] <= vfrB[47:40]
vfrD[55:48] <- repl(vfrA[55:48] <= vfrB[55:48]
vfrD[63:56] <- repl(vfrA[63:56] <= vfrB[63:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.cmp_le.h Vector Half-Word Elements Compare Less Than or Equal 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	8 bits

Format:

```
lv.cmp_le.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if element in vfrA is less than or equal to element in vfrB; otherwise element bits are cleared.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- repl(vfrA[7:0] <= vfrB[7:0]
vfrD[31:16] <- repl(vfrA[23:16] <= vfrB[23:16]
vfrD[47:32] <- repl(vfrA[39:32] <= vfrB[39:32]
vfrD[63:48] <- repl(vfrA[55:48] <= vfrB[55:48]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.cmp_lt.b Vector Byte Elements Compare Less Than lv.cmp_lt.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x48
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.cmp_lt.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if element in vfrA is less than element in vfrB; otherwise element bits are cleared.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- repl(vfrA[7:0] <= vfrB[7:0]
vfrD[15:8] <- repl(vfrA[15:8] <= vfrB[15:8]
vfrD[23:16] <- repl(vfrA[23:16] <= vfrB[23:16]
vfrD[31:24] <- repl(vfrA[31:24] <= vfrB[31:24]
vfrD[39:32] <- repl(vfrA[39:32] <= vfrB[39:32]
vfrD[47:40] <- repl(vfrA[47:40] <= vfrB[47:40]
vfrD[55:48] <- repl(vfrA[55:48] <= vfrB[55:48]
vfrD[63:56] <- repl(vfrA[63:56] <= vfrB[63:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.cmp_lt.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if element in vfrA is less than element in vfrB; otherwise element bits are cleared.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- repl(vfrA[7:0] <= vfrB[7:0]
vfrD[31:16] <- repl(vfrA[23:16] <= vfrB[23:16]
vfrD[47:32] <- repl(vfrA[39:32] <= vfrB[39:32]
vfrD[63:48] <- repl(vfrA[55:48] <= vfrB[55:48]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.cmp_ne.b rD,rA,rB
```

Description:

All byte elements of vector/floating-point register vfrA are compared to byte elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if two corresponding compared elements are not equal; otherwise element bits are cleared.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- repl(vfrA[7:0] != vfrB[7:0])
vfrD[15:8] <- repl(vfrA[15:8] != vfrB[15:8])
vfrD[23:16] <- repl(vfrA[23:16] != vfrB[23:16])
vfrD[31:24] <- repl(vfrA[31:24] != vfrB[31:24])
vfrD[39:32] <- repl(vfrA[39:32] != vfrB[39:32])
vfrD[47:40] <- repl(vfrA[47:40] != vfrB[47:40])
vfrD[55:48] <- repl(vfrA[55:48] != vfrB[55:48])
vfrD[63:56] <- repl(vfrA[63:56] != vfrB[63:56])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.cmp_ne.h rD,rA,rB
```

Description:

All half-word elements of vector/floating-point register vfrA are compared to half-word elements of vector/floating-point register vfrB. Bits of the element in vector/floating-point register vfrD are set if two corresponding compared elements are not equal; otherwise element bits are cleared.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- repl(vfrA[7:0] != vfrB[7:0])
vfrD[31:16] <- repl(vfrA[23:16] != vfrB[23:16])
vfrD[47:32] <- repl(vfrA[39:32] != vfrB[39:32])
vfrD[63:48] <- repl(vfrA[55:48] != vfrB[55:48])
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	4 bits

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
```

Notes:

Instruction Class	Implementation
ORVDX64 II	Optional

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	4 bits

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
```

Notes:

Instruction Class	Implementation
ORVDX64 II	Optional

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	4 bits

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
```

Notes:

Instruction Class	Implementation
ORVDX64 II	Optional

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	4 bits

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
```

Notes:

Instruction Class	Implementation
ORVDX64 II	Optional

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	8 bits

Format:

```
lv.madds.h rD,rA,rB
```

Description:

The signed half-word elements of vector/floating-point register vfrA are multiplied by the signed half-word elements of vector/floating-point register vfrB to form intermediate results. They are added to the signed half-word VMAC elements to form the final results that are placed again in VMAC registers. Intermediate result is placed into vector/floating-point register vfrD. If any of the final results exceeds min/max value, it is saturated.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- sat32s(vfrA[15:0] * vfrB[15:0] +
VMACLO[31:0])
vfrD[31:16] <- sat32s(vfrA[31:16] * vfrB[31:16] +
VMACLO[63:32])
vfrD[47:32] <- sat32s(vfrA[47:32] * vfrB[47:32] +
VMACHI[31:0])
vfrD[63:48] <- sat32s(vfrA[63:48] * vfrB[63:48] +
VMACHI[63:32])
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.max.b rD,rA,rB
```

Description:

The byte elements of vector/floating-point register vfrA are compared to the byte elements of vector/floating-point register vfrB and larger elements are selected to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[7:0] <- vfrA[7:0] > vfrB[7:0] ? vfrA[7:0] :
vrfB[7:0]
vfrD[15:8] <- vfrA[15:8] > vfrB[15:8] ?
vfrA[15:8] : vrfB[15:8]
vfrD[23:16] <- vfrA[23:16] > vfrB[23:16] ?
vfrA[23:16] : vrfB[23:16]
vfrD[31:24] <- vfrA[31:24] > vfrB[31:24] ?
vfrA[31:24] : vrfB[31:24]
vfrD[39:32] <- vfrA[39:32] > vfrB[39:32] ?
vfrA[39:32] : vrfB[39:32]
vfrD[47:40] <- vfrA[47:40] > vfrB[47:40] ?
vfrA[47:40] : vrfB[47:40]
vfrD[55:48] <- vfrA[55:48] > vfrB[55:48] ?
vfrA[55:48] : vrfB[55:48]
vfrD[63:56] <- vfrA[63:56] > vfrB[63:56] ?
vfrA[63:56] : vrfB[63:56]
```

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Exceptions:

`None`

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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			8 bits							

Format:

```
lv.max.h rD,rA,rB
```

Description:

The half-word elements of vector/floating-point register vfrA are compared to the half-word elements of vector/floating-point register vfrB and larger elements are selected to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrA[15:0] > vfrB[15:0] ?
vfrA[15:0] : vrfB[15:0]
vfrD[31:16] <- vfrA[31:16] > vfrB[31:16] ?
vfrA[31:16] : vrfB[31:16]
vfrD[47:32] <- vfrA[47:32] > vfrB[47:32] ?
vfrA[47:32] : vrfB[47:32]
vfrD[63:48] <- vfrA[63:48] > vfrB[63:48] ?
vfrA[63:48] : vrfB[63:48]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.merge.b rD,rA,rB
```

Description:

Byte elements of the lower half of the vector/floating-point register vfrA are combined with the byte elements of the lower half of vector/floating-point register vfrB in such a way that lowest element is from the vfrB, second element from the vfrA, third again from the vfrB etc. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- vfrB[7:0]
vfrD[15:8] <- vfrA[15:8]
vfrD[23:16] <- vfrB[23:16]
vfrD[31:24] <- vfrA[31:24]
vfrD[39:32] <- vfrB[39:32]
vfrD[47:40] <- vfrA[47:40]
vfrD[55:48] <- vfrB[55:48]
vfrD[63:56] <- vfrA[63:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.merge.h rD,rA,rB
```

Description:

Half-word elements of the lower half of the vector/floating-point register vfrA are combined with the byte elements of the lower half of vector/floating-point register vfrB in such a way that lowest element is from the vfrB, second element from the vfrA, third again from the vfrB etc. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrB[15:0]
vfrD[31:16] <- vfrA[31:16]
vfrD[47:32] <- vfrB[47:32]
vfrD[63:48] <- vfrA[63:48]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.min.b Vector Byte Elements Minimum lv.min.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x59
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.min.b rD,rA,rB
```

Description:

The byte elements of vector/floating-point register vfrA are compared to the byte elements of vector/floating-point register vfrB and smaller elements are selected to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- vfrA[7:0] < vfrB[7:0] ? vfrA[7:0] :
vrfB[7:0]
vfrD[15:8] <- vfrA[15:8] < vfrB[15:8] ?
vfrA[15:8] : vrfB[15:8]
vfrD[23:16] <- vfrA[23:16] < vfrB[23:16] ?
vfrA[23:16] : vrfB[23:16]
vfrD[31:24] <- vfrA[31:24] < vfrB[31:24] ?
vfrA[31:24] : vrfB[31:24]
vfrD[39:32] <- vfrA[39:32] < vfrB[39:32] ?
vfrA[39:32] : vrfB[39:32]
vfrD[47:40] <- vfrA[47:40] < vfrB[47:40] ?
vfrA[47:40] : vrfB[47:40]
vfrD[55:48] <- vfrA[55:48] < vfrB[55:48] ?
vfrA[55:48] : vrfB[55:48]
vfrD[63:56] <- vfrA[63:56] < vfrB[63:56] ?
vfrA[63:56] : vrfB[63:56]
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.min.b Vector Byte Elements Minimum lv.min.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	.	0
opcode 0xa						D					A					B					reserved			opcode 0x59							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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			8 bits							

Format:

```
lv.min.h rD,rA,rB
```

Description:

The half-word elements of vector/floating-point register vfrA are compared to the half-word elements of vector/floating-point register vfrB and smaller elements are selected to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrA[15:0] < vfrB[15:0] ?
vfrA[15:0] : vrfB[15:0]
vfrD[31:16] <- vfrA[31:16] < vfrB[31:16] ?
vfrA[31:16] : vrfB[31:16]
vfrD[47:32] <- vfrA[47:32] < vfrB[47:32] ?
vfrA[47:32] : vrfB[47:32]
vfrD[63:48] <- vfrA[63:48] < vfrB[63:48] ?
vfrA[63:48] : vrfB[63:48]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.msubs.h rD,rA,rB
```

Description:

The signed half-word elements of vector/floating-point register vfrA are multiplied by the signed half-word elements of vector/floating-point register vfrB to form intermediate results. They are subtracted from the signed half-word VMAC elements to form the final results that are placed again in VMAC registers. Intermediate result is placed into vector/floating-point register vfrD. If any of the final results exceeds min/max value, it is saturated.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- sat32s(VMACLO[31:0] - vfrA[15:0] *
vfrB[15:0])
vfrD[31:16] <- sat32s(VMACLO[63:32] - vfrA[31:16]
* vfrB[31:16])
vfrD[47:32] <- sat32s(VMACHI[31:0] - vfrA[47:32]
* vfrB[47:32])
vfrD[63:48] <- sat32s(VMACHI[63:32] - vfrA[63:48]
* vfrB[63:48])
```

Exceptions:

Instruction Class	Implementation
ORVDX64 I	Required

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			8 bits							

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.muls.h rD,rA,rB
```

Description:

The signed half-word elements of vector/floating-point register vfrA are multiplied by the signed half-word elements of vector/floating-point register vfrB to form the results. The result is placed into vector/floating-point register vfrD. If any of the final results exceeds min/max value, it is saturated.

32-bit Implementation:

64-bit Implementation:

```
vfrD[15:0] <- sat32s(vfrA[15:0] * vfrB[15:0])
vfrD[31:16] <- sat32s(vfrA[31:16] * vfrB[31:16])
vfrD[47:32] <- sat32s(vfrA[47:32] * vfrB[47:32])
vfrD[63:48] <- sat32s(vfrA[63:48] * vfrB[63:48])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 II	Optional

lv.nand Vector Not And lv.nand

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x5d
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.nand rD,rA,rB
```

Description:

The contents of vector/floating-point register vfrA are combined with the contents of vector/floating-point register vfrB in a bit-wise logical NAND operation. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] NAND vfrB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.nor rD,rA,rB
```

Description:

The contents of vector/floating-point register vfrA are combined with the contents of vector/floating-point register vfrB in a bit-wise logical NOR operation. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] NOR vfrB[63:0]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.or rD,rA,rB
```

Description:

The contents of vector/floating-point register vfrA are combined with the contents of vector/floating-point register vfrB in a bit-wise logical OR operation. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] OR vfrB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.pack.b rD,rA,rB
```

Description:

Lower half of the byte elements of the vector/floating-point register vfrA are truncated and combined with the lower half of the byte truncated elements of the vector/floating-point register vfrB in such a way that lowest elements are from vfrB and highest element from vfrA. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[3:0] <- vfrB[3:0]
vfrD[7:4] <- vfrB[11:8]
vfrD[11:8] <- vfrB[19:16]
vfrD[15:12] <- vfrB[27:24]
vfrD[19:16] <- vfrB[35:32]
vfrD[23:20] <- vfrB[43:40]
vfrD[27:24] <- vfrB[51:48]
vfrD[31:28] <- vfrB[59:56]
vfrD[35:32] <- vfrA[3:0]
vfrD[39:36] <- vfrA[11:8]
vfrD[43:40] <- vfrA[19:16]
vfrD[47:44] <- vfrA[27:24]
vfrD[51:48] <- vfrA[35:32]
vfrD[55:52] <- vfrA[43:40]
vfrD[59:56] <- vfrA[51:48]
vfrD[63:60] <- vfrA[59:56]
```

Instruction Class	Implementation
ORVDX64 I	Required

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			8 bits							

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.pack.h rD,rA,rB
```

Description:

Lower half of the half-word elements of the vector/floating-point register vfrA are truncated and combined with the lower half of the half-word truncated elements of the vector/floating-point register vfrB in such a way that lowest elements are from vfrB and highest element from vfrA. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- vfrB[15:0]
vfrD[15:8] <- vfrB[31:16]
vfrD[23:16] <- vfrB[47:32]
vfrD[31:24] <- vfrB[63:48]
vfrD[39:32] <- vfrA[15:0]
vfrD[47:40] <- vfrA[31:16]
vfrD[55:48] <- vfrA[47:32]
vfrD[63:56] <- vfrA[63:48]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.packs.b rD,rA,rB
```

Description:

Lower half of the signed byte elements of the vector/floating-point register vfrA are truncated and combined with the lower half of the signed byte truncated elements of the vector/floating-point register vfrB in such a way that lowest elements are from vfrB and highest element from vfrA. If any truncated element exceeds signed 4-bit value, it is saturated. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[3:0] <- sat4s(vfrB[7:0]
vfrD[7:4] <- sat4s(vfrB[15:8]
vfrD[11:8] <- sat4s(vfrB[23:16]
vfrD[15:12] <- sat4s(vfrB[31:24]
vfrD[19:16] <- sat4s(vfrB[39:32]
vfrD[23:20] <- sat4s(vfrB[47:40]
vfrD[27:24] <- sat4s(vfrB[55:48]
vfrD[31:28] <- sat4s(vfrB[63:56]
vfrD[35:32] <- sat4s(vfrA[7:0]
vfrD[39:36] <- sat4s(vfrA[15:8]
vfrD[43:40] <- sat4s(vfrA[23:16]
vfrD[47:44] <- sat4s(vfrA[31:24]
vfrD[51:48] <- sat4s(vfrA[39:32]
vfrD[55:52] <- sat4s(vfrA[47:40]
```

Instruction Class	Implementation
ORVDX64 I	Required

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			8 bits							

```
vfrD[59:56] <- sat4s(vfrA[55:48]
vfrD[63:60] <- sat4s(vfrA[63:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.packs.h rD,rA,rB
```

Description:

Lower half of the signed halfpword elements of the vector/floating-point register vfrA are truncated and combined with the lower half of the signed half-word truncated elements of the vector/floating-point register vfrB in such a way that lowest elements are from vfrB and highest element from vfrA. If any truncated element exceeds signed 8-bit value, it is saturated. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- sat8s(vfrB[15:0])
vfrD[15:8] <- sat8s(vfrB[31:16])
vfrD[23:16] <- sat8s(vfrB[47:32])
vfrD[31:24] <- sat8s(vfrB[63:48])
vfrD[39:32] <- sat8s(vfrA[15:0])
vfrD[47:40] <- sat8s(vfrA[31:16])
vfrD[55:48] <- sat8s(vfrA[47:32])
vfrD[63:56] <- sat8s(vfrA[63:48])
```

Exceptions:

```
None
```

Instruction Class	Implementation
ORVDX64 I	Required

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			8 bits							

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.packus.b rD,rA,rB
```

Description:

Lower half of the unsigned byte elements of the vector/floating-point register vfrA are truncated and combined with the lower half of the unsigned byte truncated elements of the vector/floating-point register vfrB in such a way that lowest elements are from vfrB and highest element from vfrA. If any truncated element exceeds unsigned 4-bit value, it is saturated. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[3:0] <- sat4u(vfrB[7:0]
vfrD[7:4] <- sat4u(vfrB[15:8]
vfrD[11:8] <- sat4u(vfrB[23:16]
vfrD[15:12] <- sat4u(vfrB[31:24]
vfrD[19:16] <- sat4u(vfrB[39:32]
vfrD[23:20] <- sat4u(vfrB[47:40]
vfrD[27:24] <- sat4u(vfrB[55:48]
vfrD[31:28] <- sat4u(vfrB[63:56]
vfrD[35:32] <- sat4u(vfrA[7:0]
vfrD[39:36] <- sat4u(vfrA[15:8]
vfrD[43:40] <- sat4u(vfrA[23:16]
vfrD[47:44] <- sat4u(vfrA[31:24]
```

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

```
vfrD[51:48] <- sat4u(vfrA[39:32]
vfrD[55:52] <- sat4u(vfrA[47:40]
vfrD[59:56] <- sat4u(vfrA[55:48]
vfrD[63:60] <- sat4u(vfrA[63:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.packus.h rD,rA,rB
```

Description:

Lower half of the unsigned halfpword elements of the vector/floating-point register vfrA are truncated and combined with the lower half of the unsigned half-word truncated elements of the vector/floating-point register vfrB in such a way that lowest elements are from vfrB and highest element from vfrA. If any truncated element exceeds unsigned 8-bit value, it is saturated. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[7:0] <- sat8u(vfrB[15:0])
vfrD[15:8] <- sat8u(vfrB[31:16])
vfrD[23:16] <- sat8u(vfrB[47:32])
vfrD[31:24] <- sat8u(vfrB[63:48])
vfrD[39:32] <- sat8u(vfrA[15:0])
vfrD[47:40] <- sat8u(vfrA[31:16])
vfrD[55:48] <- sat8u(vfrA[47:32])
vfrD[63:56] <- sat8u(vfrA[63:48])
```

Exceptions:

None

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.perm.n Vector Nibble Elements Permute lv.perm.n

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x66
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.perm.n rD,rA,rB
```

Description:

The 4-bit elements of vector/floating-point register vfrA are permuted according to corresponding 4-bit values in vector/floating-point register vfrB. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[3:0] <- vfrA[vfrB[3:0]*4+3:vfrB[3:0]*4]
vfrD[7:4] <- vfrA[vfrB[7:4]*4+3:vfrB[7:4]*4]
vfrD[11:8] <- vfrA[vfrB[11:8]*4+3:vfrB[11:8]*4]
vfrD[15:12] <-
vfrA[vfrB[15:12]*4+3:vfrB[15:12]*4]
vfrD[19:16] <-
vfrA[vfrB[19:16]*4+3:vfrB[19:16]*4]
vfrD[23:20] <-
vfrA[vfrB[23:20]*4+3:vfrB[23:20]*4]
vfrD[27:24] <-
vfrA[vfrB[27:24]*4+3:vfrB[27:24]*4]
vfrD[31:28] <-
vfrA[vfrB[31:28]*4+3:vfrB[31:28]*4]
vfrD[35:32] <-
vfrA[vfrB[35:32]*4+3:vfrB[35:32]*4]
vfrD[39:36] <-
vfrA[vfrB[39:36]*4+3:vfrB[39:36]*4]
vfrD[43:40] <-
vfrA[vfrB[43:40]*4+3:vfrB[43:40]*4]
```

Instruction Class	Implementation
ORVDX64 I	Required

lv.perm.n Vector Nibble Elements Permute lv.perm.n

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x66
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

```
vfrD[47:44] <-
vfrA[vfrB[47:44]*4+3:vfrB[47:44]*4]
vfrD[51:48] <-
vfrA[vfrB[51:48]*4+3:vfrB[51:48]*4]
vfrD[55:52] <-
vfrA[vfrB[55:52]*4+3:vfrB[55:52]*4]
vfrD[59:56] <-
vfrA[vfrB[59:56]*4+3:vfrB[59:56]*4]
vfrD[63:60] <-
vfrA[vfrB[63:60]*4+3:vfrB[63:60]*4]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.rl.b Vector Byte Elements Rotate Left lv.rl.b

31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	8	7	.	.	.	.	.	.	0
opcode 0xa						D					A					B					reserved			opcode 0x67							
6 bits						5 bits					5 bits					5 bits					3 bits			8 bits							

Format:

```
lv.rl.b rD,rA,rB
```

Description:

The contents of byte elements of vector/floating-point register vfrA are rotated left by the number of bits specified in lower 3 bits in each byte element of vector/floating-point register vfrB. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- vfrA[7:0] rl vfrB[2:0]
vfrD[15:8] <- vfrA[15:8] rl vfrB[10:8]
vfrD[23:16] <- vfrA[23:16] rl vfrB[18:16]
vfrD[31:24] <- vfrA[31:24] rl vfrB[26:24]
vfrD[39:32] <- vfrA[39:32] rl vfrB[34:32]
vfrD[47:40] <- vfrA[47:40] rl vfrB[42:40]
vfrD[55:48] <- vfrA[55:48] rl vfrB[50:48]
vfrD[63:56] <- vfrA[63:56] rl vfrB[58:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.rl.h rD,rA,rB
```

Description:

The contents of half-word elements of vector/floating-point register vfrA are rotated left by the number of bits specified in lower 4 bits in each half-word element of vector/floating-point register vfrB. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrA[15:0] rl vfrB[3:0]
vfrD[31:16] <- vfrA[31:16] rl vfrB[19:16]
vfrD[47:32] <- vfrA[47:32] rl vfrB[35:32]
vfrD[63:48] <- vfrA[63:48] rl vfrB[51:48]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.sll rD,rA,rB
```

Description:

The contents of vector/floating-point register vfrA are shifted left by the number of bits specified in lower 4 bits in each byte element of vector/floating-point register vfrB, inserting zeros into the low-order bits of vfrD. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] << vfrB[2:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.sll.b rD,rA,rB
```

Description:

The contents of byte elements of vector/floating-point register vfrA are shifted left by the number of bits specified in lower 3 bits in each byte element of vector/floating-point register vfrB, inserting zeros into the low-order bits elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- vfrA[7:0] << vfrB[2:0]
vfrD[15:8] <- vfrA[15:8] << vfrB[10:8]
vfrD[23:16] <- vfrA[23:16] << vfrB[18:16]
vfrD[31:24] <- vfrA[31:24] << vfrB[26:24]
vfrD[39:32] <- vfrA[39:32] << vfrB[34:32]
vfrD[47:40] <- vfrA[47:40] << vfrB[42:40]
vfrD[55:48] <- vfrA[55:48] << vfrB[50:48]
vfrD[63:56] <- vfrA[63:56] << vfrB[58:56]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.sll.h rD,rA,rB
```

Description:

The contents of half-word elements of vector/floating-point register vfrA are shifted left by the number of bits specified in lower 4 bits in each half-word element of vector/floating-point register vfrB, inserting zeros into the low-order bits elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrA[15:0] << vfrB[3:0]
vfrD[31:16] <- vfrA[31:16] << vfrB[19:16]
vfrD[47:32] <- vfrA[47:32] << vfrB[35:32]
vfrD[63:48] <- vfrA[63:48] << vfrB[51:48]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.sra.b rD,rA,rB
```

Description:

The contents of byte elements of vector/floating-point register vfrA are shifted right by the number of bits specified in lower 3 bits in each byte element of vector/floating-point register vfrB, inserting most significat bit of each element into the high-order bits. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- vfrA[7:0] sra vfrB[2:0]
vfrD[15:8] <- vfrA[15:8] sra vfrB[10:8]
vfrD[23:16] <- vfrA[23:16] sra vfrB[18:16]
vfrD[31:24] <- vfrA[31:24] sra vfrB[26:24]
vfrD[39:32] <- vfrA[39:32] sra vfrB[34:32]
vfrD[47:40] <- vfrA[47:40] sra vfrB[42:40]
vfrD[55:48] <- vfrA[55:48] sra vfrB[50:48]
vfrD[63:56] <- vfrA[63:56] sra vfrB[58:56]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.sra.h rD,rA,rB
```

Description:

The contents of half-word elements of vector/floating-point register vfrA are shifted right by the number of bits specified in lower 4 bits in each half-word element of vector/floating-point register vfrB, inserting most significant bit of each element into the low-order bits. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrA[15:0] sra vfrB[3:0]
vfrD[31:16] <- vfrA[31:16] sra vfrB[19:16]
vfrD[47:32] <- vfrA[47:32] sra vfrB[35:32]
vfrD[63:48] <- vfrA[63:48] sra vfrB[51:48]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.srl Vector Shift Right Logical lv.srl

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x70
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.srl rD,rA,rB
```

Description:

The contents of vector/floating-point register vfrA are shifted right by the number of bits specified in lower 4 bits in each byte element of vector/floating-point register vfrB, inserting zeros into the high-order bits of vfrD. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] >> vfrB[2:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.srl.b rD,rA,rB
```

Description:

The contents of byte elements of vector/floating-point register vfrA are shifted right by the number of bits specified in lower 3 bits in each byte element of vector/floating-point register vfrB, inserting zeros into the high-order bits elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- vfrA[7:0] >> vfrB[2:0]
vfrD[15:8] <- vfrA[15:8] >> vfrB[10:8]
vfrD[23:16] <- vfrA[23:16] >> vfrB[18:16]
vfrD[31:24] <- vfrA[31:24] >> vfrB[26:24]
vfrD[39:32] <- vfrA[39:32] >> vfrB[34:32]
vfrD[47:40] <- vfrA[47:40] >> vfrB[42:40]
vfrD[55:48] <- vfrA[55:48] >> vfrB[50:48]
vfrD[63:56] <- vfrA[63:56] >> vfrB[58:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.srl.h rD,rA,rB
```

Description:

The contents of half-word elements of vector/floating-point register vfrA are shifted right by the number of bits specified in lower 4 bits in each half-word element of vector/floating-point register vfrB, inserting zeros into the low-order bits of elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrA[15:0] >> vfrB[3:0]
vfrD[31:16] <- vfrA[31:16] >> vfrB[19:16]
vfrD[47:32] <- vfrA[47:32] >> vfrB[35:32]
vfrD[63:48] <- vfrA[63:48] >> vfrB[51:48]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.sub.b rD,rA,rB
```

Description:

The byte elements of vector/floating-point register vfrB are subtracted from the byte elements of vector/floating-point register vfrA to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- vfrA[7:0] - vfrB[7:0]
vfrD[15:8] <- vfrA[15:8] - vfrB[15:8]
vfrD[23:16] <- vfrA[23:16] - vfrB[23:16]
vfrD[31:24] <- vfrA[31:24] - vfrB[31:24]
vfrD[39:32] <- vfrA[39:32] - vfrB[39:32]
vfrD[47:40] <- vfrA[47:40] - vfrB[47:40]
vfrD[55:48] <- vfrA[55:48] - vfrB[55:48]
vfrD[63:56] <- vfrA[63:56] - vfrB[63:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.sub.h rD,rA,rB
```

Description:

The half-word elements of vector/floating-point register vfrB are subtracted from the half-word elements of vector/floating-point register vfrA to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrA[15:0] - vfrB[15:0]
vfrD[31:16] <- vfrA[31:16] - vfrB[31:16]
vfrD[47:32] <- vfrA[47:32] - vfrB[47:32]
vfrD[63:48] <- vfrA[63:48] - vfrB[63:48]
```

Exceptions:

None

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.subs.b rD,rA,rB
```

Description:

The byte elements of vector/floating-point register vfrB are subtracted from the byte elements of vector/floating-point register vfrA to form the result elements. If the result exceeds min/max value for the destination data type, it is saturated to min/max value and placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- sat8s(vfrA[7:0] + vfrB[7:0])
vfrD[15:8] <- sat8s(vfrA[15:8] + vfrB[15:8])
vfrD[23:16] <- sat8s(vfrA[23:16] + vfrB[23:16])
vfrD[31:24] <- sat8s(vfrA[31:24] + vfrB[31:24])
vfrD[39:32] <- sat8s(vfrA[39:32] + vfrB[39:32])
vfrD[47:40] <- sat8s(vfrA[47:40] + vfrB[47:40])
vfrD[55:48] <- sat8s(vfrA[55:48] + vfrB[55:48])
vfrD[63:56] <- sat8s(vfrA[63:56] + vfrB[63:56])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.subs.h rD,rA,rB
```

Description:

The half-word elements of vector/floating-point register vfrB are subtracted from the half-word elements of vector/floating-point register vfrA to form the result elements. If the result exceeds min/max value for the destination data type, it is saturated to min/max value and placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- sat16s(vfrA[15:0] - vfrB[15:0])
vfrD[31:16] <- sat16s(vfrA[31:16] - vfrB[31:16])
vfrD[47:32] <- sat16s(vfrA[47:32] - vfrB[47:32])
vfrD[63:48] <- sat16s(vfrA[63:48] - vfrB[63:48])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.subu.b rD,rA,rB
```

Description:

The unsigned byte elements of vector/floating-point register vfrB are subtracted from the unsigned byte elements of vector/floating-point register vfrA to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- vfrA[7:0] - vfrB[7:0]
vfrD[15:8] <- vfrA[15:8] - vfrB[15:8]
vfrD[23:16] <- vfrA[23:16] - vfrB[23:16]
vfrD[31:24] <- vfrA[31:24] - vfrB[31:24]
vfrD[39:32] <- vfrA[39:32] - vfrB[39:32]
vfrD[47:40] <- vfrA[47:40] - vfrB[47:40]
vfrD[55:48] <- vfrA[55:48] - vfrB[55:48]
vfrD[63:56] <- vfrA[63:56] - vfrB[63:56]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.subu.h rD,rA,rB
```

Description:

The unsigned half-word elements of vector/floating-point register vfrB are subtracted from the unsigned half-word elements of vector/floating-point register vfrA to form the result elements. Result elements are placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- vfrA[15:0] - vfrB[15:0]
vfrD[31:16] <- vfrA[31:16] - vfrB[31:16]
vfrD[47:32] <- vfrA[47:32] - vfrB[47:32]
vfrD[63:48] <- vfrA[63:48] - vfrB[63:48]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

lv.subus.b Vector Byte Elements Subtract Unsigned Saturated lv.subus.b

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . 8	7 0
opcode 0xa	D	A	B	reserved	opcode 0x77
6 bits	5 bits	5 bits	5 bits	3 bits	8 bits

Format:

```
lv.subus.b rD,rA,rB
```

Description:

The unsigned byte elements of vector/floating-point register vfrB are subtracted from the unsigned byte elements of vector/floating-point register vfrA to form the result elements. If the result exceeds min/max value for the destination data type, it is saturated to min/max value and placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[7:0] <- sat8u(vfrA[7:0] + vfrB[7:0])
vfrD[15:8] <- sat8u(vfrA[15:8] + vfrB[15:8])
vfrD[23:16] <- sat8u(vfrA[23:16] + vfrB[23:16])
vfrD[31:24] <- sat8u(vfrA[31:24] + vfrB[31:24])
vfrD[39:32] <- sat8u(vfrA[39:32] + vfrB[39:32])
vfrD[47:40] <- sat8u(vfrA[47:40] + vfrB[47:40])
vfrD[55:48] <- sat8u(vfrA[55:48] + vfrB[55:48])
vfrD[63:56] <- sat8u(vfrA[63:56] + vfrB[63:56])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.subus.h rD,rA,rB
```

Description:

The unsigned half-word elements of vector/floating-point register vfrB are subtracted from the unsigned half-word elements of vector/floating-point register vfrA to form the result elements. If the result exceeds min/max value for the destination data type, it is saturated to min/max value and placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- sat16u(vfrA[15:0] - vfrB[15:0])
vfrD[31:16] <- sat16u(vfrA[31:16] - vfrB[31:16])
vfrD[47:32] <- sat16u(vfrA[47:32] - vfrB[47:32])
vfrD[63:48] <- sat16u(vfrA[63:48] - vfrB[63:48])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.unpack.b rD,rA,rB
```

Description:

Lower half of 4-bit elements in vector/floating-point register vfrA are sign-extended and placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[7:0] <- exts(vfrA[3:0])
vfrD[15:8] <- exts(vfrA[7:4])
vfrD[23:16] <- exts(vfrA[11:8])
vfrD[31:24] <- exts(vfrA[15:12])
vfrD[39:32] <- exts(vfrA[19:16])
vfrD[47:40] <- exts(vfrA[23:20])
vfrD[55:48] <- exts(vfrA[27:24])
vfrD[63:56] <- exts(vfrA[31:28])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.unpack.h rD,rA,rB
```

Description:

Lower half of 8-bit elements in vector/floating-point register vfrA are sign-extended and placed into vector/floating-point register vfrD.

32-bit Implementation:
64-bit Implementation:

```
vfrD[15:0] <- exts(vfrA[7:0])
vfrD[31:16] <- exts(vfrA[15:8])
vfrD[47:32] <- exts(vfrA[23:16])
vfrD[63:48] <- exts(vfrA[31:24])
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

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	8 bits

Format:

```
lv.xor rD,rA,rB
```

Description:

The contents of vector/floating-point register vfrA are combined with the contents of vector/floating-point register vfrB in a bit-wise logical XOR operation. The result is placed into vector/floating-point register vfrD.

32-bit Implementation:

64-bit Implementation:

```
vfrD[63:0] <- vfrA[63:0] XOR vfrB[63:0]
```

Exceptions:

```
None
```

Notes:

Instruction Class	Implementation
ORVDX64 I	Required

9 Exception Model

This chapter describes exception mechanism, exception types and their handling.

9.1 Introduction

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.

OpenRISC 1000 defines 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.

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.

9.2 Exception Classes

All exceptions can be described as precise or imprecise and either synchronous or asynchronous. Synchronous are caused by instructions and asynchronous are caused by events external to the processor.

TYPE	EXCEPTION
Asynchronous/nonmaskable	Bus Error Reset
Asynchronous/maskable	External Interrupt
Synchronous/precise	Instruction-caused exceptions excluding floating-point imprecise exceptions
Synchronous/imprecise	Instruction-caused floating-point imprecise exceptions

Table 9-1. Exception Classes

Whenever an exception occurs, current PC is saved to current EPCR and new PC is set with the vector address according to the Table 9-2.

EXCEPTION TYPE	VECTOR OFFSET	CAUSING CONDITIONS
Reset	0x100	Caused by soft and hard 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.
Low Priority External Interrupt	0x500	Low priority external interrupt asserted.
Alignment	0x600	Load/store access to naturally not aligned location.
Illegal Instruction	0x700	Illegal instruction in the instruction stream. On OpenRISC implementations with lower 16 GPRs when accessing upper 16 registers out of 32 architectural GPRs. On all implementations if SR[CID] would have to go out of range in order to process next exception.
High Priority External Interrupt	0x800	High priority 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 SR, setting of certain flags, like SR[OV], causes range exception.
System Call	0xC00	System call initiated by software.
Breakpoint	0xD00	Breakpoint instruction detected in instruction Stream or initiated by the debug module.
Reserved	0xE00	Reserved for future use.
Reserved	0xF00	Reserved for future use.
Reserved	0x1000 - 0x1800	Reserved for implementation-specific exceptions.
Reserved	0x1900 - 0x1F00	Reserved for custom exceptions.

Table 9-2. Exception Types and causing conditions

9.3 Exception Processing

Whenever an exception occurs, current PC is saved to current EPCR. SR is saved to the current ESR. Furthermore 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

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, range exception is invoked.

However if SR[CE] is not set, fast context switching is not enabled. In this case all registers must be saved by the exception handler routine.

All exceptions set new SR where both MMUs are disabled (address translation disabled), supervisor mode is turned on and new exceptions are disabled (SR[DME]=0, SR[IME]=0, SR[SUPV=1, SR[EXR]=0).

When enough machine state information has been saved by the exception handler, SR[EXR] should be set so that new exceptions can be nested. If desired, interrupt exceptions can be masked with SR[EIR], regardless that all other exceptions have been re-enabled.

When returning from exception handler with **l.rfe**, CID will be automatically decremented and previous machine state will be restored. If shadowed registers are not enabled, clear SR[EXR] and set appropriately current EPCR and ESR registers before executing **l.rfe**.

9.4 Fast Context Switching (optional)

Fast context switching is a technique, which reduces register storing to stack when exceptions occur. Only one type of exception can be handled, so its up to the software to figure out what caused it. Using software, both interrupt handler invokation and thread switching can be handled very fast. Hardware should be capable of switching between contexts in only one cycle.

Context can also be switched during exception, or by using a supervisor register CXR (context register) available only in supervisor mode. CXR is the same for all contexts.

9.4.1 Changing Context in Supervisor Mode

Read/write register CXR consists of two parts: lower 16 bits represents current context register set and upper ones, which represents current CID. CCID cannot be accessed in user mode. Writing to CCID causes immediate context change. Reading from CCID returns running (current) context ID. Context with CID=0 is also called the main context.

BIT	31-16	15-0
Identifier	CCID	CCRS
Reset	0	0

CCRS has two functions:

- When exception occurs it holds previous CID
- It is used to access other context's registers

9.4.2 Context Switch Caused by Exception

When exception occurs and fast context switching is enabled, CCID is copied to CCRS and then set to zero, thus switching to main context.
Functions of the main context are:

- switch between threads
- handle exceptions
- prepare, load, save and release context identifiers to/from CID table

Also algorithm should store CXR in its general purpose register as soon as possible, to allow further exception nesting.

The following table shows example how CID table could be used. Generally there is no need that free exception contexts are equal.

CID	Function
7	
6	Exception contexts
5	
4	
3	Thread contexts
2	
1	
0	Main context

As we can see, we have four thread contexts loaded and we can switch between them freely using main context, running in supervisor mode. When exception occurs, we find out what caused it and switch to the next free exception context. Since exceptions can be nested we may need more free contexts as we have them available. We are then forced to store some of them to memory and then switch to new exception.

Algorithm used in main context should be kept as simple as possible. It should have enough (its own) registers to store information such:

- current running CID
- next exception
- thread cycling info
- pointers to context table in memory
- copy of CXR

If number of interrupts is large, we can use some sort of defered interrupt calls. Main context algorithm should store just I/O information passed by interrupt for further execution and return from main context as soon as possible.

9.4.3 Accessing Other Context Registers

This operation can be done only in supervisor mode. In basic instruction set we have l.mtspr and l.mfspr instructions.

9.4.4 System Calls and Parameter Passing

System calls can also be called through context switching. We can deliberately cause exception like `l.j -x`, which tries to jump in protected system area and causes page fault exception. Main context algorithm then finds out, that exception was caused by illegal jump and addresses function with index x. Called kernel function then retrieves registers and stores result to previous thread registers using copy of CXR.

10 Memory Model

This chapter describes the OpenRISC 1000 weakly-ordered memory model.

10.1 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.

10.2 Memory Access Ordering

The OpenRISC 1000 architecture specifies weakly-ordered memory model for uniprocessor and shared memory multiprocessor systems. This model has advantage of higher-performance memory system but places 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 more restrictive strongly-ordered memory model. Programs written for weakly-ordered memory model will automatically work on processors with strongly-ordered memory model.

10.2.1 Memory Synchronize Instruction

The **l.msync** instruction permit 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 strongly-ordered memory model instead of weakly-ordered one, can execute memory synchronization instruction as a no-operation instruction.

10.2.2 Pages Designated as Weakly-Ordered-Memory

When a memory page is designated as 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 weakly-ordered one and perform load and store operations in-order, are not required to implement WOM bit in the MMU.

10.3 Atomicity

A memory access is atomic if it is always performed in its entirerty with no visible fragmentation. Atomic memory accesses are specifically required to implement software semaphores and other shared structures in systems where two different processes on the same processor, or two different processors in a multiprocessor environment, access the same memory location with intent to modify it.

OpenRISC 1000 architecture provides two dedicated instructions that together perform atomic read-modify-write operation.

l.lwa rD, I(rA)
l.swa I(rA), rB

Instruction **l.lwa** loads single word from memory, creating a reservation for a subsequent conditional store operation. Special register, invisible to the programmer, is used to hold address of the memory location, which is used in the atomic read-modify-write operation.
Reservation for subsequent l.swa is cancelled if another master read the same memory location (snoop hit), another l.lwa is executed or if the software explicitly clears reservation register.

If reservation is still valid when corresponding l.swa is executed, l.swa stores general-purpose register rB into the memory. If reservation was cancelled, l.swa is executed as *no operation*.

11 Memory Management

This chapter describes the virtual memory and access protection mechanism of memory management of OpenRISC 1000 architecture.

Note that this chapter describes 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.

11.1 MMU Features

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)
  - Level 1 pages (16 MByte)
  - Level 2 pages (8 Kbyte)
- 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)

11.2 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 basis. Note that this chapter describes conceptual model of OpenRISC 1000 MMU and implementations may differ in the specific hardware used to implement the MMU model.

Two general types of accesses generated by OpenRISC 1000 processors require address translation – instruction accesses generated by fetch unit, and data accesses generated by 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.

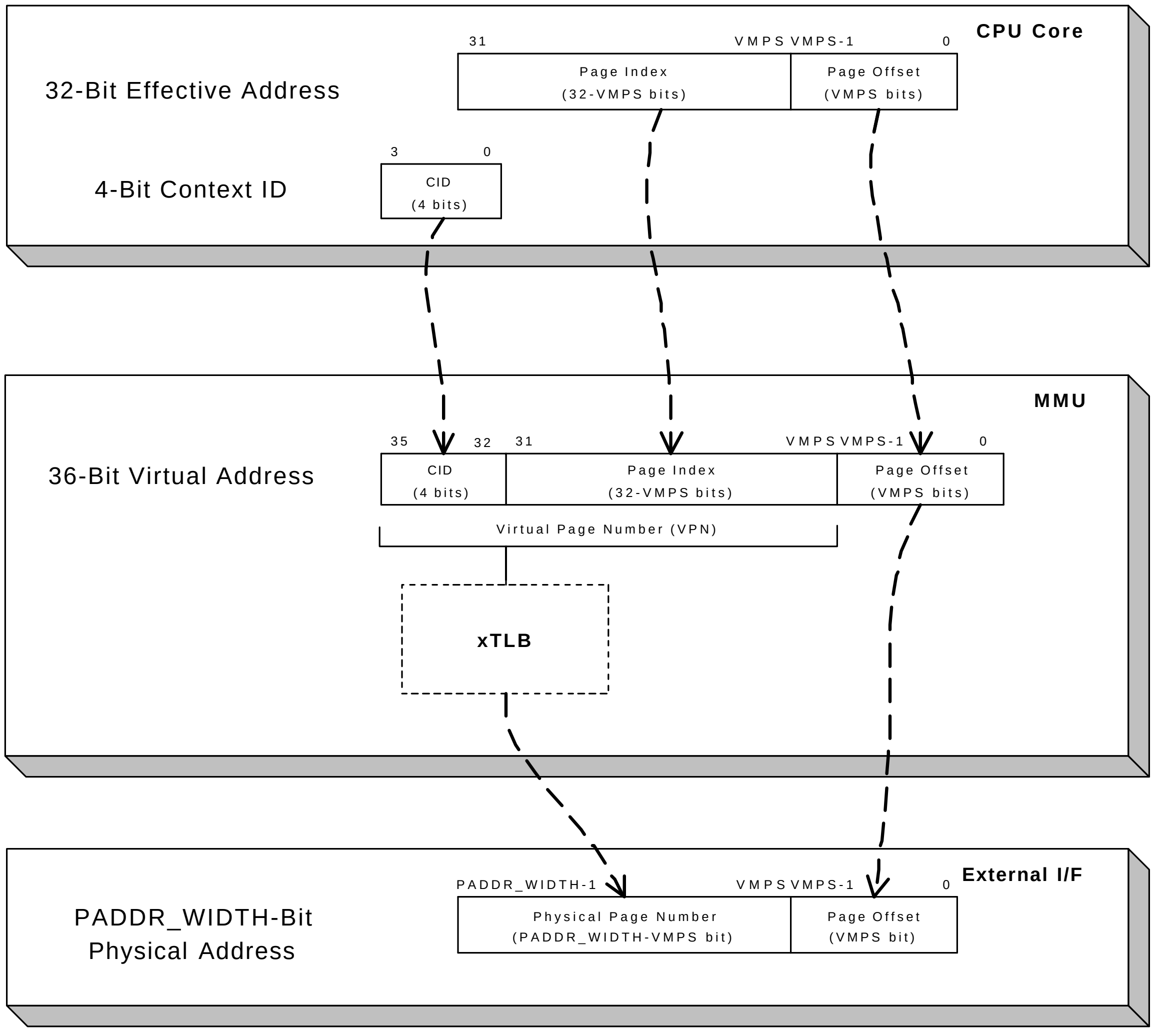


Figure 11-1. Translation of Effective to Physical Address – Simplified block diagram for 32-bit processor implementations

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.

11.3 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 only on OpenRISC 1000 processor implementations that do TLB reload in hardware.

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 TLB.

Table 11-1. 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 chapter "Exception Model" on page 271 for more detailed description of exception processing.

11.4 MMU Special-Purpose Registers

Table 11-2 summarizes the registers that the operating system uses to program the MMU. These registers are 32-bit special-purpose supervisor-level registers accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode.

GRP #	REG #	REG NAME	USER MODE	SUPV MODE	DESCRIPTION
1	0-255	DTLBMR0-DTLBMR255	-	R/W	Data TLB Match Registers
1	256-511	DTLBTR0-DTLBTR255	-	R/W	Data TLB Translate Registers

1	512	DMMUCR	-	R/W	Data MMU Control Register
1	513	DMMUPR	-	R/W	Data MMU Protection Register
1	514	DTLBEIR	-	W	Data TLB Entry Invalidate Register
2	0-255	ITLBMR0-ITLBMR255	-	R/W	Instruction TLB Match Registers
2	256-511	ITLBTR0-ITLBTR255	-	R/W	Instruction TLB Translate Registers
2	512	IMMUCR	-	R/W	Instruction MMU Control Register
2	513	IMMUPR	-	R/W	Instruction MMU Protection Register
2	514	ITLBEIR	-	R/W	Instruction TLB Entry Invalidate Register

Table 11-2. 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.

11.4.1 Data MMU Control Register (DMMUCR)

The DMMUC register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is a 32 bits wide register.

It provides general control of the DMMU.

BIT	31-10	9-1	0
Identifier	PTBP	Reserved	DTF
Reset	0	X	0
R/W	R/W	R	R/W

DTF	DTLB Flush 0 DTLB ready for operation 1 DTLB flush request/status
PTBP	Page Table Base Pointer N 22-bit pointer to the base of page directory/table

Table 11-3. DMMUCR Field Descriptions

PTB field in 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 page table base pointer is stored in a TLB miss exception handler's variable.
DTF is optional and when implemented it flushes entire DTLB.

11.4.2 Data MMU Protection Register (DMMUPR)

The DMMUP register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is a 32 bits wide register.

It defines 7 protection groups indexed by PPI fields in PTEs and xTLBTRs.

BIT	31-28	27	26	25	24
Identifier	Reserved	UWE7	URE7	SWE7	SRE7
Reset	X	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W

BIT	23	22	21	20	19	18	17	16
Identifier	UWE6	URE6	SWE6	SRE6	UWE5	URE5	SWE5	SRE5
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

BIT	15	14	13	12	11	10	9	8
Identifier	UWE4	URE4	SWE4	SRE4	UWE3	URE3	SWE3	SRE3
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

BIT	7	6	5	4	3	2	1	0
Identifier	UWE2	URE2	SWE2	SRE2	UWE1	URE1	SWE1	SRE1
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

SREx	Supervisor Read Enable x 0 Load operation in supervisor mode not permitted 1 Load operation in supervisor mode permitted
SWEx	Supervisor Write Enable x 0 Store operation in supervisor mode not permitted 1 Store operation in supervisor mode permitted
UREx	User Read Enable x 0 Load operation in user mode not permitted 1 Load operation in user mode permitted
UWEx	User Write Enable x 0 Store operation in user mode not permitted 1 Store operation in user mode permitted

Table 11-4. DMMUPR Field Descriptions

11.4.3 Instruction MMU Control Register (IMMUCR)

The IMMUC register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is a 32 bits wide register.

It provides general control of the IMMU.

BIT	31-10	9-1	0
Identifier	PTBP	Reserved	ITF
Reset	0	X	0
R/W	R/W	R	R/W

ITF	ITLB Flush 0 ITLB ready for operation 1 ITLB flush request/status
PTBP	Page Table Base Pointer N 22-bit pointer to the base of page directory/table

Table 11-5. IMMUCR Field Descriptions

PTB field in xMMUCR1 is required only in implementations with hardware PTE reload support. Implementations that use software TLB reload are not required to implement this field because page table base pointer is stored in a TLB miss exception handler's variable.
ITF is optional and when implemented it flushes entire ITLB.

11.4.4 Instruction MMU Protection Register (IMMUPR)

The IMMUP register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is a 32 bits wide register.

It defines 7 protection groups indexed by PPI fields in PTEs and xTLBTRs.

BIT	31-14	13	12	11	10	9	8
Identifier	Reserved	UXE7	SXE7	UXE6	SXE6	UXE5	SXE5
Reset	X	0	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W	R/W	R/W

BIT	7	6	5	4	3	2	1	0
Identifier	UXE4	SXE4	UXE3	SXE3	UXE2	SXE2	UXE1	SXE1
Reset	0	0	0	0	0	0	0	0

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

SXEx	Supervisor Execute Enable x 0 Instruction fetch in supervisor mode not permitted 1 Instruction fetch in supervisor mode permitted
UXEx	User Execute Enable x 0 Instruction fetch in user mode not permitted 1 Instruction fetch in user mode permitted

Table 11-6. IMMUPR Field Descriptions

11.4.5 Instruction/Data TLB Entry Invalidate Registers (xTLBEIR)

The instruction/data TLB entry invalidate registers are special-purpose registers accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. They are 32 bits wide register in 32-bit implementation and 64 bits wide in 64-bit implementation.

The xTLBEIR is written with the effective address. Corresponding xTLB entry is invalidated in the local processor.

BIT	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets TLB entry inside TLB

Table 11-7. xTLBEIR Field Descriptions

11.4.6 Instruction/Data Translation Lookaside Buffer Match Registers (xTLBMR0-xTLBMR255)

The xTLBM registers are special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. They are 32 bits wide registers.

Together with xTLBT registers they cache translation entries used for translating virtual to physical address. Virtual address is formed from EA generated during instruction fetch or load/store operation, and SR[CID] field. xTLBM registers hold a tag that is compared with the current virtual address generated by the CPU core. Together with the xTLBT registers and match logic they form a core of the xMMU.

BIT	31-10
Identifier	VPN
Reset	X
R/W	R/W

BIT	9-5	5-2	1-0
Identifier	Reserved	CID	PS
Reset	X	0	0
R/W	R	R/W	R/W

PS	Page Size 00 This TLB entry translates 8KB pages 01 This TLB entry translates 16MB pages 10 Reserved 11 Reserved
CID	Context ID 0-15 TLB entry translates for CID
VPN	Virtual Page Number 0-N Number of the virtual frame that must match EA

Table 11-8. xTLBMR Field Descriptions

The CID bits can be hardwired to zero if implementation does not support fast context switching and SR[CID] bits.

11.4.7 Instruction/Data Translation Lookaside Buffer Translate Registers (xTLBTR0-xTLBTR255)

The xTLBT registers are special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. They are 32 bits wide registers.

Together with xTLBM registers they cache translation entries used for translating virtual to physical address. Virtual address is formed from EA generated during instruction fetch or load/store operation, and SR[CID] field. Together with the xTLBM registers and match logic they form a core of the xMMU.

BIT	31-10	9
Identifier	PPN	Reserved
Reset	X	X
R/W	R/W	-

BIT	8-6	5	4	3	2	1	0
Identifier	PPI	D	A	WOM	WBC	CI	CC

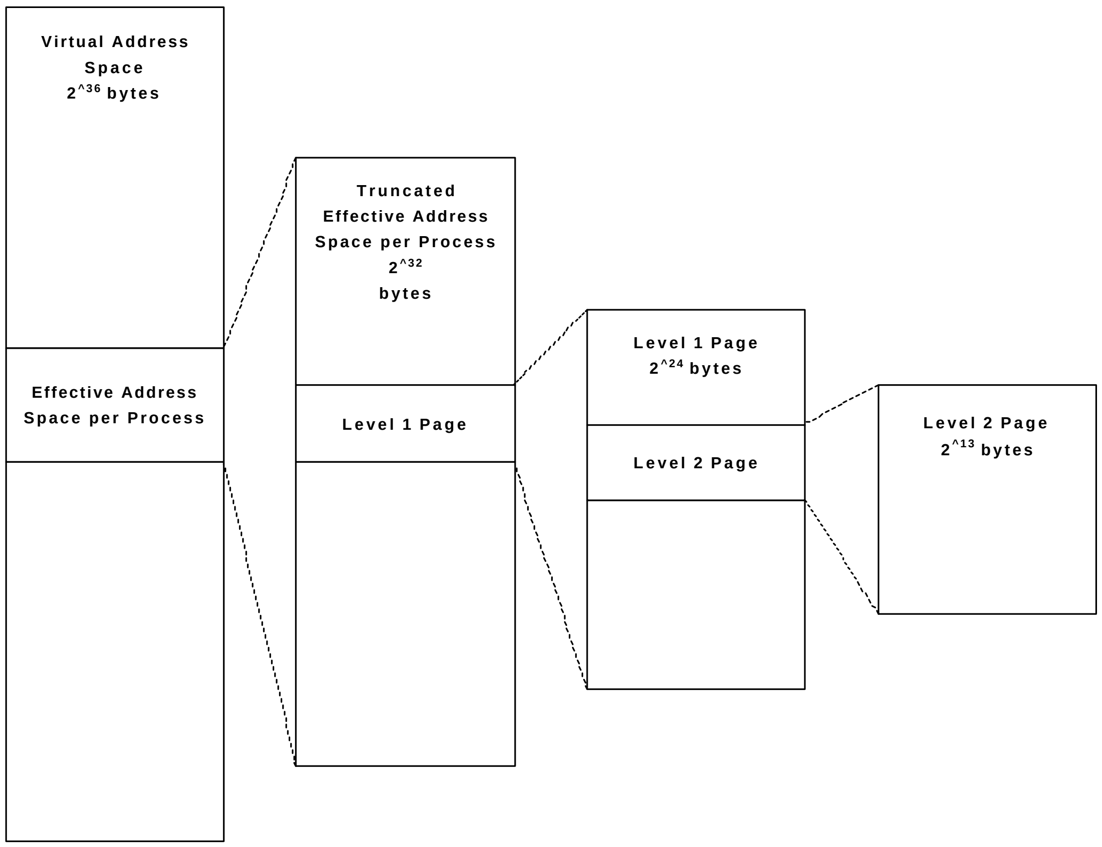
Reset	X	X	X	X	X	X	X
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

CC	Cache Coherency 0 Data cache coherency is not enforced for this page 1 Data cache coherency is enforced for this page
CI	Cache Inhibit 0 Cache is enabled for this page 1 Cache is disabled for this page
WBC	Write-Back Cache 0 Data cache uses write-through strategy for data from this page 1 Data cache uses write-back strategy for data from this page
WOM	Weakly-Ordered Memory 0 Strongly-ordered memory model for this page 1 Weakly-ordered memory model for this page
A	Accessed 0 Page was not accessed 1 Page was accessed
D	Dirty 0 Page was not modified 1 Page was modified
PPI	Page Protection Index 0 TLB entry is invalid 1-7 Selects a group of six bits from a set of seven protection attribute groups in xMMUCR
PPN	Physical Page Number 0-N Number of the physical frame in memory

Table 11-9. xTLBTR Field Descriptions

11.5 Address Translation Mechanism in 32-bit Implementations

Memory in OpenRISC 1000 with 32-bit effective address (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.



Virtual Address Space 2^36 bytes
Effective Address Space per Process
Truncated Effective Address Space per Process 2^32 bytes
Level 1 Page
Level 1 Page 2^24 bytes
Level 2 Page
Level 2 Page 2^13 bytes

Figure 11-2. Memory Divided Into L1 and L2 pages

The first step in page address translation is to append current SR[CID] bits as most significant bits to the 32-bit effective address and combined them into 36-bit virtual address. The 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.

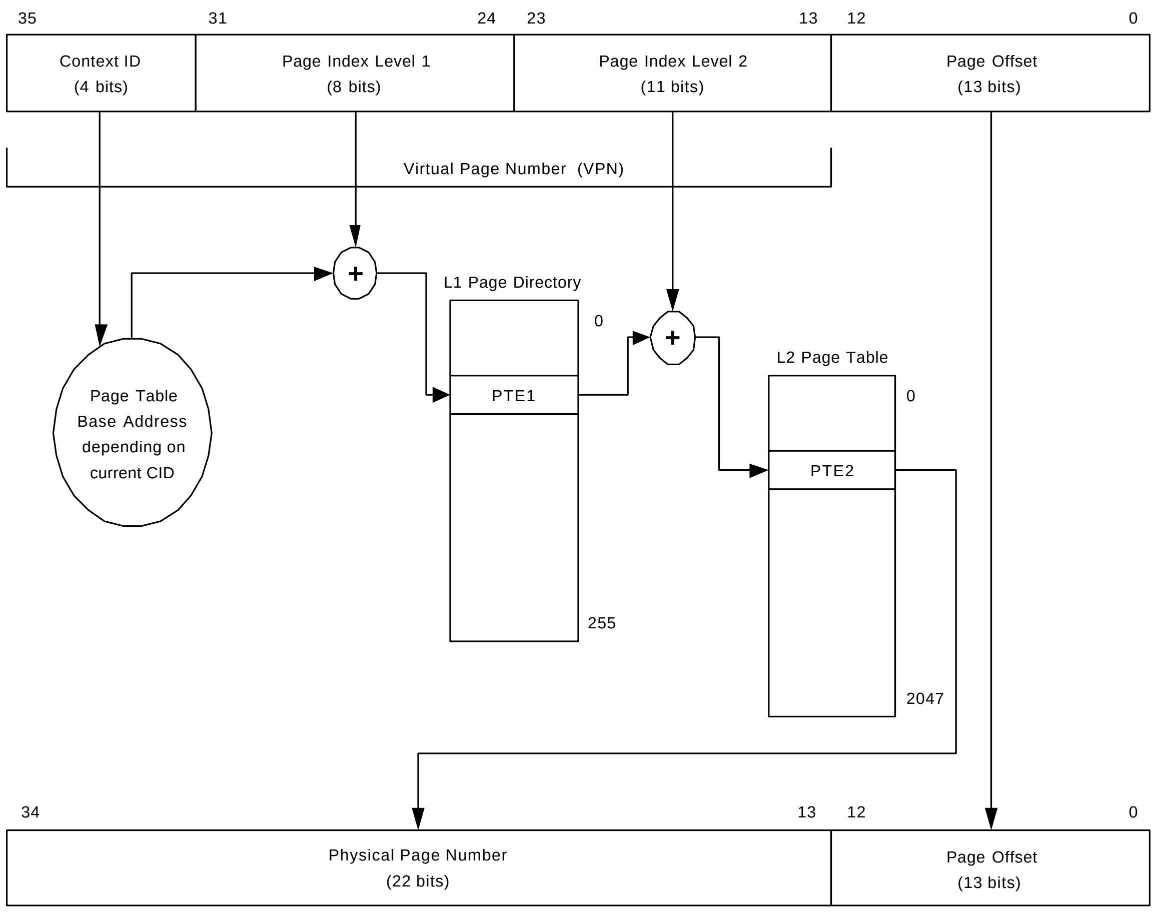


Figure 11-3. Address Translation Mechanism using Two-Level Page Table

Figure 11-3 shows an overview of the two-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 current context's virtual space. The L1 page index is used to index L1 page directory and to retrieve 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 current context's virtual space. The L2 page index is used to index L2 page table and to retrieve 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

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.

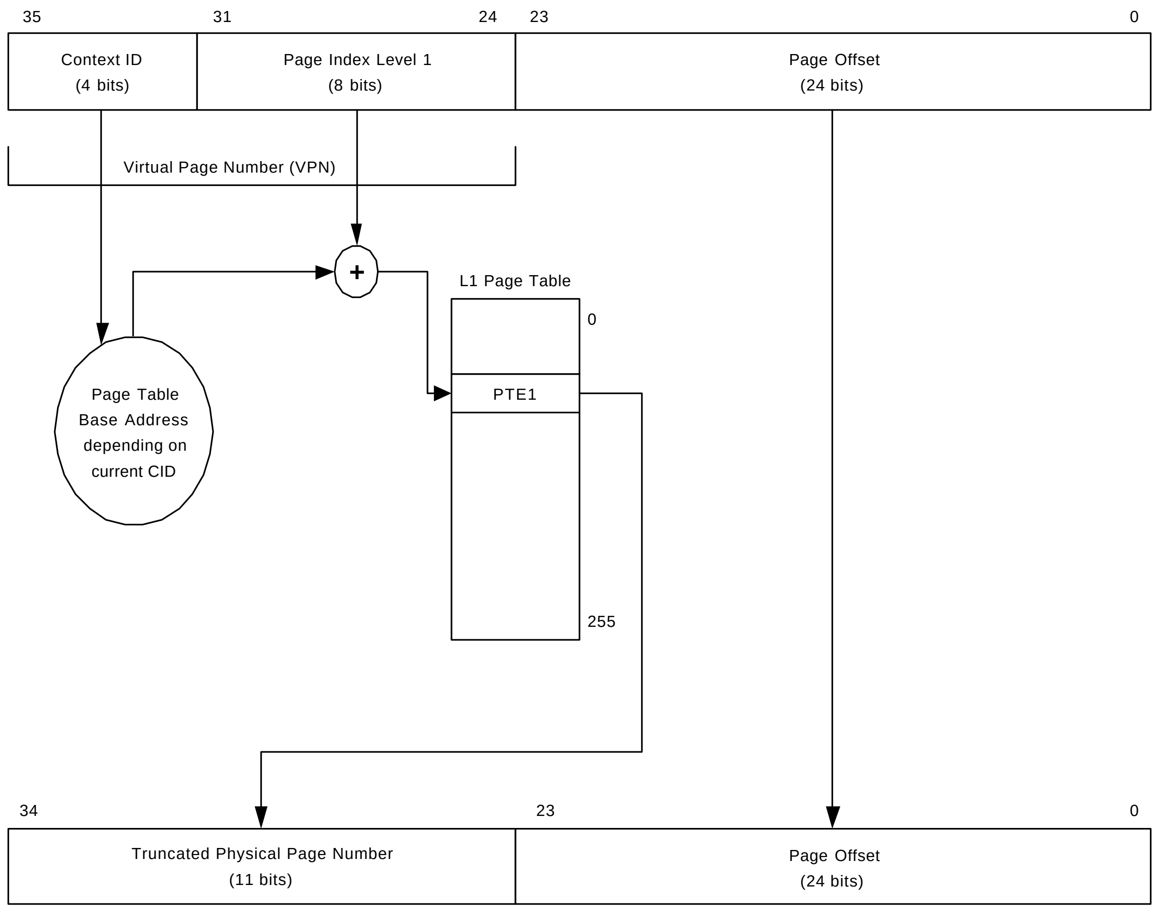


Figure 11-4. Address Translation Mechanism using only L1 Page Table

Figure 11-4 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 current context's virtual space. The L1 page index is used to index L1 page table and to retrieve 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

11.6 Address Translation Mechanism in 64-bit Implementations

Memory in OpenRISC 1000 with 64-bit effective address (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.

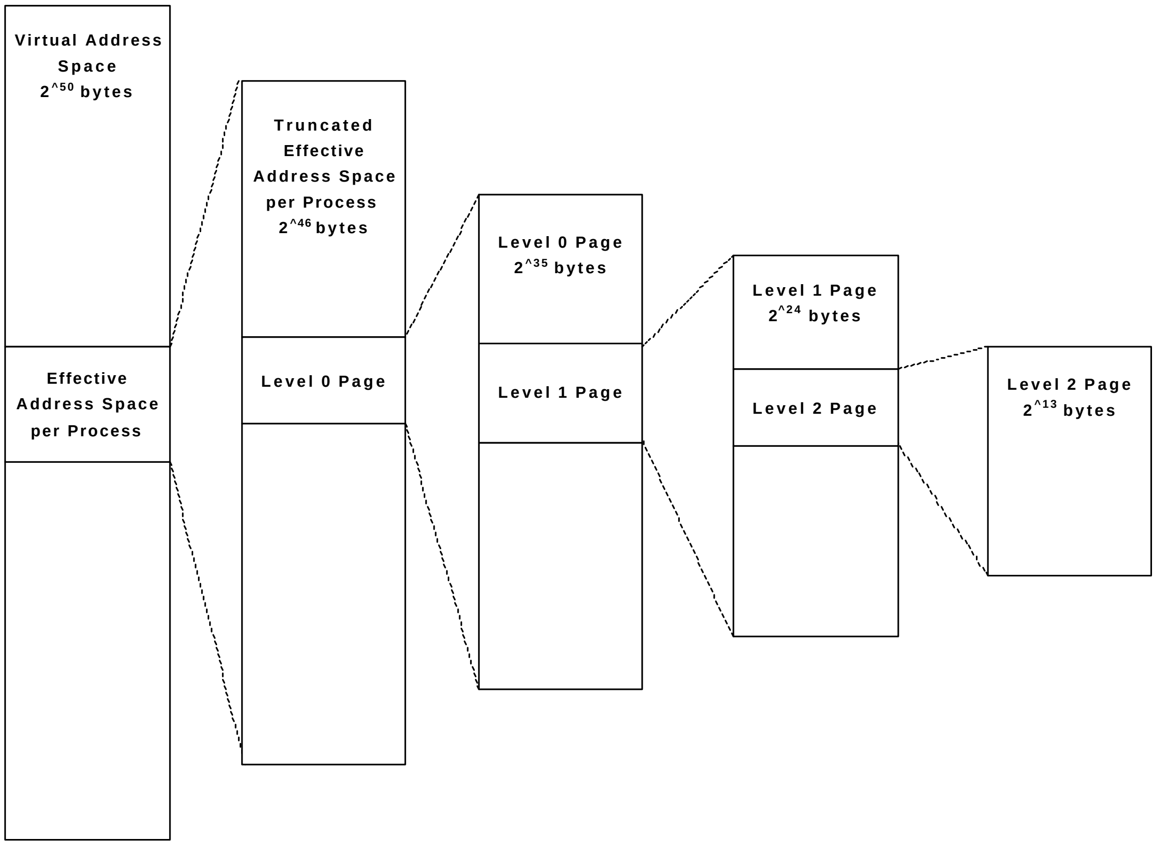
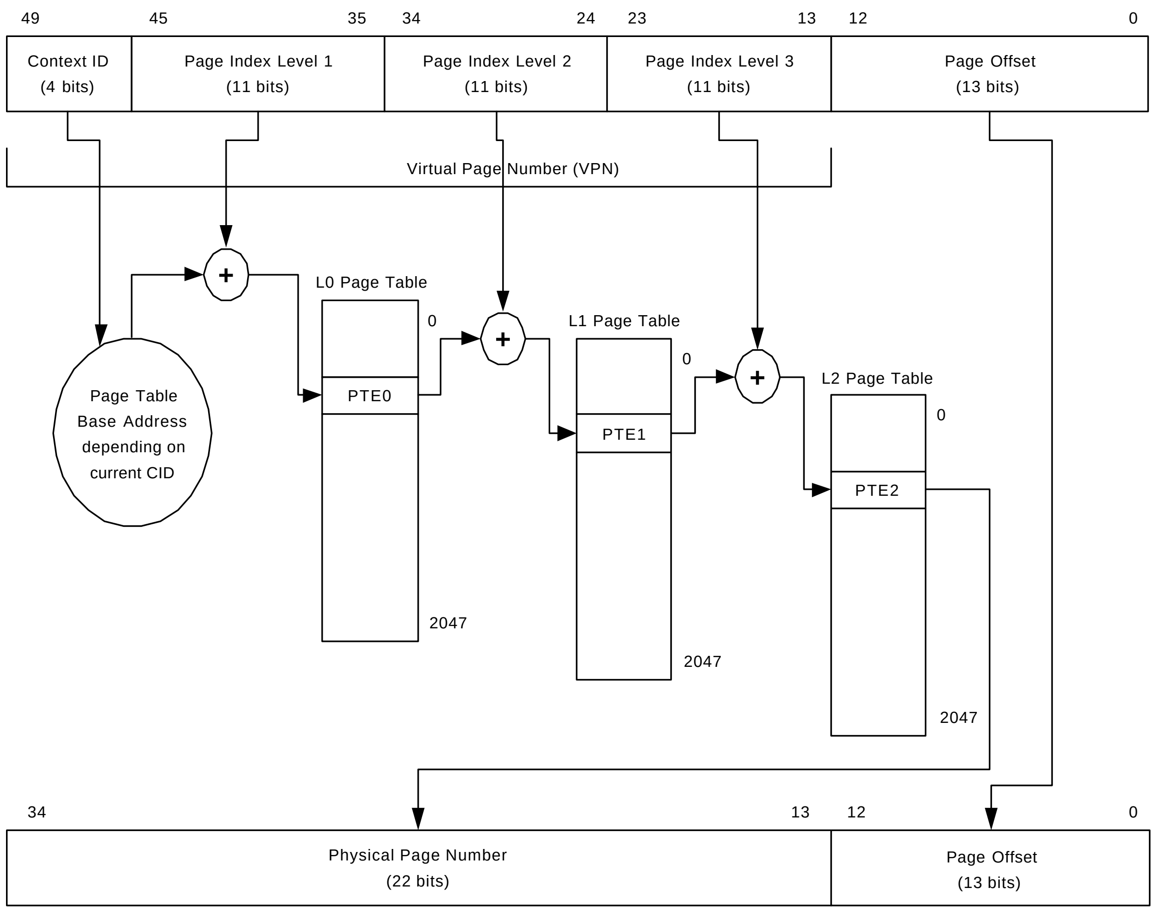

Virtual Address Space 2^50 bytes
Effective Address Space per Process
Truncated Effective Address Space per Process 2^46 bytes
Level 0 Page
Level 0 Page 2^35 bytes
Level 1 Page
Level 1 Page 2^24 bytes
Level 2 Page
Level 2 Page 2^13 bytes

Figure 11-5. Memory Divided Into L0, L1 and L2 pages

The first step in page address translation is truncation of the 64-bit effective address into 46-bit address. Then the current SR[CID] bits are appended as most significant bits. The 50-bit 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.



49
45
35
34
24
23
13
12
0
Context ID (4 bits)
Page Index Level 1 (11 bits)
Page Index Level 2 (11 bits)
Page Index Level 3 (11 bits)
Page Offset (13 bits)
Virtual Page Number (VPN)
Page Table Base Address depending on current CID
L0 Page Table
PTE0
L1 Page Table
PTE1
L2 Page Table
PTE2
0
2047
34
13
12
0
Physical Page Number (22 bits)
Page Offset (13 bits)

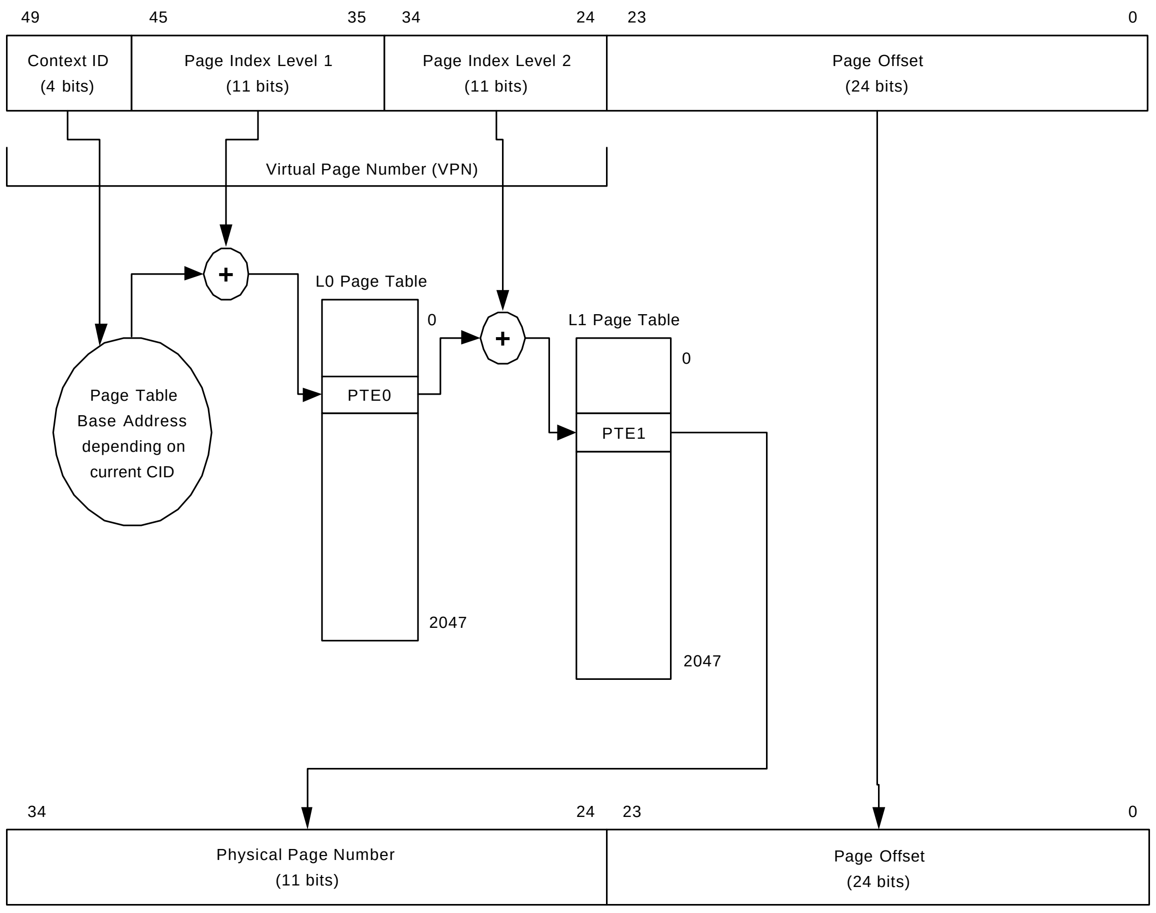
Figure 11-6. Address Translation Mechanism using Three-Level Page Table

Figure 11-6 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 L0 page directory and to retrieve 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 current context's virtual space. The L1 page index is used to index L1 page directory and to retrieve PTE from it, or together with the L0 and 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 current context's virtual space. The L2 page index is used to index L2 page table and to retrieve PTE from it, or together with the L0 and 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 truncated PPN field of the PTE to form the physical address used to access memory

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.



49
45
35
34
24
23
0
Context ID (4 bits)
Page Index Level 1 (11 bits)
Page Index Level 2 (11 bits)
Page Offset (24 bits)
Virtual Page Number (VPN)
+
L0 Page Table
0
+
L1 Page Table
0
Page Table Base Address depending on current CID
PTE0
PTE1
2047
2047
34
24
23
0
Physical Page Number (11 bits)
Page Offset (24 bits)

Figure 11-7. Address Translation Mechanism using Two-Level Page Table

Figure 11-7 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 current context's virtual space. The L0 page index is used to index L0 page directory and to retrieve 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 current context's virtual space. The L1 page index is used to index L1 page table and to retrieve 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

11.7 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 11-10. Protection Attributes

Table 11-10 lists page protection attributes defined in MMU protection registers. For the individual page appropriate strategy out of seven possible strategies programmed in MMU protection registers is selected with the PPI field of the PTE.

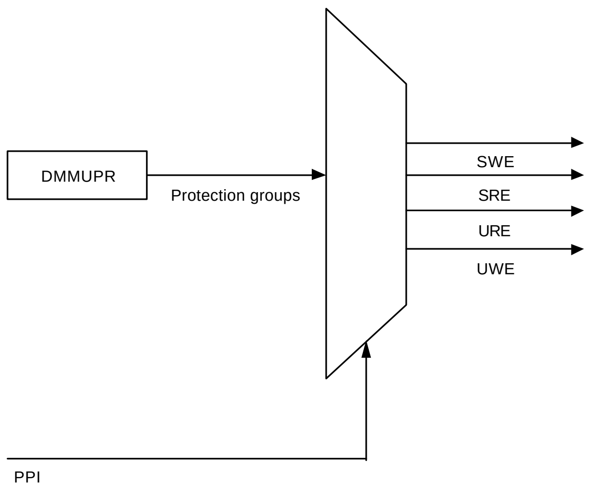


Figure 11-8. Selection of Page Protection Attributes for Data Accesses

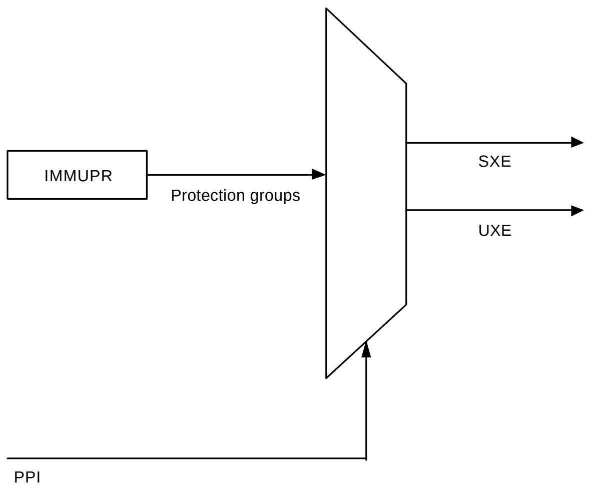


Figure 11-9. Selection of Page Protection Attributes for Instruction Fetch Accesses

11.8 Page Table Entry Definition

Page table entries (PTEs) are generated and placed in page tables in memory by the operating system. PTE is 32 bits wide and is the same for 32-bit and 64-bit OpenRISC 1000 processor implementations.

PTE translates virtual memory area into physical memory area. How much virtual memory is translated depends on which level 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.

31 10	9	7 6	5	4	3	2	1	0
Physical Page Number (22 bits)	L	PP Index (3 bits)	D	A	WOM	WBC	CI	CC

Figure 11-10. Page Table Entry Format

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

Table 11-11. PTE Field Descriptions

11.9 Page Table Search Operation

Implementation may choose to implement page table search operation in hardware or in software. For all page table search operations data addresses are untranslated (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 software is responsible for maintaining accessed and dirty bits in the page tables.

11.10 Page History Recording

Accessed (A) bit and dirty (D) bit 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).

Accessed (A) bit resides both in the PTE in page table and in the copy of PTE in the TLB. Every time when page is accessed by a load, store or instruction fetch operation, accessed bit is set.

If TLB reload is performed in software, then 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 accessed bit should be set or not. Since 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.

Dirty (D) bit resides both in the PTE in page table and in the copy of PTE in the TLB. Every time when page is modified by a store operation, dirty bit is set.

If TLB reload is performed in software, then 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 dirty bit should be set or not. Since 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 dirty bit.

It is up to the operating system to determine when to explicitly clear the dirty bit for a given page.

11.11 Page Table Updates

Updates to the page tables include operatins like adding PTE, deleting PTE and modifying 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 processor reloads PTEs even during update 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.

12 Cache Model and 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 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 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 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.

12.1 Cache Special-Purpose Registers

Table 12-1 summarizes the registers that the operating system uses to manage the cache(s).

For implementations that have unified cache, registers that control data and instruction cache 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	W	Data Cache Block Lock Register

4	0	ICCR	–	R/W	Instruction Cache Control Register
4	1	ICBPR	W	W	Instruction Cache Block PreFetch Register
4	3	ICBIR	W	W	Instruction Cache Block Invalidate Register
4	5	ICBLR	W	W	Instruction Cache Block Lock Register

Table 12-1. Cache Registers

12.1.1 Data Cache Control Register

The data cache control register is special-purpose register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is 32 bits wide register.

The DCCR controls the operation of the data cache.

BIT	31-8	7-0
Identifier	Reserved	EW
Reset	X	0
R/W	R	R/W

EW	Enable Ways 0000 0000 All ways disabled/locked … 1111 1111 All ways enabled/unlocked

Table 12-2. DCCR Field Descriptions

12.1.2 Instruction Cache Control Register

The instruction cache control register is special-purpose register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is 32 bits wide register.

The ICCR controls the operation of the instruction cache.

BIT	31-8	7-0
Identifier	Reserved	EW
Reset	X	0
R/W	R	R/W

EW	Enable Ways 0000 0000 All ways disabled/locked … 1111 1111 All ways enabled/unlocked

Table 12-3. ICCR Field Descriptions

12.2 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 instruction cache coherent with modified instructions in the memory. Typical way to accomplish this:

1. Data cache block write-back (update of the memory)
2. l.sync (wait for update to finish)
3. Instruction cache block invalidate (clear instruction cache block)
4. Flush pipeline

12.2.1 Data Cache Block Prefetch (optional)

The data cache block prefetch register is optional special-purpose register accessible with l.mtspr/l.mfspr instruction pair in both user and supervisor mode. It is 32 bits wide register in 32-bit implementation and 64 bits wide in 64-bit implementation. Implementation may choose not to implement this register and ignore all writes to this register.

The DCBPR is written with the effective address and corresponding block from memory is prefetched into the cache. Memory accesses are not recorded (unlike load or store instructions) and cannot invoke any exception.
Data cache block prefetch is used strictly for improving performance.

BIT	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block

Table 12-4. DCBPR Field Descriptions

12.2.2 Data Cache Block Flush

The data cache block flush register is special-purpose register accessible with l.mtspr/l.mfspr instruction pair in both user and supervisor mode. It is 32 bits wide register in 32-bit implementation and 64 bits wide in 64-bit implementation.

The DCBFR is written with the effective address. If coherency is required then 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 corresponding:

- Unmodified data cache block in the local processor is invalidated.
- Modified data cache block is written back to the memory and invalidated in local processor.
- Missing cache block in the local processor does not cause any action.

BIT	31-0
Identifier	EA
Reset	0
R/W	Write only

EA	Effective Address EA that targets byte inside cache block

Table 12-5. DCBFR Field Descriptions

12.2.3 Data Cache Block Invalidate

The data cache block invalidate register is special-purpose register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is 32 bits wide register in 32-bit implementation and 64 bits wide in 64-bit implementation.

The DCBIR is written with the effective address. If coherency is required then corresponding:

- Unmodified data cache block is invalidated in all processors.
- Modified data cache block is invalidated in all processors.
- Missing data cache block in the local processor causes that data cache blocks in other processors are invalidated.

If coherency is not required then corresponding:

- Unmodified data cache block in the local processor is invalidated.

- Modified data cache block in the local processor is invalidated.
- Missing cache block in the local processor does not cause any action.

BIT	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block

Table 12-6. DCBIR Field Descriptions

12.2.4 Data Cache Block Write-Back

The data cache block write-back register is special-purpose register accessible with l.mtspr/l.mfspr instruction pair in both user and supervisor mode. It is 32 bits wide register in 32-bit implementation and 64 bits wide in 64-bit implementation.

The DCBWR is written with the effective address. If coherency is required then corresponding data cache block in any of the processor is written back to memory if it was modified. If coherency is not required then corresponding data cache block in the local processor is written back to memory if it was modified.

BIT	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block

Table 12-7. DCBWR Field Descriptions

12.2.5 Data Cache Block Lock (optional)

The data cache block lock register is optional special-purpose register accessible with l.mtspr/l.mfspr instruction pair in both user and supervisor mode. It is 32 bits wide register in 32-bit implementation and 64 bits wide in 64-bit implementation.

The DCBLR is written with the effective address. Corresponding data cache block in the local processor is locked.
If all blocks of the same set in all cache ways are locked, then the cache refill may automatically unlock the least-recently used block.

BIT	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block

Table 12-8. DCBLR Field Descriptions

12.2.6 Instruction Cache Block Prefetch (optional)

The instruction cache block prefetch register is optional special-purpose register accessible with l.mtspr/l.mfspr instruction pair in both user and supervisor mode. It is 32 bits wide register in 32-bit implementation and 64 bits wide in 64-bit implementation. Implementation may choose not to implement this register and ignore all writes to this register.

The ICBPR is written with the effective address and corresponding block from memory is prefetched into the instruction cache.
Instruction cache block prefetch is used strictly for improving performance.

BIT	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block

Table 12-9. ICBPR Field Descriptions

12.2.7 Instruction Cache Block Invalidate

The instruction cache block invalidate register is special-purpose register accessible with l.mtspr/l.mfspr instruction pair in both user and supervisor mode. It is 32 bits wide register in 32-bit implementation and 64 bits wide in 64-bit implementation.

The ICBIR is written with the effective address. If coherency is required then corresponding instruction cache blocks in all processors are invalidated. If coherency is not required then corresponding instruction cache block is invalidated in the local processor.

BIT	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block

Table 12-10. ICBIR Field Descriptions

12.2.8 Instruction Cache Block Lock (optional)

The instruction cache block lock register is optional special-purpose register accessible with l.mtspr/l.mfspr instruction pair in both user and supervisor mode. It is 32 bits wide register in 32-bit implementation and 64 bits wide in 64-bit implementation.

The ICBLR is written with the effective address. 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.

BIT	31-0
Identifier	EA
Reset	0
R/W	Write Only

EA	Effective Address EA that targets byte inside cache block

Table 12-11. ICBLR Field Descriptions

12.3 Cache/Memory Coherency

The primary role of the cache coherency 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.

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 memory management unit, it can 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. 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.

12.3.1 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, 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.

12.3.2 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 operating system sets 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.sync**.

12.3.3 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 system uses multilevel hierarchy caches, 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**. 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 local data cache. Data from 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 requires cache snooping hardware support in data cache controllers to gurantee cache coherency.

13 Debug Unit

This chapter describes the OpenRISC 1000 debug facility. Debug unit assists software developers to debug 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. Breakpoint unlike watchpoints also suspends execution of the current program-flow and starts breakpoint exception. Breakpoint is a result of the watchpoints.

13.1 Features

OpenRISC 1000 architecture defines eight sets of debug registers. Additional debug register sets can be defined by the implementation itself. Debug unit is optional and its implementation is specified 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 generate a breakpoint
- 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 breakpoints.

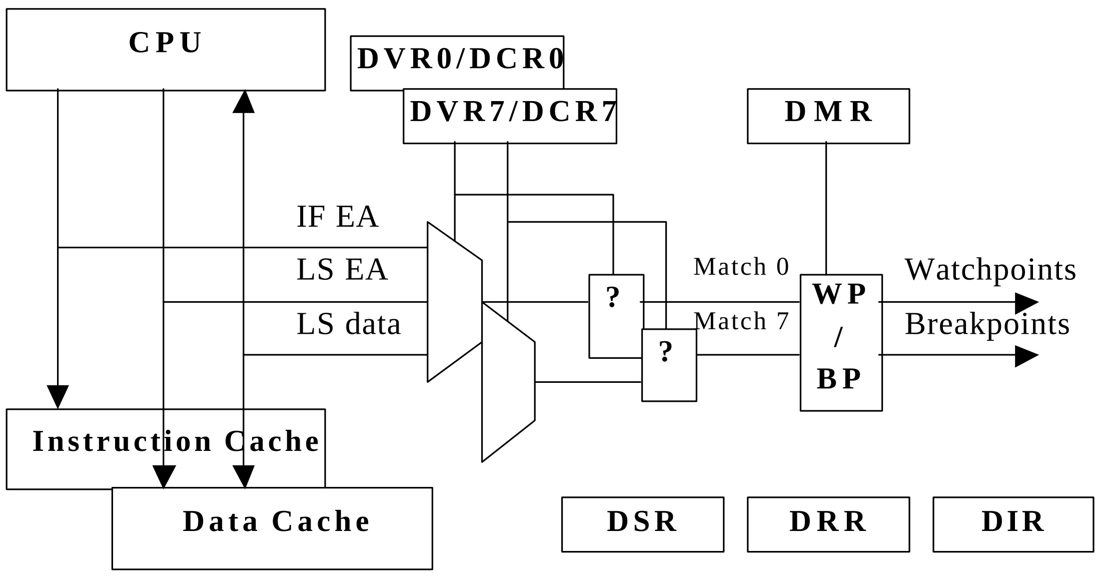


Figure 13-1. Block Diagram of Debug Support

13.2 Debug Value Registers (DVR0-DVR7)

The debug value registers are special-purpose supervisor-level registers accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. They are 32 bits wide registers.

The DVRs are programmed with the watchpoint/breakpoint 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 or breakpoint is generated.

BIT	31-0
Identifier	VALUE
Reset	0
R/W	R/W

VALUE	Watchpoint/Breakpoint Address/Data

Table 13-1. DVR Field Descriptions

13.3 Debug Control Registers (DCR0-DCR7)

The debug control registers are special-purpose supervisor-level registers accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. They are 32 bits wide registers.

The DCRs are programmed with the watchpoint/breakpoint 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/W

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

Table 13-2. DCR Field Descriptions

13.4 Debug Mode Register 1 (DMR1)

The debug mode register 1 is special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. It is 32 bits wide register.

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	24	23	22	21-20	19-18	17-16
Identifier	Reserved	DXFW	BT	ST	CW10	CW9	CW8
Reset	X	0	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W	R/W	R/W

BIT	15-14	13-12	11-10	9-8	7-6	5-4	3-2	1-0
Identifier	CW7	CW6	CW5	CW4	CW3	CW2	CW1	CW0
Reset	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

CW0	Chain Watchpoint 0 00 Watchpoint 0 = Match 0 01 Watchpoint 0 = Match 0 10 Watchpoint 0 = Match 0 11 Reserved
CW1	Chain Watchpoint 1 00 Watchpoint 1 = Match 1 01 Watchpoint 1 = Match 1 & Watchpoint 0 10 Watchpoint 1 = Match 1 | Watchpoint 0 11 Reserved
CW2	Chain Watchpoint 2 00 Watchpoint 2 = Match 2 01 Watchpoint 2 = Match 2 & Watchpoint 1 10 Watchpoint 2 = Match 2 | Watchpoint 1 11 Reserved
CW3	Chain Watchpoint 3 00 Watchpoint 3 = Match 3 01 Watchpoint 3 = Match 3 & Watchpoint 2 10 Watchpoint 3 = Match 3 | Watchpoint 2 11 Reserved
CW4	Chain Watchpoint 4 00 Watchpoint 4 = Match 4 01 Watchpoint 4 = Match 4 & Watchpoint 3 10 Watchpoint 4 = Match 4 | Watchpoint 3 11 Reserved
CW5	Chain Watchpoint 5 00 Watchpoint 5 = Match 5 01 Watchpoint 5 = Match 5 & Watchpoint 4 10 Watchpoint 5 = Match 5 | Watchpoint 4 11 Reserved

CW6	Chain Watchpoint 6 00 Watchpoint 6 = Match 6 01 Watchpoint 6 = Match 6 & Watchpoint 5 10 Watchpoint 6 = Match 6 | Watchpoint 5 11 Reserved
CW7	Chain Watchpoint 7 00 Watchpoint 7 = Match 7 01 Watchpoint 7 = Match 7 & Watchpoint 6 10 Watchpoint 7 = Match 7 | Watchpoint 6 11 Reserved
CW7	Chain Watchpoint 7 00 Watchpoint 7 = Match 7 01 Watchpoint 7 = Match 7 & Watchpoint 6 10 Watchpoint 7 = Match 7 | Watchpoint 6 11 Reserved
CW8	Chain Watchpoint 8 00 Watchpoint 8 = Watchpoint counter 0 match 01 Watchpoint 8 = Watchpoint counter 0 match & Watchpoint 7 10 Watchpoint 8 = Watchpoint counter 0 match | Watchpoint 7 11 Reserved
CW9	Chain Watchpoint 9 00 Watchpoint 9 = Watchpoint counter 1 match 01 Watchpoint 9 = Watchpoint counter 1 match & Watchpoint 8 10 Watchpoint 9 = Watchpoint counter 1 match | Watchpoint 8 11 Reserved
CW10	Chain Watchpoint 10 00 Watchpoint 10 = external watchpoint 01 Watchpoint 10 = external watchpoint & Watchpoint 9 10 Watchpoint 10 = external watchpoint | Watchpoint 9 11 Reserved
ST	Single-step Trace 0 Single-step trace disabled 1 Every executed instruction causes breakpoint exception
BT	Branch Trace 0 Branch trace disabled 1 Every executed branch instruction causes breakpoint exception
DXFW	Disable eXternal Force Watchpoint 0 External debugger can force watchpoint condition 1 Input from external debugger is ignored

Table 13-3. DMR1 Field Descriptions

13.5 Debug Mode Register 2(DMR2)

The debug mode register 2 is special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. It is 32 bits wide register.

The DMR2 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	31-24	23-13	12-2	1	0
Identifier	Reserved	WGB	AWCP	WCE1	WCE 0
Reset	X	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W

WCE0	Watchpoint Counter Enable 0 0 Counter 0 disabled 1 Counter 0 enabled
WCE1	Watchpoint Counter Enable 1 0 Counter 1 disabled 1 Counter 1 enabled
AWPC	Assign Watchpoints to Counter 000 0000 0000 All Watchpoints increment counter 0 000 0000 0001 Watchpoint 0 increments counter 1 … 000 0000 1111 First four watchpoints increment counter 1, rest increment counter 0 … 111 1111 1111 All watchpoints increment counter 1
WGB	Watchpoints Generating Breakpoint 000 0000 0000 Breakpoint disabled 000 0000 0001 Watchpoint 0 generates breakpoint … 001 0000 0000 Watchpoint counter 0 generates breakpoint … 111 1111 1111 All watchpoints generate breakpoint

Table 13-4. DMR2 Field Descriptions

13.6 Debug Watchpoint Counter Register (DWCR0-DWCR1)

The debug watchpoint counter registers are special-purpose supervisor-level registers accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. They are 32 bits wide registers.

The DWCRs contain 16-bit counters that count watchpoints programmed in DMR. Value in DWCR can be accessed by the resident debug software or by the development interface. DWCRs also contain match value. When counter reaches match value, watchpoint is generated.

BIT	31-16	15-0
Identifier	MATCH	COUNT
Reset	0	0
R/W	R/W	R/W

COUNT	Number of watchpoints programmed in DMR N 16-bit counter of generated watchpoints assigned to this counter
MATCH	N 16-bit value that when matched generates a watchpoint

Table 13-5. DWCR Field Descriptions

13.7 Debug Stop Register (DSR)

The debug stop register is special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. It is 32 bits wide register.

The DSR is 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	31-13	12	11	10	9	8	7
Identifier	Reserved	BE	SCE	RE	IME	DME	HPINTE
Reset	X	0	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W	R/W	R/W

BIT	6	5	4	3	2	1	0
Identifier	IIE	AE	LPINTE	IPFE	DPFE	BUSEE	RSTE
Reset	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

RSTE	Reset Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
BUSEE	Bus Error Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
DPFE	Data Page Fault Exception

	0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
IPFE	Instruction Page Fault Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
LPINTE	Low Priority Interrupt Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
AE	Alignment Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
IIE	Illegal Instruction Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
HPINTE	High Priority Interrupt Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
DME	DTLB Miss Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
IME	ITLB Miss Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
RE	Range Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
SCE	Instruction TLB Miss Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface
BE	Breakpoint Exception 0 This exception does not transfer control to the development I/F 1 This exception transfers control to the development interface

Table 13-6. DSR Field Descriptions

13.8 Debug Reason Register (DRR)

The debug reason register is special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. It is 32 bits wide register.

The DRR is specifies which event caused the core to stop the execution of program flow and turned over control to the development interface. It should be cleared by the resident debug software or by the development interface.

BIT	31-13	12	11	10	9	8	7
Identifier	Reserved	BE	SCE	RE	IME	DME	HPINTE
Reset	X	0	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W	R/W	R/W

BIT	6	5	4	3	2	1	0
Identifier	IIE	AE	LPINTE	IPFE	DPFE	BUSEE	RSTE
Reset	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

RSTE	Reset Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
BUSEE	Bus Error Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
DPFE	Data Page Fault Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
IPFE	Instruction Page Fault Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
LPINTE	Low Priority Interrupt Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
AE	Alignment Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
IIE	Illegal Instruction Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
HPINTE	High Priority Interrupt Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
DME	DTLB Miss Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
IME	ITLB Miss Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
RE	Range Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface
SCE	Instruction TLB Miss Exception 0 This exception did not transfer control to the development I/F

	1 This exception transfered control to the development interface
BE	Breakpoint Exception 0 This exception did not transfer control to the development I/F 1 This exception transfered control to the development interface

Table 13-7. DRR Field Descriptions

13.9 Debug Instruction Register (DIR)

The debug instruction register is special-purpose read-only supervisor-level register accessible with l.mfspr instruction in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. It is 32 bits wide register.

The DIR holds the next instruction to be executed by the core. It can be programmed only by the development interface. Instruction in the DIR is executed only once and the register must be set with the new instruction to increment the program flow.

BIT	31-0
Identifier	II
Reset	0
R/W	R

II	Induced Instruction

Table 13-8. DIR Field Descriptions

14 Performance Counters Unit

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.

Performance counters unit can be used for the following:

- To improve performance in many computer systems by developing better application level algorithms, more optimal operating system routines and for improvements in hardware architecture of these systems (memory subsystems).
- To improve future OpenRISC implementations and add future enhancements to the OpenRISC architecture.
- To help system developers debug and test their systems.

14.1 Features

OpenRISC 1000 architecture defines eight performance counters. Additional performance counters can be defined by implementation itself. Performance counters unit is optional and its implementation is specified by UPR[PCUP] bit.

- Optional implementation.
- Eight architecture defined performance counters
- Eight custom performance counters
- Programmable counting conditions.

14.2 Performance Counters Count Registers (PCCR0-PCCR7)

The performance counters count registers are a special-purpose supervisor-level registers accessible with l.mtspr/l.mfspr instruction pair in supervisor mode. Read access in user mode is possible, if it is enabled in PCMR0[UMRA]. They are 32 bits wide registers.

They count number of events programmed in PCMR registers.

BIT	31-0
Identifier	COUNT
Reset	0
R/W	R/W

COUNT	Event counter

Table 14-1. PCCR0 Field Descriptions

14.3 Performance Counters Mode Registers (PCMR0-PCMR7)

The performance counters mode registers are a special-purpose supervisor-level registers accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. They are 32 bits wide registers.

They define which events the performance counters count.

BIT	31-26	25-15	14	13	12	11	10
Identifier	Reserved	WPE	DDS	ITL BM	DTL BM	BS	LSU S
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	CIU M	CIS M	UM RA	CP
Reset	0	0	0	0	0	0	0	0	0	1
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R

CP	Counter Present 0 Counter not present 1 Counter present
UMRA	User Mode Read Access 0 Read of PCCR in user mode returns 0 1 Read of PCCR in user mode returns PCCR value
CISM	Count in Supervisor Mode 0 Counter disabled in supervisor mode 1 Counter counts events in supervisor mode
CIUM	Count in User Mode 0 Counter disabled in user mode 1 Counter counts events in user mode
LA	Load Access event 0 Event ignored 1 Count load accesses
SA	Store Access event 0 Event ignored 1 Count store accesses

IF	Instruction Fetch event 0 Event ignored 1 Count instruction fetches
DCM	Data Cache Miss event 0 Event ignored 1 Count data cache missed
ICM	Instruction Cache Miss event 0 Event ignored 1 Count instruction cache misses
IFS	Instruction Fetch Stall event 0 Event ignored 1 Count instruction fetch stalls
LSUS	LSU Stall event 0 Event ignored 1 Count LSU stalls
BS	Branch Stalls event 0 Event ignored 1 Count branch stalls
DTLBM	DTLB Miss event 0 Event ignored 1 Count DTLB misses
ITLBM	ITLB Miss event 0 Event ignored 1 Count ITLB misses
DDS	Data Dependency Stalls event 0 Event ignored 1 Count data dependency stalls
WPE	Watchpoint Events 000 0000 0000 All watchpoint events ignored 000 0000 0001 Watchpoint 0 counted … 111 1111 1111 All watchpoints counted

Table 14-2. PCMR Field Descriptions

15 Power Management

This chapter describes the OpenRISC 1000 power management facility. Power management facility is optional and implementation may choose, which features to implement, and which not.

Note that this chapter describes architectural control of power management from the perspective of the programming model. As such, it does not describes a technology specific optimizations or implementation techniques.

15.1 Features

OpenRISC 1000 architecture defines four architectural features for minimizing power consumption:

- slow down feature
- doze mode
- sleep mode
- dynamic clock gating feature

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 a power consumption is reduced.
Slow down feature is software controlled with the 4-bit value in PMR[SDF]. 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 timer. However other on-chip blocks 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 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.

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.

15.2 Power Management Register (PMR)

The power management register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is 32 bits wide register.

PMR is used to enable or disable power management features and modes.

BIT	31-7	6	5	4	3-0
Identifier	Reserved	DCGE	SME	DME	SDF
Reset	X	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W

SDF	Slow Down Factor 0 Full speed 1-15 Logarithmic clock frequency reduction
DME	Doze Mode Enable 0 Doze mode not enabled 1 Doze mode enabled
SME	Sleep Mode Enable 0 Sleep mode not enabled 1 Sleep mode enabled
DCGE	Dynamic Clock Gating Enable 0 Dynamic clock gating not enabled 1 Dynamic clock gating enabled

Table 15-1. PMR Field Descriptions

16 Programmable Interrupt Controller

This chapter describes the OpenRISC 1000, level one programmable interrupt controller. Interrupt controller facility is optional and implementation may chose to implement it or not. If it is not implemented, interrupt inputs 0 and 1 are directly connected to high and low priority interrupt exception inputs.

Programmable interrupt controller has three special-purpose registers and 32 interrupt inputs. Interrupt input 0 and 1 are always enabled and connected to high and low priority interrupt input, respectively.
30 other interrupt inputs can be masked and assigned low or high priority through programming special-purpose registers.

16.1 Features

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

- Unmaskable interrupt input 0 connected to high priority interrupt
- Unmaskable interrupt input 1 connected to low priority interrupt
- From 0 to 30 maskable interrupt inputs with programmable priority

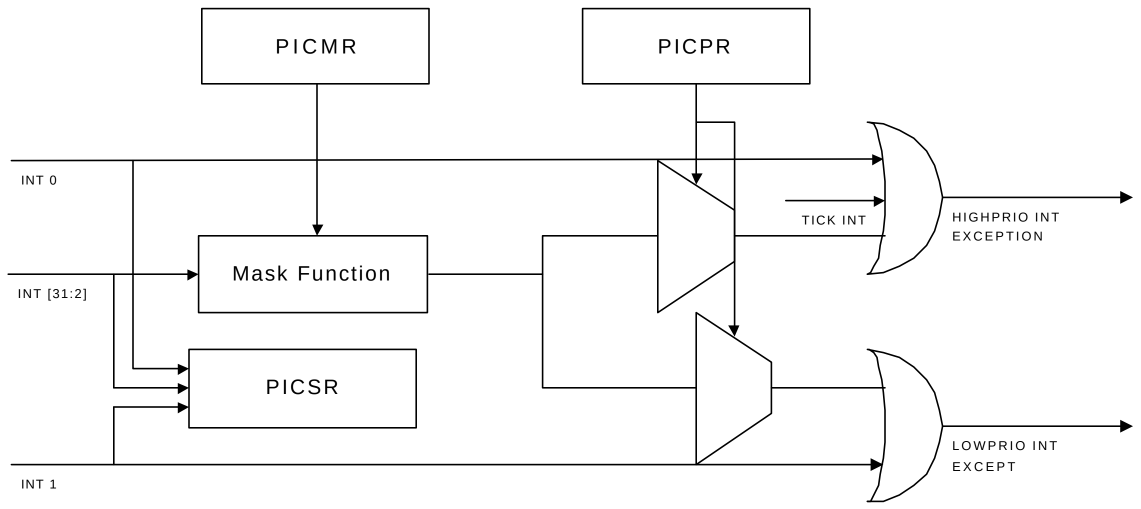


Figure 16-1. Programmable Interrupt Controller Block Diagram

16.2 PIC Mask Register (PICMR)

The interrupt controller mask register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is 32 bits wide register.

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

BIT	31-2	1-0
Identifier	IUM	Reserved
Reset	X	X
R/W	R	R

IUM	Interrupt UnMask 0x00000000 All interrupts are masked 0x00000001 Interrupt input 2 is enabled, all others are masked … 0x3FFFFFFF All interrupt inputs are enabled

Table 16-1. PICMR Field Descriptions

16.3 PIC Priority Register (PICPR)

The interrupt controller priority register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is 32 bits wide register.

PICPR is used to assign low or high priority to 30 interrupt sources.

BIT	31-2	1-0
Identifier	IPRIO	Reserved
Reset	X	X
R/W	R	R

IPRIO	Interrupt Priority 0x00000000 All interrupts are low priority 0x00000001 Interrupt input 2 is high priority, all others are low priority … 0x3FFFFFFF All interrupt inputs are high priority

Table 16-2. PICPR Field Descriptions

16.4 PIC Status Register (PICSR)

The interrupt controller status register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is 32 bits wide register.

PICSR is used to determine status of each interrupt input. Bits in PICSR represent status of the interrupt inputs and the actual interrupt must be cleared in the device which is source of the interrupt.

BIT	31-0
Identifier	IS
Reset	X
R/W	R

IS	Interrupt Status 0x00000000 All interrupts are inactive 0x00000001 Interrupt input 0 is pending … 0xFFFFFFFF All interrupts are pending

Table 16-3. PICSR Field Descriptions

17 Tick Timer Facility

This chapter describes the OpenRISC 1000 tick timer facility. It is optional and implementation may chose to implement it or not.

Tick timer is used to schedule operating system and user tasks on regular time basis or as a high precision time reference.

Tick timer facility is enabled with TTCR[TTE]. TTIR is incremented with each clock cycle and a high priority interrupt can be asserted whenever lower 28 bits of TTIR match TTCR[TP] and TTCR[IE] is set.
TTIR restarts counting from zero when match event happens and TTCR[SR] is cleared. If TTCR[SR] is set, TTIR is halted when match event happens and TTIR must be cleared (writing 1 to TTCR[SR]) to start counting again from zero.

17.1 Features

OpenRISC 1000 architecture defines 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 or restartable timer

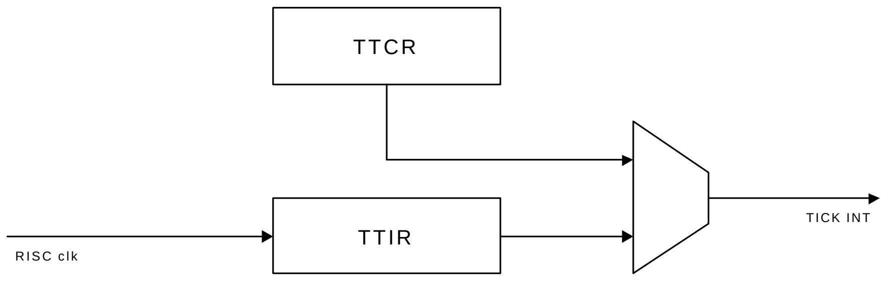


Figure 17-1. Tick Timer Block Diagram

17.2 Tick Timer Control Register (TTCR)

The tick timer control register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is 32 bits wide register.

TTCR is programmed with the time period of the tick timer as well as with control bits that control operation of the tick timer.

BIT	31	30	29	28	27-0
Identifier	TTE	SR	IE	IP	TP
Reset	0	0	0	0	X
R/W	R/W	R/W	R/W	R	R/W

TP	Time Period 0x00000000 Shortest comparison time period … 0xFFFFFFF Longest comparison time period
IP	Interrupt Pending 0 Tick timer interrupt is not pending 1 Tick timer interrupt pending (cleared with every ready of the register)
IE	Interrupt Enable 0 Tick timer does not generate an interrupt 1 Tick timer generates an interrupt after timer expires
SR	Single Run 0 Timer is restarted automatically after expiration 1 Timer is not restarted after expiration (writing 1 into TTCR[SR] will restart it)
TTE	Tick Timer Enable 0 Tick timer is disabled 1 Tick timer is enabled

Table 17-1. TTCR Field Descriptions

17.3 Tick Timer Incrementing Register (TTIR)

The tick timer incrementing register is a special-purpose register accessible with l.mtspr/l.mfspr instruction pair in supervisor mode and as read-only register in user mode. It is 32 bits wide register.

TTIR holds the current value of the timer.

BIT	31-0
Identifier	CNT

Reset	0
R/W	R/W

CNT	Count 32-bit incrementing counter

Table 17-2. TTIR Field Descriptions

18 OpenRISC 1000 Implementations

18.1 Overview

Implementations of the OpenRISC 1000 architecture come in different configurations and version releases.
Version and unit present registers both identify for which version/release it goes and what is the configuration of it. Detailed configuration of each unit is available in unit's own registers.

18.2 Version Register (VR)

The version register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is 32 bits wide register.

It identifies the version (model) and revision level of the OpenRISC 1000 processor. It also specifies possible standard template on which this implementation is based.

BIT	31-16	15-6	5-0
Identifier	VER	Reserved	REV
Reset	-	X	-
R/W	R	R	R

REV	Revision 0..63 A 6-bit number that identifies various releases of a particular version. This number is changed for each revision of the device.
VER	Version 0..0xFFFF A 16-bit number that identifies a particular processor version and version of the OpenRISC architecture. Values below 0x1000 and above 0x1FFF are illegal for OpenRISC 1000 processor implementations.

Table 18-1. VR Field Descriptions

18.3 Unit Present Register (UPR)

The unit present register is a special-purpose supervisor-level register accessible with l.mtspr/l.mfspr instruction pair only in supervisor mode. It is 32 bits wide register.

It identifies the present units in the processor. It has a bit for each possible unit or functionality. Lower sixteen bits identify present units defined in the OpenRISC 1000 architecture. Upper sixteen bits define present custom units.

BIT	31-16	14	13	12	11	10	9	8
Identifier	CUP	SRP	PICP	DUP	PCUP	OV64P	OV32P	OF64P
Reset	-	-	-	-	-	-	-	-
R/W	R	R	R	R	R	R	R	R

BIT	7	6	5	4	3	2	1	0
Identifier	OF32P	OB64P	OB32P	IMP	DMP	ICP	DCP	UP
Reset	-	-	-	-	-	-	-	-
R/W	R	R	R	R	R	R	R	R

UP	UPR Present 0 UPR is not present 1 UPR is present
DCP	Data Cache Present 0 Unit is not present 1 Unit is present
ICP	Instruction Cache Present 0 Unit is not present 1 Unit is present
DMP	Data MMU Present 0 Unit is not present 1 Unit is present
IMP	Instruction MMU Present 0 Unit is not present 1 Unit is present
OB32P	ORBIS32 Present 0 Not supported 1 Supported
OB64P	ORBIS64 Present 0 Not supported 1 Supported
OF32P	ORFPX32 Present 0 Not supported 1 Supported
OF64P	ORFP64P Present 0 Supported 1 Not supported
OV32P	ORVDX32 Present

	0 Not supported 1 Supported
OV64P	ORVDX64 Present 0 Not supported 1 Supported
PCUP	Performance Counters Unit Present 0 Unit is not present 1 Unit is present
DUP	Debug Unit Present 0 Unit is not present 1 Unit is present
PICP	Programmable Interrupt Controller Present 0 Unit is not present 1 Unit is present
TTP	Tick Timer Present 0 Unit is not present 1 Unit is present
SRP	Shadowed Registers Present 0 Shadowed registers not present 1 Shadowed registers present
CUP	Custom Units Present

Table 18-2. UPR Field Descriptions

19 Application Binary Interface

19.1 Data Representation

19.1.1 Fundamental Types

Scalar types in ISO/ANSI C language are based on memory operands definitions from chapter “Addressing Modes and Operand Conventions” on page 18. Similar relations between architecture and language types can be used for any other language.

TYPE	C TYPE	SIZEOF	ALIGNMENT (BYTES)	OPENRISC EQUIVALENT
Integral	Char Signed char	1	1	Signed byte
	Unsigned char	1	1	Unsigned byte
	Short Signed short	2	2	Signed halfword
	Unsigned short	2	2	Unsigned halfword
	Int Signed int Long Signed long Enum	4	4	Signed singleword
	Unsigned int	4	4	Unsigned singleword
	Long long Signed long long	8	8	Signed doubleword
	Unsigned long long	8	8	Unsigned doubleword
Pointer	Any-type * Any-type (*) ()	4	4	Unsigned singleword
Floating-point	Float	4	4	Single precision float
	Double	8	8	Double precision float
	Long double	16	8	Quad precision float

Table 19-1. Scalar Types

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 19-2. For vector assignments both side of assignment must be of the same vector type.

VECTOR TYPE	SIZEOF	ALIGNMENT (BYTES)	OPENRISC EQUIVALENT
Vector char Vector signed char	8	8	Vector of signed bytes
Vector unsigned char	8	8	Vector of unsigned bytes
Vector short Vector signed short	8	8	Vector of signed halfwords
Vector unsigned short	8	8	Vector of unsigned halfwords
Vector int Vector signed int Vector long Vector signed long	8	8	Vector of signed singlewords
Vector unsigned int	8	8	Vector of unsigned singlewords
Vector float	8	8	Vector of single-precisions

Table 19-2. Vector Types

For alignment restrictions of all types see chapter "Addressing Modes and Operand Conventions" on page 20.

19.1.2 Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned element.

- An array uses alignment of its elements.
- Structures and unions can require padding to meet alignment restrictions. Each element is assigned to lowest aligned address.

```
struct {
  char   C;
};
```

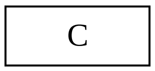


Figure 19-1. Byte aligned, sizeof is 1

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

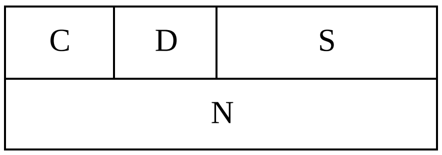


Figure 19-2. No padding, sizeof is 8

```
struct {
  char   C;
  double D;
  short  S;
};
```

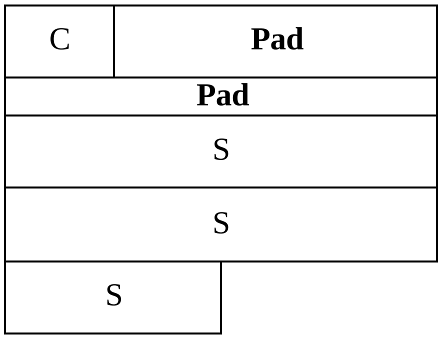


Figure 19-3. Padding, sizeof is 18

19.1.3 Bit-fields

C structure and union definitions can have elements defined by a specified number of bits. Table 19-3 describes valid bit-field types and their ranges.

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

Table 19-3. Bit-Field Types and Ranges

Bit-fields follow the same alignment rules as aggregates and unions, with the following additions:

- Bit-field 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 part of storage unit.
- Unnamed bit-fields' types do not affect alignment of a structure or union

```
struct {
  short   S:9;
  int     J:9;
  char    C;
```

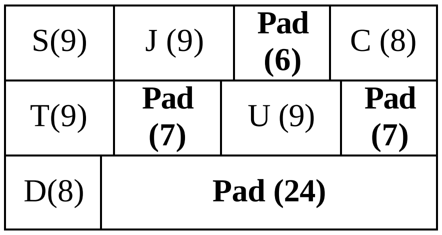


Figure 19-4. Storage unit sharing and alignment padding, sizeof is 12

19.2 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.

19.2.1 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	Yes	Callee-saved register
R30	No	Temporary register
R29	Yes	Callee-saved register
R28	No	Temporary register
R27	Yes	Callee-saved register
R26	No	Temporary register
R25	Yes	Callee-saved register
R24	No	Temporary register
R23	Yes	Callee-saved register
R22	No	Temporary register
R21	Yes	Callee-saved register
R20	No	Temporary register
R19	Yes	Callee-saved register
R18	No	Temporary register
R17	Yes	Callee-saved register
R16	No	Temporary register
R15	Yes	Callee-saved register

R14	No	Temporary register
R13	Yes	Callee-saved register
R12	No	Temporary register
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 19-4. General-Purpose Registers

General-purpose registers are divided into two groups. Registers from R0 to R15 are always present. Registers from R16 to R31 are present only in high-performance implementations.

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. 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 value is not defined. For functions returning union or structure, pointer to the result is placed into return value register.

In addition implementations with hard floating-point support or vector support also have 32 vector/floating-point registers. Each vector/floating-point register is 64 bits wide.

Register	Preserved across	Usage

	function calls	
VFR7- VFR31	No	Temporary
VFR6	No	Return value
VFR5	No	Function parameter number 5
VFR4	No	Function parameter number 4
VFR3	No	Function parameter number 3
VFR2	No	Function parameter number 2
VFR1	No	Function parameter number 1
VFR0	No	Function parameter number 0

Table 19-5. Vector/Floating-Point Registers

Some registers have assigned roles:

VFR0 through VFR5	**Vector/Floating-point parameters** use up to 6 vector/floating-point registers. Parameters beyond the sixth parameter appear on the stack.
VFR6 [RV]	**Return value** of the functions that return vector/floating-point type of data.

Furthermore, OpenRISC 1000 implementation might have several sets of shadowed general-purpose and vector/floating-point registers. These shadowed registers are used for fast context switching and sets can be switched only by the operating system.

19.2.2 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 19-6 shows the stack frame organization.

Position	Contents	Frame
FP + 8N … FP + 0	Parameter N … Parameter 0	Previous
FP – 8	Return address	Current
FP – 16	Previous FP value	
FP – 24 SP + 0	Function variables	
SP – 8 SP – 2092	For use by leaf functions w/o function prologue/epilogue	Future
SP – 2100 SP – 2536	For use by exception handlers	

Table 19-6. Stack Frame

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

19.2.3 Parameter Passing

Functions receive their first 6 arguments in general-purpose parameter registers or in vector/floating-point parameter registers. If arguments are combination of integer and vector/floating-point arguments, they are passed in first lowest general-purpose and vector/floating-point parameter registers. If there is more than six arguments, remaining arguments are passed on the stack.

Structure and union arguments are passed as pointers.

19.2.4 Functions Returning Scalars or No Value

A function that returns an integral or pointer value places its result in general-purpose RV register. A vector/floating-point return value appears in the vector/floating-point RV register. *Void* functions put no particular value in any RV register.

19.2.5 Functions Returning Structures or Unions

A function that returns structure or union, places address of the structure or union in the general-purpose RV register.

19.3 Operating System Interface

Exception Interface
Virtual Address Space
Page Size
Virtual Address Assignments

19.4 Position-Independent Code

19.5 ELF

The OpenRISC tools use 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.

19.5.1 Header Convention

The *e_machine* member of the ELF header contains the decimal value FOO (hexadecimal BAR) that is defined as the name EM_OPENRISC.

OpenRISC *e_ident* Fields		
e_ident[EI_CLASS]	ELFCLASS32	For all 32-bit implementations
e_ident[EI_DATA]	ELFDATA2MSB	For all implementations

Table 19-7. *e_ident* Field Values

The ELF header *e_flags* member contains zero, because the OpenRISC processor family defines no flags at this time.

19.5.2 Sections

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

19.5.3 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_OPENRISC_NONE	0	0	None
R_OPENRISC_INSN_REL_26	1	26	(S + A -P) >> 2
R_OPENRISC_INSN_ABS_26	2	26	(S + A) >> 2
R_OPENRISC_LO_16_IN_INSN	3	16	A & 0xffff
R_OPENRISC_HI_16_IN_INSN	4	16	(A >> 16) & 0xffff
R_OPENRISC_8	5	8	A & 0xff
R_OPENRISC_16	6	16	A & 0xffff

R_OPENRISC_32	7	32	A

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).

Values greater than R_OPENRISC_32 are compiler/assembler specific.

19.6 COFF

19.6.1 Sections

19.6.2 Relocation