LOONGSON

Dragon Architecture Reference Manual

Volume One: Infrastructure

V1.11

March 2025

Loongson Technology Co., Ltd.

Copyright Notice

Disclaimer

This document provides information at this stage only, and its content is subject to change based on the actual product situation without notice. Improper use of this document may result in consequences.

Our company assumes no responsibility for any direct or indirect losses caused.

Loongson Technology Co., Ltd.

Loongson Technology Corporation Limited

Address: Building 2, Longxin Industrial Park, Zhongguancun Environmental Protection Science and Technology Demonstration Park, Haidian District, Beijing

Building No.2, Loongson Industrial Park,

Zhongguancun Environmental Protection Park, Haidian District, Beijing

Telephone: 010-62546668

Fax: 010-62600826

Reading Guide

This manual is the first volume of the Dragon Architecture Reference Manual, which introduces the basic architecture of the Dragon Architecture.

Version History

Document Update History: Document Name, Version Number		Dragon Architecture Reference Manual Volume 1 Infrastructure
		1.11
	Creator	Chip R&D Department
	Creation Date	2020/04/27

Update History

Version number	Update content
0.80	Internal review version.
0.90	Internal review version.
0.91	Internal review version.
1.00	Official release version.
1.01	Manual content revisions: 1) The floating-point to integer conversion operation in section 3.2.3 does not check whether an exception to the floating-point inaccuracy report is allowed; that is, it is always executed. The `convertToIntegerExact...` or `roundToIntegralExact` operations. 2) Sections 4.2.1 and 7.2.3 further clarify the behavior of CSR directives accessing undefined or unimplemented CSRs. 3) In section 5.1, the maximum physical address range under the LA32 architecture is adjusted to 36 bits. 4) Corrected the inconsistency in the names of some CSR registers and their fields, and corrected several writing errors. The manual's content has been improved: 1) In sections 2.2.1.3, 2.2.4, 2.2.5.3, 2.2.7.2, and 3.2.5, the immediate values used in the instruction assembly representation are neutral with the instruction code. The relationship between numbers was explained.
1.02	Instruction content adjustment: 1) It is clear that the six instructions FSCALEB.S/D, FLOGB.S/D, and FRINT.S/D only need to be implemented under the LA64 architecture. Improvements and revisions to the manual: 2) Section 2.2.7.2 provides details on the specific operation method for calculating the offset value when the LL/SC instruction calculates the address. 3) Sections 2.1.4, 2.1.5 and 5.2.1 provide supplementary explanations on the rules for determining the exception triggered by the application's memory access. 4) In the last sentence of section 5.4.2 , [log2PS-1:12] should be [PS-1:12]. 5) In section 5.4.4, the page privilege level non-compliance exception should be SignalException(PPI). 6) In section 7.5.3, the description of the PPN field in the CSR registers TLBELO0 and TLBELO1 under the LA32 architecture should consider PALEN < 36. The situation. 7) The ASID field description in Section 7.5.4 should not be used as an INVTLB instruction to query the ASID key value information of the TLB. 8) Section 7.5.15 Definition of CSR registers TLBRELO0 and TLBRELO1 in the PPN field of the LA32 architecture and CSR Registers TLBELO0 and TLBELO1 are kept consistent.
1.03	Improvements and revisions to the manual: 1) Section 2.1.5 has been adjusted to no longer restrict the range of memory address space that application software can access at the instruction set level. The control is specifically implemented by the system software.

	2) The pseudocode description in section 2.2.3.3 was incorrect when the value of sa2 or sa3 was 0, and the expression was adjusted. 3) Adjustment of the meaning of bit 25 of CPUCFG word 1 in section 2.2.10.5. 4) In section 3.1.4.4, 2Emin should appear in the exponent position. 5) In the table of section 3.2.2.1, the IEEE 754-2008 function corresponding to the mnemonic CUNE should be compareQuietNotEqual. Instead of compareSignalingNotEqual. 6) The original wording of MOVCF2FR in section 3.2.4.6 and MOVCF2GR in section 3.2.4.7 is easily misunderstood as referring to the target. The register remains unchanged except for bit 0, while the exact behavior should be the rest of the register except for bit 0. Set the value to 0. 7) The instruction function descriptions in sections 4.2.4.3 and 4.2.4.4 , Supplementing TLB refill exception handling process regardless of Whether the CSR.TLBIDX.NE bit is 1 or not, a description of a valid TLB item is filled into the TLB. 8) The last two paragraphs in section 4.2.5.1 have been revised to clarify the input information related to the Huge Page. Judgment and result generation operations. 9) In section 4.2.5.2, the instruction format should be seq instead of req; at the same time, the original wording in the last paragraph has been adjusted to clarify... Input information determination and result generation operations related to Huge Pages. 10) In Section 5.2, paragraph 4, the content following "Rules for determining the legality of virtual address spaces:" is all... It should be changed to VALEN. 11) The last paragraph of section 5.2.1 is deleted because it does not restrict the range of memory address space that application software can access at the instruction set level. 12) In section 5.4.3.4, "INVTLB r0, r0" should be "INVTLB 0, r0, r0". 13) The range truncated in the second-to-last line of the pseudocode description in section 5.4.4 is incorrect. 14) In section 5.4.5, the subscript of the PA field in the large page table entry format should be log2PageSize; the original 24 only applies to 16MB large pages. Smaller pages, larger pages. 15) The order of the descriptive statements related to the ordinary exception entry offset in section 6.2.1 was adjusted to resolve the ambiguity in the original statement when CFG.VS=0. The description was corrected; the typo "its value equals 2 (CSR.ECFG.VS+2)×(ecode+64)" was also corrected to "its value equals...". "2 (CSR.ECFG.VS+2)×ecode". 16) Regarding the priority of exceptions triggered during the execution phase in Section 6.2.2, the Address Error Exception (ADE) should have a higher priority than the required address error exception. Address alignment error exception (ALE) occurs when an address-aligned memory access instruction causes an address misalignment. 17) In the descriptions of the DATF and DATM fields in Table 7-2 of Section 7.4.1, "..., it is necessary to change ..." to "..., push..." At the same time, it is recommended that... 18) In the description of the VS field in Table 7-6 of Section 7.4.5, "...error exceptions have their own independent entry base address," should be changed to "...error..." The exception has an independent entry base address. 19) Sections 7.5.8 and 7.5.9, in the descriptions of Tables 7-29 and 7-30, have added explanations of which level of page table, in relation to LDDIR and LDPTE. The function description in the instruction corresponds to the function description. 20) In Section 7.5.8, the bit widths represented by the PTEWIDTH field values 2 and 3 in Table 7-29 are corrected to 256 bits and 512 bits, respectively.
1.10	Improvements and revisions to the manual: 1) Update the Chinese expression for TiesToEven to "round down to the nearest (intermediate)". 2) Correct FTINT.WD , FTINT.LS , FTINTRM.W.D , FTINTRM.L.S , FTINTRP.WD , The FR[fd] update range is incorrect in the descriptions of the FTINTRP.LS, FTINTNE.WD, and FTINTNE.LS instructions. 3) The CPUCFG instruction configuration information word number is prefixed with "0x" to eliminate ambiguity. Added content related to LoongArchV1.1.
1.11	Improvements and revisions to the manual:

	1) Add a description to section 2.2.1.3 explaining why the immediate value of this instruction is only 2 bits. 2) Corrected redundant content in the natural language description of the LU52I.D instruction in section 2.2.1.4. 3) Corrected the description of non-compliant values for the DIV.W[U] and MOD.W[U] instructions in section 2.2.1.13. 4) Adjust the pseudocode description of the BSTRINS.{W/D} instruction in section 2.2.3.8 to eliminate the original description's lsbw=0, Ambiguity in these boundary cases: msbw=31, lsbd=0, msbd=63. 5) Corrected the typo of writing amcas as awcas in section 2.2.7.3. 6) Revised the description of basic floating-point instructions at the beginning of Chapter 3, which was only implemented under the LA64 architecture, and added "operating on single-precision floating-point numbers and..." List of basic floating-point instructions for word integers. 7) Corrected errors in the description of the LDDIR instruction in section 4.2.5.1 and adjusted the narrative style to increase readability. 8) Corrected three instances in sections 5.4.5 and 6.3.4 where VALEN was mistakenly written as PALEN. 9) The number of load/store monitoring points and instruction fetch monitoring points in Table 7-1 can be adjusted to a maximum of 14.

Table of contents

Catalog

Table of Contents

1 Introduction

LoongArch is a Reduced Instruction Set Computing (RISC) style architecture.

This document describes the system architecture. The Dragon Architecture Reference Manual is used to explain the Dragon Architecture specification and consists of three volumes. This document is Volume One, which describes the fundamentals of the Dragon Architecture.

The content of the section.

1.1 Overview of Dragon Architecture

The Dragon architecture exhibits typical characteristics of a RISC instruction set architecture. Its instructions have fixed lengths and regular encoding formats, with the vast majority of instructions having only two sources.

The system uses one operand and one destination operand, employing a load/store architecture. This means that only load/store memory access instructions can access memory; other instructions cannot access it.

The objects are all immediate values in the registers or instruction codes inside the processor core.

The Dragon architecture comes in two versions: 32-bit and 64-bit, referred to as the LA32 and LA64 architectures, respectively. The LA64 architecture uses application-level backward binary.

Compatible with LA32 architecture. "Application-level backward binary compatibility" means that the binary representation of application software using the LA32 architecture can be directly...

This means that the same results are obtained when running on a machine compatible with the LA64 architecture. On the other hand, this backward binary compatibility is limited to application software.

The architecture specification does not guarantee that the binaries of system software (such as the operating system kernel) running on LA32-compatible machines will be directly compatible with LA64.

The same results are always obtained when running on machines with the same architecture.

The Dragon architecture uses a base component (Loongson Base) plus an extension component (as shown in Figure 1-1). The extension component includes:

Loongson Binary Translation (LBT) and Loongson Virtualization (LVZ) extensions

Loongson SIMD Extension (LSX) and Loongson Advanced SIMD Extension (LSX)

(Referred to as LASX).

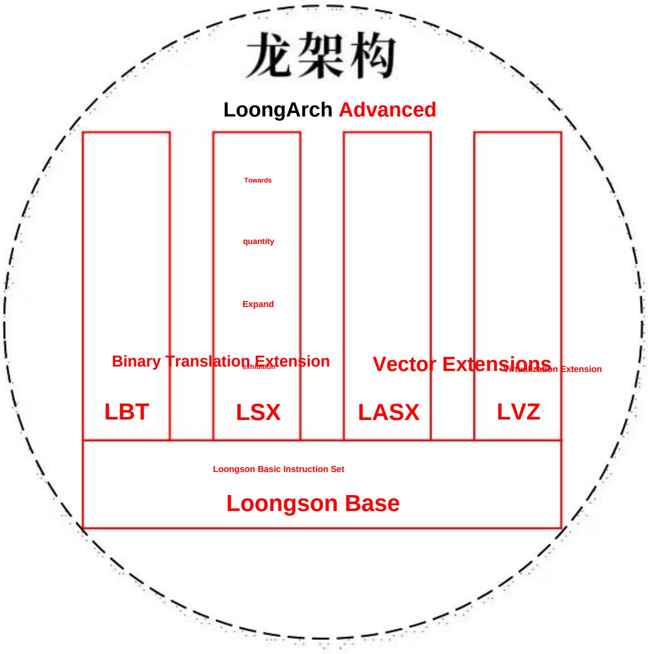

Figure 1-1 Components of the Dragon Architecture

The core of the Dragon architecture consists of two parts: a non-privileged instruction set and a privileged instruction set. The non-privileged instruction set defines commonly used integer instructions.

It supports both integer and floating-point instructions, enabling it to generate efficient target code for all current mainstream compilation systems.

The virtualization extension of the Dragon architecture provides hardware acceleration for operating system virtualization to improve performance. This part mainly involves...

Privileged resources include privileged instructions and control status registers, as well as new features added in areas such as exceptions and interrupts, and memory management.

The binary translation extension for the Dragon architecture is designed to improve the execution efficiency of cross-instruction set binary translation on the Dragon architecture platform. Its foundation...

It extends beyond the existing parts, also including both non-privileged instruction sets and privileged instruction sets.

Both the Loongson Vector Instruction Extension and the Advanced Vector Instruction Extension utilize SIMD instructions to accelerate computationally intensive applications. (The two extensions...)

Some are basically the same in terms of instruction functionality, the difference being that the vector bit width of vector instruction extension operations is 128 bits, while that of high-level vector instruction extension operations is...

The vector width is 256 bits.

An implementation of a dragon-compatible architecture must include at least the basic components of the architecture; the extended components can be implemented selectively. Each extended component can...

The choice is flexible; however, when choosing to implement LASX, LSX must also be implemented. Besides making an entire extension section optional, in the base section...

The sub-sections and their extensions further include optional subsets of functionality. All these optional extensions, or sub-sections, contain...

The specific implementation details of the optional feature subset can be dynamically identified by the software using configuration information words read by the CPUCFG instruction.

This manual will begin a detailed description of the Dragon architecture specification from Chapter 2 onwards. Chapters 2 and 3 specifically cover the infrastructure aspects...

The non-privileged instruction set section includes the functional definitions of basic integer instructions and basic floating-point instructions, as well as their application-level programming models. Chapters 4 through 7

This chapter discusses privileged resources in the infrastructure, primarily including privileged instructions and the Control and Status Register.

This document introduces the Command and Execution System (CSR) and its functional specifications regarding operating modes, exceptions and interrupts, and memory management. The main body of this document describes the instructions.

The pseudocode descriptions involved in the function definition are concentrated in Appendix A, and the specific encoding definitions of the instructions involved are uniformly listed in Appendix B.

1.2 Instruction Encoding Format

All instructions in the Dragon architecture are 32-bit fixed-length, and instruction addresses must be aligned to 4-byte boundaries. When instruction addresses are not aligned...

An address error exception will be triggered at that time.

The instruction encoding style is such that register operand fields are typically arranged sequentially from bit 0 to bit 1. Opcodes always start from bit 31.

They are arranged from highest to lowest order. If the instruction contains immediate operands, the immediate field is located between the register field and the opcode field.

The length varies depending on the instruction type. Specifically, it includes 9 typical instruction encoding formats, namely 3 encoding formats without immediate values.

2R, 3R, 4R, and six encoding formats containing immediate values: 2RI8, 2RI12, 2RI14, 2RI16, 1RI21, and I26. Table 1-1 lists these nine.

The specific definitions of the nine typical encoding formats are provided. It should be noted that a few instructions have an instruction encoding field that is not entirely equivalent to these nine typical formats.

The encoding format is not modified, but slightly altered. However, the number of such instructions is small, and the changes are minor, so they will not significantly affect the encoding.

This caused inconvenience to the developers of the translation system.

Table 1-1 Typical instruction encoding format of the Dragon architecture

	3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
2R-type	opcode																						rj					rd				
3R-type	opcode																	rk					rj					rd				
4R-type	opcode												day					rk					rj					rd				
2RI8-type	opcode														I8								rj					rd				
2RI12-type	opcode										I12												rj					rd				
2RI14-type	opcode								I14														rj					rd				
2RI16-type	opcode						I16																rj					rd				
1RI21-type	opcode						I21[15:0]																rj					I21[20:16]				
I26-type	opcode						I26[15:0]																I26[25:16]									

1.3 Mnemonic Format for Instruction Assembly

The instruction assembly mnemonic format mainly consists of two parts: the instruction name and the operands. The Dragon architecture standardizes the prefixes and suffixes for both instruction names and operands. This is intended to facilitate use by assembly programmers and compiler developers.

First, non-vector instructions and vector instructions, as well as integer and floating-point instructions, are distinguished by the prefix letters of their instruction names. All 128-bit vector instructions... Instruction names begin with the letter "V"; all 256-bit vector instructions begin with the letter "XV". All non-vector floating-point instructions... Instruction names begin with the letter "F"; all 128-bit vector floating-point instructions begin with "VF"; all 256-bit vector floating-point instructions... Instruction names begin with "XVF".

Secondly, the vast majority of instructions use a suffix in the form of ".XX" in the instruction name to indicate the target of the instruction, and this type of suffix is only used in specific instructions. This is used to characterize the type of the operand. For operands of integer type, the instruction name suffix is .B, .H, .W, .D, .BU, .HU, .WU, or .DU. These represent the data types operated on by the instruction: signed byte, signed half-word, signed word, signed double-word, unsigned byte, and unsigned double-word, respectively. Half-sign, unsigned, and unsigned double-sign. However, there is a special case where whether the operands are signed or unsigned does not affect the operation. When calculating the result, the instruction name does not include the suffix "U," but this does not restrict the operands to only signed numbers. For operands that are floats... Point-based instructions, or more specifically, those whose names begin with "F", "VF", and "XVF", have a suffix indicating their value. The .H, .S, .D, .W, .L, .WU, and .LU symbols respectively indicate that the data type operated on by this instruction is half-precision floating-point, single-precision floating-point, and double-precision floating-point, respectively. Floating-point numbers, signed words, signed double words, unsigned words, and unsigned double words. Additionally, in instructions involving vector operations, the instruction name suffix is .V. This indicates that the instruction operates on the entire vector data as a whole. It should be noted that not all instructions use the ".XX" form.

The suffix indicates the operand of the instruction. The data width of the operand is determined by whether the processor is 32-bit or 64-bit.

Instructions such as SLT and SLTU are not appended to a suffix. Additionally, privileged instructions that operate on CSR, TLB, and Cache, as well as those in different...

Instructions for moving data between register files also do not include this suffix indicating the type of the operand.

When the data width and sign of the source and destination operands are the same, the instruction name has only one suffix. If all source operations

If the data width and whether it is signed or unsigned are the same, but different from the destination operand, then the instruction name will have two suffixes, from left to right.

The first suffix indicates the destination operand, and the second suffix indicates the source operand. If the source and destination operand details are more...

If the instruction is complex, then the instruction name will list the destination operand and each source operand from left to right, in the same order as the later operands in the instruction mnemonic.

The order of operands must be consistent. For example, in the instruction "MULW.D.WU rd, rj, rk", .D corresponds to the destination operand rd, and .WU corresponds to the source operands rj and rk.

`rk` indicates that this multiplication involves multiplying two unsigned words, and the resulting double word is written to `rd`. For example, the instruction "CRC.WBW rd, rj, rk"

The first .W corresponds to rd, .B corresponds to rj, and the second .W corresponds to rk, indicating that this CRC check operation compares the byte message in rj with the byte message in rk.

The original 32-bit checksum is used to generate a new 32-bit checksum, which is then written into rd.

Register operands are identified by their initial letter, indicating which register file they belong to. General-purpose registers are labeled "rN", and those are labeled "fN".

Floating-point registers are labeled with "vN", 128-bit vector registers are labeled with "vN", and 256-bit vector registers are labeled with "xN". Where N is a number...

The word indicates that the operation is performed on register number N in the register file.

1.4 Some writing rules adopted in this manual

1.4.1 Command Name Abbreviation Rules

Among the instructions defined in the Dragon architecture, there are often some instructions that have the same or similar operation patterns, differing only in the objects they operate on.

Different. In the introduction of commands and functions in this manual, such commands are often grouped together in one place for easy learning and reference by users.

To maintain brevity, this manual employs a rule for abbreviating instruction names. In this rule, {A/B/C} indicates that A, B, and C are used respectively to refer to the instruction name.

Instead of using A[B] to form different instruction names, A[B] indicates that A and AB are used to form different instruction names. For example, ADD.{W/D} means...

The first one represents the instruction names ADD.W and ADD.D, while BLT[U] represents the instruction names BLT and BLTU. A more complex one...

ADD[I].{W/D} represents the four instruction names: ADD.W, ADD.D, ADDI.W, and ADDI.D.

It is important to note that this abbreviation rule is merely a writing rule; it does not mean that several instructions abbreviated together must also be...

They have very similar instruction codes.

1.4.2 Control Status Register Designation Method

The Dragon architecture defines a series of Control and Status Registers (CSRs) for controlling instruction execution.

Each CSR typically contains several fields. Throughout this manual, the abbreviation will be referred to using the form CSR.%%%%.####.

Write the field named #### in the control status register %%%% of the specified value. For example, CSR.CRMD.PLV represents the PLV field in the CRMD register.

domain.

In the case of virtualization extension, there will be two CSRs in the processor, one for the host and one for the guest.

When the context alone cannot distinguish between the two CSRs, CSR.XXXX represents the host's CSR, and GCSR.XXXX represents the host's CSR.

This refers to the client's CSR.

Version Evolution of Dragon Architecture 1.5

The initial version of the Dragon Architecture was V1, denoted as LoongArch V1. Unless otherwise specified in the *Dragon Architecture Reference Manual*, the standard's content is assumed to be standard.

This belongs to LoongArch V1. Since LoongArch V1, subsequent evolutions of the Dragon architecture have adopted a fine-grained incremental evolution approach. Here, "fine-grained" refers to...

Evolution refers to the fact that each functional subset within the basic or extended parts can evolve independently; "incremental" means that for any

Each part can evolve independently, and higher versions are always backward binary compatible with lower versions. To more concisely illustrate the stages of the above architectural evolution process,

A new version extension will refer to a collection of new features added during a specific phase. For example, the new features added compared to LoongArch V1...

Hardware page table traversal support, byte/half-word atomic memory access instructions, and other features are collectively referred to as LoongArch V1.1. It should be noted that each new version...

The newly added feature subset has independent flags in the CPUCFG instruction return value. It is recommended that software follow this information rather than the Dragon architecture version.

The architecture specification does not require the processor hardware implementation to directly reflect the supported architecture version number to determine the functionality supported by the processor.

able.

New features in **LoongArch V1.1 version 1.5.1**

LoongArch V1.1 adds the following features:

1. Added instructions for approximate calculation of floating-point square roots and reciprocals of floating-point square roots, including FRECIPE.S and FRECIPE.D instructions for scalar operations.

   The FRSQRTE.S and FRSQRTE.D instructions, and the VFRECIPE.S, VFRECIPE.D, and VFRSQRTE.S instructions for 128-bit SIMD operations.

   The VFRSQRTE.D instruction and XVFRECIPE.S, XVFRECIPE.D, XVFRSQRTE.S for 256-bit SIMD operations.

   The XVFRSQRTE.D instruction.

2. Added the SC.Q command.

3. Added the commands LLACQ.W, SCREL.W, LLACQ.D, and SCREL.D.

4. ÿÿ AMCAS.BÿAMCAS.HÿAMCAS.WÿAMCAS.DÿAMCAS_DB.BÿAMCAS_DB.HÿAMCAS_DB.Wÿ

   The AMCAS_DB.D, AMSWAP.B, and AMSWAP.H commands.

5. ÿÿ AMADD.BÿAMADD.HÿAMSWAP_DB.BÿAMSWAP_DB.HÿAMADD_DB.BÿAMADD_DB.H

   instruction.

6. Added the functionality definition for non-zero hint values in the dbar command section.

7. Added a method to determine whether the execution of a 32-bit integer division instruction on a 64-bit machine is affected by the value of the high 32 bits of the source operand register.

8. Standardize the method for determining the order of execution of load memory access operations at the same address.

9. Add a definition for message interruption.

10. Allows hardware page table traversal.

2 Basic Integer Instructions

The non-privileged instruction set of the Dragon architecture's foundation can be divided into basic integer instructions and basic floating-point instructions based on differences in the software runtime context.

The instruction set consists of two parts: integer instructions and numbers. This chapter will describe the integer instruction part. The basic integer instruction part is the most fundamental part of the non-privileged instruction subset.

point.

2.1 Basic Integer Instruction Programming Model

The basic integer instruction programming model described in this section only covers the aspects that application software developers need to focus on. This content primarily belongs to...

The non-privileged parts of the architecture, however, are always related to some privileged resources in the runtime environment of application software, so they are used where necessary.

The concept of privileged resources will be introduced to ensure the completeness of the narrative. While the topic of privileged resources is touched upon here, it will not be elaborated upon.

Readers who require a more comprehensive and in-depth understanding can refer to the relevant chapters in the manual based on the prompts in the text.

2.1.1 Data Types

There are five data types that basic integer instructions operate on: bit (b), byte (B, 8 bits), halfword (H, 16 bits), word (W, 32 bits), and doubleword (D, 32 bits).

(64b). In the LA32 architecture, there are no integer instructions for operating double words.

Byte, half-word, word, and double-word data types all use the two's complement encoding method.

2.1.2 Registers

The registers involved in basic integer instructions include the general-purpose register (GR) and the program counter.

Couner (abbreviated as PC), as shown in Figure 2-1.

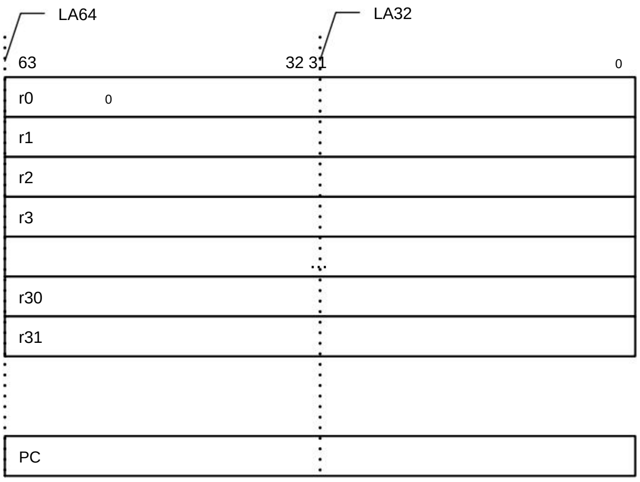

Figure 2-1 General-purpose registers and **PC**

[1] Application software refers to software that cannot directly manipulate privileged resources within the architecture. In the Linux operating system, it refers to software that runs in user mode.

2.1.2.1 General-purpose registers

There are 32 general-purpose registers GR, denoted as r0~r31, with register 0 (r0) always having a value of 0. The bit width of GR is denoted as GRLEN. LA32

In the LA64 architecture, the GR instruction has a 32-bit width, while in the LA64 architecture, the GR instruction has a 64-bit width. There is an orthogonality between basic integer instructions and general-purpose registers.

From an architectural perspective, any register operand in these instructions can use any of the 32 registers (GRs). The only exception is...

The destination register implicitly included in the BL instruction is always register r1, which is the first register in the standard Dragon architecture application binary interface.

In the Binary Interface (ABI), r1 is fixed as a register for storing the return address of a function call.

2.1.2.2PC

There is only one PC (Program Counter), which records the address of the current instruction. The PC register cannot be directly modified by instructions; it can only be modified by jump instructions and exception traps.

The PC register is indirectly modified by inbound and exception return instructions. However, it can be directly read as a source operand for some non-jump instructions.

The width of is always the same as the width of GR.

2.1.3 Execution Privilege Level

The Dragon architecture defines four privilege levels (PLVs): PLV0 to PLV3. Application software should run...

At the three non-privileged levels PLV1 to PLV3, application software is isolated from system software such as the operating system running at PLV0.

The specific privilege level at which an application runs is determined by the system software at runtime; the application software has no precise knowledge of this. Under the Dragon architecture, applications...

The software typically runs at PLV3 level. For more information on privilege levels, see Section 4.1 .

2.1.3.1 Privileged Resources Accessible by Application Software

Generally, privileged resources cannot be directly accessed by applications running at a non-privileged level, but when CSR.MISC ...

When RPCNTL1/RPCNTL2/RPCNTL3 is configured to 1, CSRRD instruction reads can be executed under the PLV1/PLV2/PLV3 privilege levels.

It can monitor counters. For more information on performance monitoring counters, please see Section 7.8 .

2.1.3.2 Disabling some non-privileged functions

Some non-privileged functions that are enabled by default after a power-on reset can be disabled by the system software during operation. This can be done by configuring CSR.MISC .

Setting DRDTL1/DRDTL2/DRDTL3 to 1 disables the execution of RDTIME type instructions at PLV1/PLV2/PLV3 levels. Violation will trigger a penalty.

Instruction privilege level error exception (IPE).

2.1.4 Exceptions and Interruptions

Exceptions and interrupts interrupt the currently executing application, switching the program execution flow to the exception/interrupt.

Execution begins at the entry point of the handler. Exceptions are triggered by unusual conditions that occur during instruction execution, while interrupts are caused by external events (such as...).

An interrupt input signal triggers this. In this architecture reference manual, we will strictly distinguish between "generating an exception/interrupt" and "triggering an exception/interrupt".

The difference between the two concepts is that the former may not necessarily cause a change in the execution flow, while the latter will definitely change the current execution flow and transfer it to the exception/interrupt handler.

At the mouth.

The handling of exceptions and interruptions falls under the scope of privileged resource management in the architecture. This section primarily focuses on exceptions that are perceptible to application software.

Here is a brief introduction.

ÿ System call exception: Executing the SYSCALL instruction will immediately trigger a system call exception (SYS).

ÿ Breakpoint Exception: Executing the BREAK instruction will immediately trigger a breakpoint exception (BRK).

ÿ No exceptions to the instruction: The instruction code being executed is not defined in the architecture, or the architecture specification defines the instruction in the current context.

If it is treated as not existing, then the instruction not existing exception (INE) will be triggered immediately.

ÿ Privileged Instruction Exceptions: Except for the special cases listed in Section 2.1.3 , executing a privileged instruction in an application will definitely result in an error.

The privilege level exception (IPE) of the trigger instruction is set.

ÿ Address Incorrect Exception: When a program malfunction causes an invalid address for an instruction fetch or memory access instruction, such as an address that is not 4.

Errors such as byte boundary alignment errors or accessing illegal address spaces will trigger an Address Fetch Error Exception (ADEF) or a memory access instruction address error.

Error exception (ADEM).

ÿ Floating-point error exception: When an abnormal data condition occurs during the execution of a floating-point instruction, special handling is required, which may generate or trigger a basic error.

Floating-point error exception (FPE). See section 3.1.4 for more information.

2.1.5 Memory Address Space

This section only covers the virtual memory address space visible to the application software. The translation from virtual memory addresses to physical memory addresses is determined by the runtime environment.

These contents involve the relevant specifications of privileged resources in the architecture, which will be introduced in the latter half of this manual.

In the Dragon architecture, the memory address space is a linear, contiguous address space that is byte-addressable.

Under the LA32 architecture, the recommended memory address space access range for application software is 0 to 2^ 31 - 1.

Under the LA64 architecture, the recommended memory address space access range for application software is 0 to 2VALEN-1 -1. Here, VALEN is theoretically a...

Integers less than or equal to 64, their specific values are determined by the hardware implementation. Common VALEN values are in the range [40, 48]. Application software can execute...

The CPUCFG instruction reads the VALEN field of configuration word 0x1 to determine the specific value of VALEN.

2.1.6 Tail end

The Dragon architecture uses only a small-tailed storage method.

2.1.7 Storage Access Types

The Dragon architecture supports three storage access types: Coherent Cached (CC) and Strong-Order Non-Cache.

Strongly-ordered Uncached (SUC) and Weakly-ordered Uncached (WUC) are two types of memory access types. The memory access type is bound to the accessed virtual address, determined by the MAT (Memory Access Type) field in the page table entry. The value range of the MAT field determines the memory access type.

[1] In the Loongson architecture, interrupts are always invisible to application

[2] software. This only applies to the scope of application software. For system software, in direct address translation mode or mapped address translation mode, the address falls within the address range configured in the direct mapping window. Within this range, the storage access type is configured by the specified control status register.

The type correspondence is as follows: 0 – Strongly ordered, non-cached; 1 – Consistent, cacheable; 2 – Weakly ordered, non-cached; 3 – Reserved. Storage access classes.

The configuration process is transparent to the application software.

When accessing objects using a consistent cacheable access type, the accessed object can be either the final stored object or a cached object maintained in the processor.

Consistent caching. This type of memory access is typically used to achieve high performance.

When accessing data using either strong-order uncached or weak-order uncached types, only the final stored object can be accessed directly. The difference between the two is: strong-order uncached...

Memory access must satisfy sequential consistency, meaning that all accesses are executed strictly in the order specified in the program, and no new access can begin until the current memory access operation is completely completed.

The next memory access operation; while weakly ordered uncached read access allows speculative execution, and weakly ordered uncached write data can be merged into the processor core.

Larger writes (such as a cache line) are then written in a burst, where subsequent writes can overwrite earlier ones during the merging process.

according to.

The Dragon architecture only requires that strongly ordered, non-cached memory access instructions cannot have side effects, meaning that such instructions cannot be executed predictably.

Software can leverage this feature to access I/O devices in the system using strongly ordered, non-cached memory access instructions. However, the Dragon architecture allows strongly ordered, non-cached...

Fetch operations on cached types have side effects. This refers to fetch operations that access strongly ordered, non-cached types, even if they originate from transition prediction.

The result also allows execution. To prevent such speculative execution from causing out-of-core memory accesses to mistakenly enter illegal physical address spaces, on-chip...

Filter out risky access from the network.

Weakly ordered, non-cached access types are typically used to accelerate access to non-cached memory data, such as video memory data.

2.1.7.1 **Cache** Coherence Maintenance of Instruction **Cache**

The instruction cache of a certain processor core is consistent with the cache or cache coherent I/O of other processor cores.

Cache consistency between Masters must be maintained by hardware.

Maintaining cache coherence between the processor core's instruction cache and data cache can be implemented as hardware maintenance. This means that for self-repair...

By modifying the code, the software no longer needs to use cache maintenance instructions to ensure cache consistency between the instruction cache and the data cache within the same core.

However, due to the pipelined architecture and speculative instruction fetching behavior, the software still needs to use the IBAR instruction to ensure that the instruction fetch can definitely see the store.

The effect of executing the command.

2.1.8 Unaligned memory access

All memory access addresses for instruction fetch operations must be aligned to 4-byte boundaries; otherwise, an Address Fetch Error Exception (ADEF) will be triggered.

Apart from atomic memory access instructions, integer boundary-checking memory access instructions, and floating-point boundary-checking memory access instructions, the remaining load/store memory access instructions...

It can be implemented to allow misaligned memory access addresses. However, in an implementation that allows misaligned memory access addresses, the system-level software can configure...

The ALCL0~ALCL3 control bits in CSR.MISC , under privilege levels PLV0~PLV3, also address these load/store memory access instructions.

Alignment check. For memory access instructions that require address alignment checks, if the address being accessed is not naturally aligned to 1, an address misalignment check will be triggered.

ALE (Alternative for All).

Natural alignment refers to the following: when accessing a half-word object, the address is aligned to a 2-byte boundary; when accessing a word object, the address is aligned to a 4-byte boundary; when accessing a double-word object, the address is aligned to an 8-byte boundary; when accessing a 128-bit vector object, the address is aligned to a 16-byte boundary; and when accessing a 256-bit vector object, the address is aligned to a 32-byte boundary.

2.1.9 Brief Description of Storage Consistency Model

The Dragon architecture employs a weak consistency (WC) model for storage consistency. This section only discusses the architecture's implementation.

Here is a brief description of the weak consistency model.

In a weak consistency model, synchronization operations and regular memory accesses need to be distinguished. Programmers must use the synchronization operations defined by the architecture to handle these operations.

Access to the shared memory unit is protected to ensure that access to the shared memory unit by multiple processor cores is mutually exclusive. The order of memory access events is also considered.

The following restrictions shall be imposed:

1. The execution of synchronization operations satisfies the sequential consistency condition. That is, synchronization operations are executed strictly in accordance with their order of appearance in the program across all processor cores.

   The synchronization operations are executed in the order they are performed, and the next synchronization operation cannot begin until the current synchronization operation is completely completed.

2. Before any normal memory access operation is allowed to be executed, all synchronization operations that precede this memory access operation in the same processor core have already been performed.

   Completed;

3. Before any synchronization operation is allowed to be executed, all ordinary memory access operations that precede this synchronization operation in the same processor must have been completed.

   become.

In the Dragon architecture, the instructions capable of generating synchronous operations include DBAR, IBAR, AM atomic memory access instructions with DBAR functionality, and LL-SC.

Command pair.

2.1.9.1 Sequential execution of **load** memory access operations at the same address

When the Dragon architecture uses a weakly consistent storage consistency model, it does not require by default that the hardware supports sequential execution of load memory access operations at the same address.

To ensure correct program execution, software needs to add data barrier instructions (dbar 0x700 is recommended) where necessary. This depends on the specific processor implementation.

If memory access operations with the same address are executed sequentially, it should be ensured that the return value of CPUCFG.3.LD_SEQ_SA[bit23] is 1, so that the software can recognize the memory access sequence.

After identifying this characteristic, relevant performance optimizations were performed.

2.2 Overview of Basic Integer Instructions

This section describes the functionality of application-level basic integer instructions in the LA64 architecture. For the LA32 architecture, only the following needs to be implemented:

A subset of instructions is provided, and the list of instructions contained in this subset is shown in Table 2-1. Since the GR instruction has a bit width of only 32 bits in the LA32 architecture, the subsequent instruction descriptions...

The sign extension operation in "write the 32-bit result sign extended into the general-purpose register rd" is not required.

Table 2-1 Overview of **LA32** Application-Level Basic Integer Instructions

Arithmetic operation instructions	ADD.W, SUB.W, ADDI.W, ALSL.W, LU12I.W, SLT, SLTU, SLTI, SLTUI, PCADDI, PCADDU12I, PCALAU12I, AND, OR, NOR, XOR, ANDN, ORN, ANDI, ORI, XORI, MUL.W, MULH.W, MULH.WU, DIV.W, MOD.W, DIV.WU, MOD.WU
Shift operation instructions: SLL.W, SRL.W, SRA.W, ROTR.W, SLLI.W, SRLI.W, SRAI.W, ROTRI.W	
Bit manipulation instructions	EXT.W.B, EXT.W.H, CLO.W, CLZ.W, CTO.W, CTZ.W, BYTEPICK.W, REVB.2H, BITREV.4B, BITREV.W, BSTRINS.W, BSTRPICK.W, MASKEQZ, MASKNEZ
Transfer instructions and	BEQ, BNE, BLT, BGE, BLTU, BGEU, BEQZ, BNEZ, B, BL, JIRL
memory access instructions	LD.B, LD.H, LD.W, LD.BU, LD.HU, ST.B, ST.H, ST.W, PRELD

Atomic memory access instructions,	LL.W, SC.W
barrier instructions,	DBAR, IBAR
and other miscellaneous instructions	SYSCALL, BREAK, RDTIMEL.W, RDTIMEH.W, CPUCFG

Furthermore, for instructions whose operands have a data width of GR, the operand width is 32 bits in the LA32 architecture and 32 bits in the LA64 architecture.

The architecture has an operation width of 64 bits. Unless otherwise specified, the instruction function description will not be further elaborated.

2.2.1 Arithmetic Operation Instructions

2.2.1.1ADD.{W/D}, SUB.{W/D}

Command format: add.w	rd, rj, rk	add.d	rd, rj, rk
sub.in	rd, rj, rk	sub.d	rd, rj, rk

ADD.W adds the data in bits [31:0] of general-purpose register rj to the data in general-purpose register rk. The digits [31:0] of the result are delimited.

The extended number is then written into the general-purpose register rd.

ADD.W:

```
tmp = GR[rj][31:0] + GR[rk][31:0]
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

SUB.W subtracts the data in bits [31:0] of general-purpose register rj from the data in general-purpose register rk, and the sign bit [31:0] of the result is...

The extended number is then written into the general-purpose register rd.

SUB.W:

```
tmp = GR[rj][31:0] - GR[rk][31:0]
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

Add the data in bits [63:0] of general-purpose register rj to the data in bits [63:0] of general-purpose register rk, and write the result back to the general-purpose register.

In the device rd.

ADD.D:

```
tmp = GR[rj][63:0] + GR[rk][63:0]
GR[rd] = tmp[63:0]
```

SUB.D subtracts the data in bits [63:0] of general-purpose register rj from the data in bits [63:0] of general-purpose register rk, and writes the result back to the general-purpose register.

In the device rd.

SUB.D:

```
tmp = GR[rj][63:0] - GR[rk][63:0]
GR[rd] = tmp[63:0]
```

The above instructions do not perform any special handling for overflow situations.

2.2.1.2ADDI.{W/D}, ADDU16I.D

Command format: addi.w rd, rj, si12 addi.d rd, rj, si12

addu16i.d rd, rj, si16

ADDI.W adds the [31:0] bits of data in the general-purpose register rj to the 32-bit data after sign extension of the 12-bit immediate value si12. The result is...

The [31:0] bits are then written to the general-purpose register rd after sign extension.

ADDI.W:

```
tmp = GR[rj][31:0] + SignExtend(si12, 32)
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

ADDI.D adds the [63:0] bits of data in the general-purpose register rj to the 64-bit data after sign extension of the 12-bit immediate value si12. The result is...

Write it into the general-purpose register rd.

ADDI.D:

```
tmp = GR[rj][63:0] + SignExtend(si12, 64)
GR[rd] = tmp[63:0]
```

ADDU16I.D logically shifts the 16-bit immediate value si16 left by 16 bits and then sign-extends it. The resulting data is then added to [63:0] in the general-purpose register rj.

The bits of data are added together, and the result is written to the general-purpose register rd. The ADDU16I.D instruction is used in conjunction with the LDPTR.W/D and STPTR.W/D instructions to...

To accelerate access to the GOT table in location-independent code.

ADDU16I.D:

```
tmp = GR[rj][63:0] + SignExtend({si16, 16'b0}, 64)
GR[rd] = tmp[63:0]
```

The above instructions do not perform any special handling for overflow situations.

2.2.1.3ALSL.{W[U]/D}

Command format: alsl.w rd, rj, rk, sa2 etc.d rd, rj, rk, sa2

alsl.wu rd, rj, rk, sa2

ALSL.W logically shifts the data in bits [31:0] of the general-purpose register rj left by (sa2+1) bits and then adds the data in bits [31:0] of the general-purpose register rk.

The sign extension of the [31:0] bits of the result is written into the general-purpose register rd.

ALSL.W:

```
tmp = (GR[rj][31:0]<<(sa2+1)) + GR[rk][31:0]
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

ALSL.WU logically shifts the data in bits [31:0] of the general-purpose register rj left by (sa2+1) bits and then adds the data in bits [31:0] of the general-purpose register rk.

The [31:0] bits of the result are zero-extended and written into the general-purpose register rd.

ALSL.WU:

```
tmp = (GR[rj][31:0]<<(sa2+1)) + GR[rk][31:0]
GR[rd] = ZeroExtend(tmp[31:0], GRLEN)
```

ALSL.D logically shifts the data in bits [63:0] of the general-purpose register rj left by (sa2+1) bits and then adds the data in bits [63:0] of the general-purpose register rk. The result is written into the general-purpose register rd.

ALSL.D:

```
tmp = (GR[rj][63:0]<<(sa2+1)) + GR[rk][63:0]
GR[rd] = tmp[63:0]
```

The shift amount of rj in the above instructions only considers the cases of 1, 2, 3, and 4, so only a 2-bit immediate value sa2 is needed to represent it.

The above instructions do not perform any special handling for overflow situations.

It is important to note that the immediate value in the assembly representation of the above instructions should be filled in as (sa2+1), which is the actual shift value and not the immediate value in the instruction code. The value of the field.

2.2.1.4LU12I.W, LU32I.D, LU52I.D

Command format: lu12i.w rd, si20

lu32i.d rd, si20

lu52i.d rd, rj, si12

LU12I.W concatenates the least significant bit of the 20-bit immediate value si20 with 12 bits of 0, then signs-extends it and writes it into the general-purpose register rd.

LU12I.W:

```
GR[rd] = SignExtend({si20, 12'b0}, GRLEN)
```

The LU32I.D concatenates the least significant bit of the sign-extended 20-bit immediate value si20 to bits [31:0] of the general-purpose register rd, and writes the result to... It is loaded into the general-purpose register rd.

LU32I.D:

```
GR[rd] = {SignExtend(si20, 32), GR[rd][31:0]}
```

LU52I.D connects the 12-bit immediate value si12 to the [51:0] bits of data in the general-purpose register rj, and writes the result to the general-purpose register rd.

LU52I.D:

```
GR[rd] = {si12, GR[rj][51:0]}
```

The above instructions, together with the ORI instruction, are used to load immediate values of more than 12 bits into general-purpose registers.

2.2.1.5SLT[U]

Command format: slt rd, rj, rk

sltu rd, rj, rk

SLT compares the data in general-purpose register rj with the data in general-purpose register rk as signed integers. If the former is less than the latter...

If the condition is met, the value of the general-purpose register rd is set to 1; otherwise, it is set to 0.

SLT:

GR[rd] = (signed(GR[rj]) < signed(GR[rk])) ? 1 : 0

SLTU compares the data in general-purpose register rj with the data in general-purpose register rk as unsigned integers. If the former is less than...

The latter will set the value of the general-purpose register rd to 1, otherwise set it to 0.

SLTU:

GR[rd] = (unsigned(GR[rj]) < unsigned(GR[rk])) ? 1 : 0

The data bit width compared by SLT and SLTU is consistent with the bit width of the general-purpose registers of the machine being executed.

2.2.1.6 SLT[U]I instruction

format: slti		rd, rj, si12
	for the sake of	rd, rj, si12

SLTI treats the data in the general-purpose register rj and the data obtained after sign-extending the 12-bit immediate value si12 as a signed integer and performs a size comparison.

If the former is less than the latter, the value of the general-purpose register rd is set to 1; otherwise, it is set to 0.

SLTI:

tmp = SignExtend(si12, GRLEN)

GR[rd] = (signed(GR[rj]) < signed(tmp)) ? 1 : 0

SLTUI treats the data in the general-purpose register rj and the data obtained after sign-expanding the 12-bit immediate value si12 as an unsigned integer for comparison.

If the former is less than the latter, the value of the general-purpose register rd is set to 1; otherwise, it is set to 0.

For SLT:

tmp = SignExtend(si12, GRLEN)

GR[rd] = (unsigned(GR[rj]) < unsigned(tmp)) ? 1 : 0

The data bit width compared by SLTI and SLTUI is consistent with the bit width of the general-purpose registers of the machine being executed.

Please note that for SLTUI instructions, immediate values are still sign-extended.

2.2.1.7PCADDI, PCADDU12I, PCADDU18I, PCALAU12I

Command format: pcaddi		rd, si20
	pcaddu12i	rd, si20
	pcaddu18i	rd, si20
	pcalau12i	rd, si20

PCADDI appends 2 bits of 0 to the least significant bit of the 20-bit immediate value si20, then performs sign extension. The resulting data is then added to the program counter (PC) of the instruction.

The result is written to the general-purpose register rd.

PCADDI:

```
GR[rd] = PC + SignExtend({si20, 2'b0}, GRLEN)
```

PCADDU12I appends 12 bits of 0 to the least significant bit of the 20-bit immediate value si20, then performs sign extension, and adds the PC value of the instruction to the resulting data.

The sum is written to the general-purpose register rd.

PCADDU12I:

```
GR[rd] = PC + SignExtend({si20, 12'b0}, GRLEN)
```

PCADDU18I appends 18 bits of 0 to the least significant bit of the 20-bit immediate value si20, then performs sign extension, and adds the PC value of the instruction to the resulting data.

The sum is written to the general-purpose register rd.

PCADDU18I:

```
GR[rd] = PC + SignExtend({si20, 18'b0}, GRLEN)
```

PCALAU12I appends 12 bits of 0 to the least significant bit of the 20-bit immediate value si20, then performs sign extension, and adds the PC of the instruction to the resulting data.

The lowest 12 bits of the sum are erased and written to the general-purpose register rd.

PCALAU12I:

```
tmp = PC + SignExtend({si20, 12'b0}, GRLEN)
GR[rd] = {tmp[GRLEN-1:12], 12'b0}
```

The data bit width operated by the above instructions is consistent with the bit width of the general-purpose registers of the machine being executed.

2.2.1.8AND, OR, NOR, XOR, ANDN, ORN

```
Command format: and    rd, rj, rk
                or     rd, rj, rk
                nor    rd, rj, rk
                free   rd, rj, rk
                andn   rd, rj, rk
                orn    rd, rj, rk
```

The AND operation performs a bitwise logical AND operation between the data in general-purpose register rj and the data in general-purpose register rk, and writes the result to general-purpose register rd.

middle.

AND:

```
GR[rd] = GR[rj] & GR[rk]
```

OR performs a bitwise logical OR operation between the data in general-purpose register rj and the data in general-purpose register rk, and writes the result into general-purpose register rd.

OR:

```
GR[rd] = GR[rj] | GR[rk]
```

NOR performs a bitwise OR operation between the data in general-purpose register rj and the data in general-purpose register rk, and writes the result to general-purpose register rd.

middle.

NOR:

GR[rd] = ~(GR[rj] | GR[rk])

XOR performs a bitwise logical XOR operation between the data in general-purpose register rj and the data in general-purpose register rk, and writes the result to general-purpose register rd.

middle.

FREE:

GR[rd] = GR[rj] ^ GR[rk]

ANDN inverts the bits of the data in general-purpose register rk, then performs a bitwise AND operation with the data in general-purpose register rj, and writes the result to...

In the general-purpose register rd.

ANDN:

GR[rd] = GR[rj] & (~GR[rk])

ORN inverts the bits of the data in general-purpose register rk, then performs a bitwise OR operation with the data in general-purpose register rj, and writes the result to the register.

Use register rd.

ORN:

GR[rd] = GR[rj] | (~GR[rk])

The data bit width operated by the above instructions is consistent with the bit width of the general-purpose registers of the machine being executed.

2.2.1.9ANDI, ORI, HORI

Command format: andi rd, rj, ui12

OR rd, rj, ui12

choir rd, rj, ui12

ANDI performs a bitwise logical AND operation between the data in the general-purpose register rj and the zero-extended 12-bit immediate value, and writes the result into the general-purpose register rj.

In register rd.

ANDI:

GR[rd] = GR[rj] & ZeroExtend(ui12, GRLEN)

ORI performs a bitwise logical OR operation between the data in the general-purpose register rj and the zero-extended 12-bit immediate value, and writes the result into the general-purpose register rj.

In the register rd.

WHEN:

GR[rd] = GR[rj] | ZeroExtend(ui12, GRLEN)

XORI performs a bitwise logical XOR operation between the data in the general-purpose register rj and the zero-extended 12-bit immediate value, and writes the result into the register.

Use register rd.

CHORUS:

GR[rd] = GR[rj] ^ ZeroExtend(ui12, GRLEN)

The data bit width operated by the above instructions is consistent with the bit width of the general-purpose registers of the machine being executed.

2.2.1.10 NOP

The NOP instruction is an alias for the instruction "andi r0, r0, 0". Its function is simply to occupy a 4-byte instruction code location and increment the PC by 4; otherwise...

It will not change the processor state visible to any other software.

2.2.1.11 MUL.{W/D}, MULH.{W[U]/D[U]}

Command format: mul.w rd, rj, rk

mulh.w rd, rj, rk

mulh.wu rd, rj, rk

thank you rd, rj, rk

mulh.d rd, rj, rk

mulh.du rd, rj, rk

MUL.W multiplies the data in bits [31:0] of general-purpose register rj with the data in bits [31:0] of general-purpose register rk, and the [31:0] bits of the product are...

The data is then written to the general-purpose register rd after symbol expansion.

MUL.W:

product = signed(GR[rj][31:0]) * signed(GR[rk][31:0])

GR[rd] = SignExtend(product[31:0], GRLEN)

MULH.W treats the data in bits [31:0] of general-purpose register rj and the data in bits [31:0] of general-purpose register rk as signed numbers and multiplies them.

The result, with its [63:32] bits of data sign extended, is written into the general-purpose register rd.

MULH.W:

product = signed(GR[rj][31:0]) * signed(GR[rk][31:0])

GR[rd] = SignExtend(product[63:32], GRLEN)

MULH.WU multiplies the data in bits [31:0] of general-purpose register rj with the data in bits [31:0] of general-purpose register rk as unsigned numbers.

The [63:32] bits of data in the product result are sign-extended and written into the general-purpose register rd.

MULH.WU:

product = unsigned(GR[rj][31:0]) * unsigned(GR[rk][31:0])

GR[rd] = SignExtend(product[63:32], GRLEN)

MUL.D multiplies the data in bits [63:0] of general-purpose register rj with the data in bits [63:0] of general-purpose register rk, and the [63:0] bits of the product are...

Data is written to the general-purpose register rd.

MUL.D:

```
product = signed(GR[rj][63:0]) * signed(GR[rk][63:0])
GR[rd] = product[63:0]
```

MULH.D treats the data in bits [63:0] of general-purpose register rj and the data in bits [63:0] of general-purpose register rk as signed numbers and multiplies them. The product...

The result [127:64] bits of data are written into the general-purpose register rd.

MULH.D:

```
product = signed(GR[rj][63:0]) * signed(GR[rk][63:0])
GR[rd] = product[127:64]
```

MULH.DU multiplies the data in bits [63:0] of general-purpose register rj with the data in bits [63:0] of general-purpose register rk as unsigned numbers.

The [127:64] bits of the product result are written into the general-purpose register rd.

YOU CAN:

```
product = unsigned(GR[rj][63:0]) * unsigned(GR[rk][63:0])
GR[rd] = product[127:64]
```

2.2.1.12 MULW.DW[U]

Command format: mulw.dw rd, rj, rk

mulw.dwu rd, rj, rk

MULW.DW multiplies the data in bits [31:0] of general-purpose register rj with the data in bits [31:0] of general-purpose register rk as signed numbers, resulting in 64...

The result of the bit product is written into the general-purpose register rd.

MULW.DW:

```
product = signed(GR[rj][31:0]) * signed(GR[rk][31:0])
GR[rd] = product[63:0]
```

MULW.D.WU treats the data in bits [31:0] of general-purpose register rj and the data in bits [31:0] of general-purpose register rk as unsigned numbers and multiplies them.

The 64-bit product result is written to the general-purpose register rd.

MULW.D.WU:

```
product = unsigned(GR[rj][31:0]) * unsigned(GR[rk][31:0])
GR[rd] = product[63:0]
```

2.2.1.13 DIV.{W[U]/D[U]}, MOD.{W[U]/D[U]}

Command format: div.w rd, rj, rk

mod.w rd, rj, rk

div.wu rd, rj, rk

mod.wu rd, rj, rk

div.d rd, rj, rk

mod.d rd, rj, rk

you rd, rj, rk

mod.du rd, rj, rk

DIV.W and DIV.WU divide the data in bits [31:0] of general-purpose register rj by the data in bits [31:0] of general-purpose register rk, and the quotient is sign expanded.

After the expansion, it is written into the general-purpose register rd.

DIV.W:

quotient = signed(GR[rj][31:0]) / signed(GR[rk][31:0])

GR[rd] = SignExtend(quotient[31:0], GRLEN)

DIV.WU:

quotient = unsigned(GR[rj][31:0]) / unsigned(GR[rk][31:0])

GR[rd] = SignExtend(quotient[31:0], GRLEN)

MOD.W and MOD.WU divide the data in bits [31:0] of general-purpose register rj by the data in bits [31:0] of general-purpose register rk, and the remainder is the sign of the result.

The extended number is then written into the general-purpose register rd.

MOD.W:

remainder = signed(GR[rj][31:0]) % signed(GR[rk][31:0])

GR[rd] = SignExtend(remainder[31:0], GRLEN)

MOD.WU:

remainder = unsigned(GR[rj][31:0]) % unsigned(GR[rk][31:0])

GR[rd] = SignExtend(remainder[31:0], GRLEN)

On a LoongArch 64-bit compatible machine, when executing the DIV.W[U] and MOD.W[U] instructions, if the general-purpose register rj or rk contains...

If bits 63 to 31 of the stored data are not 0x0 or 0x1ffffffff, the result of the instruction execution can be any meaningless value.

DIV.D and DIV.DU divide the data in bits [63:0] of general-purpose register rj by the data in bits [63:0] of general-purpose register rk, and write the quotient into the general-purpose register rj.

Use register rd.

DIV.D:

GR[rd] = signed(GR[rj][63:0]) / signed(GR[rk][63:0])

DIV. YOU:

GR[rd] = unsigned(GR[rj][63:0]) / unsigned(GR[rk][63:0])

MOD.D and MOD.DU divide the data in bits [63:0] of general-purpose register rj by the data in bits [63:0] of general-purpose register rk, and write the remainder to...

Enter it into the general-purpose register rd.

MOD.D:

```
GR[rd] = signed(GR[rj][63:0]) % signed(GR[rk][63:0])
```

FOR YOU:

```
GR[rd] = unsigned(GR[rj][63:0]) % unsigned(GR[rk][63:0])
```

When performing division operations with DIV.W, MOD.W, DIV.D, and MOD.D, the operands are all treated as signed numbers. DIV.WU, MOD.WU,

When performing division operations with DIV.DU and MOD.DU, the source operands are both treated as unsigned numbers.

Each pair of instructions for finding the quotient/remainder is executed on DIV.W/MOD.W, DIV.WU/MOD.WU, DIV.D/MOD.D, and DIV.DU/MOD.DU.

The result of the calculation satisfies the following conditions: the remainder has the same sign as the dividend, and the absolute value of the remainder is less than the absolute value of the divisor.

When the divisor is 0, the result can be any value, but no exceptions will be triggered.

2.2.2 Shift Operation Instructions

2.2.2.1SLL.W, SRL.W, SRA.W, ROTR.W

```
Command format: sll.w     rd, rj, rk
                srl.w     rd, rj, rk
                sra.w     rd, rj, rk
                rotr.w    rd, rj, rk
```

SLL.W logically shifts the data in bits [31:0] of the general-purpose register rj to the left, and writes the sign extension of the shift result into the general-purpose register rd.

SLL.W:

```
tmp = SLL(GR[rj][31:0], GR[rk][4:0])
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

SRL.W logically right-shifts the data in bits [31:0] of the general-purpose register rj, and writes the sign-extended shift result into the general-purpose register rd.

SRL.W:

```
tmp = SRL(GR[rj][31:0], GR[rk][4:0])
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

SRA.W performs an arithmetic right shift of the data in bits [31:0] of the general-purpose register rj, and writes the sign-extended shift result into the general-purpose register rd.

SRA.W:

```
tmp = SRA(GR[rj][31:0], GR[rk][4:0])
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

ROTR.W cyclically shifts the data in bits [31:0] of the general-purpose register rj to the right, and writes the sign extension of the shift result into the general-purpose register rd.

ROTR.W:

```
tmp = ROTR(GR[rj][31:0], GR[rk][4:0])
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

The shift amount of the above shift instructions is the data in bits [4:0] of the general-purpose register rk, and is regarded as an unsigned number.

2.2.2.2SLLI.W, SRLI.W, SRAI.W, ROTRI.W

```
Command format: slli.w    rd, rj, ui5
                srli.w    rd, rj, ui5
                srai.w    rd, rj, ui5
                rotri.w   rd, rj, ui5
```

SLLI.W logically shifts the data in bits [31:0] of the general-purpose register rj to the left, and writes the sign extension of the shift result into the general-purpose register rd.

SLLI.W:

```
tmp = SLL(GR[rj][31:0], ui5)
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

SRLI.W logically right-shifts the data in bits [31:0] of the general-purpose register rj, and writes the sign extension of the shift result into the general-purpose register rd.

SRLI.W:

```
tmp = SRL(GR[rj][31:0], ui5)
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

SRAI.W performs an arithmetic right shift of the data in bits [31:0] of the general-purpose register rj, and writes the sign-extended shift result into the general-purpose register rd.

SRAI.W:

```
tmp = SRA(GR[rj][31:0], ui5)
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

ROTRI.W cyclically shifts the data in bits [31:0] of the general-purpose register rj to the right, and writes the sign extension of the shift result into the general-purpose register rd.

ROTRI.W:

```
tmp = ROTR(GR[rj][31:0], ui5)
GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

The shift amount of the above shift instructions is the 5-bit unsigned immediate value ui5 in the instruction code.

2.2.2.3SLL.D, SRL.D, SRA.D, ROTR.D

```
Command format: sll.d    rd, rj, rk
                srl.d    rd, rj, rk
                sra.d    rd, rj, rk
                rotr.d   rd, rj, rk
```

SLL.D logically shifts the data in bits [63:0] of the general-purpose register rj to the left, and writes the shift result into the general-purpose register rd.

SLL.D:

GR[rd] = SLL(GR[rj][63:0], GR[rk][5:0])

SRL.D logically right-shifts the data in bits [63:0] of the general-purpose register rj, and writes the shift result into the general-purpose register rd.

SRL.D:

GR[rd] =SRL(GR[rj][63:0], GR[rk][5:0])

SRA.D performs an arithmetic right shift of the data in bits [63:0] of the general-purpose register rj, and writes the shift result into the general-purpose register rd.

SRA.D:

GR[rd] = SRA(GR[rj][63:0], GR[rk][5:0])

ROTR.D cyclically shifts the data in bits [63:0] of the general-purpose register rj to the right, and writes the shift result into the general-purpose register rd.

ROTR.D:

GR[rd] = ROTR(GR[rj][63:0], GR[rk][5:0])

The shift amount of the above shift instructions is the data in bits [5:0] of the general-purpose register rk, and is regarded as an unsigned number.

2.2.2.4SLLI.D, SRLI.D, SRAI.D, ROTRI.D

```
Command format: slli.d   rd, rj, ui6
                srli.d   rd, rj, ui6
                srai.d   rd, rj, ui6
                rotri.d  rd, rj, ui6
```

SLLI.D logically shifts the data in bits [63:0] of the general-purpose register rj to the left, and writes the shift result into the general-purpose register rd.

SLLI.D:

GR[rd] = SLL(GR[rj][63:0], ui6)

SRLI.D logically right-shifts the data in bits [63:0] of the general-purpose register rj, and writes the shift result into the general-purpose register rd.

SRLI.D:

GR[rd] =SRL(GR[rj][63:0], ui6)

SRAI.D performs an arithmetic right shift of the data in bits [63:0] of the general-purpose register rj, and writes the shift result into the general-purpose register rd.

SRAI.D:

GR[rd] = SRA(GR[rj][63:0], ui6)

ROTRI.D cyclically shifts the data in bits [63:0] of the general-purpose register rj to the right, and writes the shift result into the general-purpose register rd.

ROTRI.D:

GR[rd] = ROTR(GR[rj][63:0], ui6)

The shift amount of the above shift instructions is the 6-bit unsigned immediate value ui6 in the instruction code.

2.2.3 Bit manipulation instructions

2.2.3.1EXT.W.{B/H}

Command format: ext.wb	rd, rj
ext.w.h	rd, rj

EXT.WB sign-extends the data bits [7:0] in general-purpose register rj and writes them into general-purpose register rd.

EXT.W.B:

GR[rd] = SignExtend(GR[rj][7:0], GRLEN)

EXT.WH writes the sign-extended data bits [15:0] in general-purpose register rj into general-purpose register rd.

EXT.W.H:

GR[rd] = SignExtend(GR[rj][15:0], GRLEN)

2.2.3.2CL{O/Z}.{W/D}, CT{O/Z}.{W/D}

Command format: clo.w	rd, rj	clo.d	rd, rj
clz.w	rd, rj	clz.d	rd, rj
cto.w	rd, rj	cto.d	rd, rj
ctz.w	rd, rj	ctz.d	rd, rj

For the data in bits [31:0] of the general-purpose register rj, CLO.W counts the number of consecutive "1" bits starting from bit 31 towards bit 0. The result is written into the general-purpose register rd.

CLO.W:

GR[rd] = CLO(GR[rj][31:0])

For the data in bits [31:0] of the general-purpose register rj, CLZ.W counts the number of consecutive "0" bits starting from bit 31 towards bit 0.

The result is written into the general-purpose register rd.

CLZ.W:

GR[rd] = CLZ(GR[rj][31:0])

CTO.W counts the number of consecutive "1" bits in bits [31:0] of the general-purpose register rj, starting from bit 0 and moving towards bit 31.

The result is written into the general-purpose register rd.

CTO.W:

GR[rd] = CTO(GR[rj][31:0])

For the data in bits [31:0] of the general-purpose register rj, CTZ.W counts the number of consecutive "0" bits starting from bit 0 and moving towards bit 31.

The result is written into the general-purpose register rd.

CTZ.W:

GR[rd] = CTZ(GR[rj][31:0])

CLO.D counts the number of consecutive "1" bits in the [63:0] bits of the general-purpose register rj, starting from bit 63 and moving towards bit 0.

The result is written into the general-purpose register rd.

CLO.D:

GR[rd] = CLO(GR[rj][63:0])

For the data in bits [63:0] of the general-purpose register rj, CLZ.D counts the number of consecutive "0" bits starting from bit 63 towards bit 0.

The result is written into the general-purpose register rd.

CLZ.D:

GR[rd] = CLZ(GR[rj][63:0])

CTO.D counts the number of consecutive "1" bits in bits [63:0] of the general-purpose register rj, starting from bit 0 and moving towards bit 63.

The result is written into the general-purpose register rd.

CTO.D:

GR[rd] = CTO(GR[rj][63:0])

For the data in bits [63:0] of the general-purpose register rj, CTZ.D counts the number of consecutive "0" bits starting from bit 0 and moving towards bit 63.

The result is written into the general-purpose register rd.

CTZ.D:

GR[rd] = CTZ(GR[rj][63:0])

2.2.3.3BYTEPICK.{W/D}

Command format: bytepick.w	rd, rj, rk, sa2	bytepick.d	rd, rj, rk, sa3

BYTEPICK.W concatenates bits [31:0] in general-purpose register rk with bits [31:0] in general-purpose register rj to form a 64-bit (8-byte) array.

The bit string is truncated by extracting four consecutive bytes starting from the leftmost byte sa2, and the resulting 32-bit bit string is then sign-extended and written into a general-purpose register.

rd in.

BYTEPICK.W:

```
tmp = {GR[rk][31:0], GR[rj][31:0]}
GR[rd] = SignExtend(tmp[8×(8-h2)-1 : 8×(4-h2)], GRLEN)
```

BYTEPICK.D concatenates bits [63:0] in general-purpose register rk with bits [63:0] in general-purpose register rj to form a 128-bit (16-word) concatenation.

The bit string of section (s) is truncated by extracting 8 consecutive bytes starting from the leftmost byte sa3, and the resulting 64-bit bit string is written into the general-purpose register rd.

BYTEPICK.D:

```
tmp = {GR[rk][63:0], GR[rj][63:0]}
GR[rd] = tmp[8×(16-sa3)-1 : 8×(8-sa3)]
```

2.2.3.4REVB.{2H/4H/2W/D}

Command format: revb.2h	rd, rj	revb.4h	rd, rj
		revb.2w	rd, rj
		revb.d	rd, rj

REVB.2H reverses the two bytes in bits [15:0] of the general-purpose register rj to form bits [15:0] of the intermediate result, and then sets the general-purpose register rj...

The two bytes in [31:16] are reversed to form the [31:16] bits of the intermediate result, and the 32-bit intermediate result sign extension is written into the general-purpose register rd.

REVB.2H:

```
tmp0 = {GR[rj][7:0], GR[rj][15:8]}
tmp1 = {GR[rj][23:16], GR[rj][31:24]}
GR[rd] = SignExtend({tmp1, tmp0}, GRLEN)
```

REVB.4H reverses the order of bits [15:0] in general-purpose register rj and writes them into bits [15:0] of general-purpose register rd. This resets the general-purpose register...

Write the two bytes from bits [31:16] of register rj in reverse order into bits [31:16] of general-purpose register rd, and write the two bytes from bits [47:32] of general-purpose register rj into general-purpose register rd.

Write the first byte in reverse order to bits [47:32] of the general-purpose register rd, and write the second byte in reverse order to bits [63:48] of the general-purpose register rj.

Use bits [63:48] of register rd.

REVB.4H:

```
tmp0 = {GR[rj][7:0], GR[rj][15:8]}
tmp1 = {GR[rj][23:16], GR[rj][31:24]}
tmp2 = {GR[rj][39:32], GR[rj][47:40]}
tmp3 = {GR[rj][55:48], GR[rj][63:56]}
GR[rd] = {tmp3, tmp2, tmp1, tmp0}
```

REVB.2W reverses the order of bits [31:0] in general-purpose register rj and writes them into bits [31:0] of general-purpose register rd, thus resetting the general-purpose register.

The four bytes in bits [63:32] of register rj are written in reverse order to bits [63:32] of general-purpose register rd.

REVB.2W:

```
tmp0 = {GR[rj][7:0], GR[rj][15:8], GR[rj][23:16], GR[rj][31:24]}
tmp1 = {GR[rj][39:32], GR[rj][47:40], GR[rj][55:48], GR[rj][63:56]}
GR[rd] = {tmp1, tmp0}
```

REVB.D writes the 8 bytes in bits [63:0] of general-purpose register rj into general-purpose register rd in reverse order.

REVB.D:

```
GR[rd] = {GR[rj][7:0], GR[rj][15:8], GR[rj][23:16], GR[rj][31:24], GR[rj][3
9:32], GR[rj][47:40], GR[rj][55:48], GR[rj][63:56]}
```

2.2.3.5REVH.{2W/D}

Command format: revh.2w rd, rj

revh.d rd, rj

REVH.2W reverses the order of bits [31:0] in general-purpose register rj and writes them into bits [31:0] of general-purpose register rd.

The two half-words in bits [63:32] of register rj are written in reverse order to bits [63:32] of general-purpose register rd.

REVH.2W:

```
tmp0 = {GR[rj][15:0], GR[rj][31:16]}
tmp1 = {GR[rj][47:32], GR[rj][63:48]}
GR[rd] = {tmp1, tmp0}
```

REVH.D writes the four half-words in bits [63:0] of general-purpose register rj into general-purpose register rd in reverse order.

REVH.D:

```
GR[rd] = { GR[rj][15:0], GR[rj][31:16], GR[rj][47:32], GR[rj][63:48]}
```

2.2.3.6BITREV.{4B/8B}

Command format: bitrev.4b rd, rj bitrev.8b rd, rj

BITREV.4B reverses the 8 bits in bits [7:0] of general-purpose register rj to form the intermediate result's bits [7:0], reverses the 8 bits in bits [15:8] of general-purpose register rj to form the intermediate result's bits [15:8], reverses the 8 bits in bits [23:16] of general-purpose register rj to form the intermediate result's bits [23:16], and reverses the 8 bits in bits [31:24] of general-purpose register rj to form the intermediate result's bits [31:24].

The 32-bit intermediate result sign extension is written into the general-purpose register rd.

BITREV.4B:

```
bstr32[31:24] = BITREV(GR[rj][31:24])
bstr32[23:16] = BITREV(GR[rj][23:16])
bstr32[15: 8] = BITREV(GR[rj][15: 8])
bstr32[ 7: 0] = BITREV(GR[rj][ 7: 0])
GR[rd] = SignExtend(bstr32, GRLEN)
```

BITREV.8B reverses the order of bits [7:0] in general-purpose register rj and writes them into bits [7:0] of general-purpose register rd, thus resetting the general-purpose register.

Write the 8 bits from bits [15:8] of register rj in reverse order into bits [15:8] of general-purpose register rd, and write the 8 bits from bits [23:16] of general-purpose register rj into general-purpose register rd.

Write bits [23:16] of the general-purpose register rd in reverse order, and write 8 bits [31:24] of the general-purpose register rj in reverse order.

Bits [31:24] of register rd will be written into bits [39:32] of general-purpose register rj in reverse order.

Reverse the order of bits [47:40] in general-purpose register rj and write them into bits [47:40] of general-purpose register rd. Then, write bits [55:48] from general-purpose register rj...

The 8 bits in the first position are reversed and written into bits [55:48] of the general-purpose register rd. The 8 bits in bits [63:56] of the general-purpose register rj are reversed.

Write the column to bits [63:56] of the general-purpose register rd.

BITREV.8B:

```
    GR[rd][63:56] = BITREV(GR[rj][63:56])
    GR[rd][55:48] = BITREV(GR[rj][55:48])
    GR[rd][47:40] = BITREV(GR[rj][47:40])
    GR[rd][39:32] = BITREV(GR[rj][39:32])
    GR[rd][31:24] = BITREV(GR[rj][31:24])
    GR[rd][23:16] = BITREV(GR[rj][23:16])
    GR[rd][15: 8] = BITREV(GR[rj][15: 8])
    GR[rd][ 7: 0] = BITREV(GR[rj][ 7: 0])
```

2.2.3.7BITREV.{W/D}

Command format: bitrev.w rd, rj bitrev.d rd, rj

BITREV.W reverses the 32 bits [31:0] in general-purpose register rj to form the intermediate result [31:0] bits.

The fruit symbol extension is written into the general-purpose register rd.

BITREV.W:

```
    bstr32[31:0] = BITREV(GR[rj][31:0])
    GR[rd] = SignExtend(bstr32, GRLEN)
```

BITREV.D reverses the order of 64 bits in bits [63:0] of general-purpose register rj and writes them into general-purpose register rd.

BITREV.D:

```
    GR[rd] = BITREV(GR[rj][63:0])
```

2.2.3.8BSTRINS.{W/D}

Instruction format: bstrins.w rd, rj, msbw, lsbw BSTRINS.W bstrins.d rd, rj, msbd, lsbd

replaces the [msbw:lsbw] bits in the lowest 32 bits of the general-purpose register rd with the [msbw-lsbw:0] bits in the general-purpose register rj, resulting in...

The obtained 32-bit result is sign-extended and written into the general-purpose register rd.

BSTRINS.W:

```
    maskv = {(msbw-lsbw+1){1'b1}}<<lsbw
    bstr32 = GR[rd][31:0]&~maskv[31:0] | (GR[rj][msbw-lsbw:0]<<lsbw)&maskv[31:
0]
    GR[rd] = SignExtend(bstr32[31:0], GRLEN)
```

BSTRINS.D replaces the [msbd:lsbd] bits in the general-purpose register rd with the [msbd-lsbd:0] bits in the general-purpose register rj. The remaining bits remain unchanged.

BSTRINS.D:

```
    maskv = {(msbd-lsbd+1){1’b1}}<<lsbd
    GR[rd] = GR[rd][63:0]&~maskv[63:0] | (GR[rj][msbd-lsbd:0]<<lsbd)&maskv[63:
0]
```

2.2.3.9BSTRPICK.{W/D}

Command format: bstrpick.w rd, rj, msbw, lsbw   bstrpick.d   rd, rj, msbd, lsbd

BSTRPICK.W extracts the [msbw:lsbw] bits from the general-purpose register rj, zero-extends them to 32 bits, and writes the resulting 32-bit intermediate result after sign extension. Enter it into the general-purpose register rd.

BSTRPICK.W:

```
    bstr32[31:0] = ZeroExtend(GR[rj][msbw:lsbw], 32)
    GR[rd] = SignExtend(bstr32[31:0], GRLEN)
```

BSTRPICK.D extracts the [msbd:lsbd] bits from the general-purpose register rj, extends them to 64 bits, and writes them into the general-purpose register rd.

BSTRPICK.D:

```
    GR[rd] = ZeroExtend(GR[rj][msbd:lsbd], 64)
```

2.2.3.10 MASKEKZ, MASKNEZ

Command format: maskeqz   rd, rj, rk

masknez   rd, rj, rk

The MASKEQZ and MASKNEZ instructions perform conditional assignment operations.

When MASKEQZ executes, if the value of the general-purpose register rk is equal to 0, then the general-purpose register rd is set to all zeros; otherwise, it is assigned the value rj. The value of the register.

MASKEQZ:

```
    GR[rd] = (GR[rk]==0) ? 0 : GR[rj]
```

When MASKNEZ executes, if the value of the general-purpose register rk is not equal to 0, then the general-purpose register rd is set to all zeros; otherwise, it is assigned the value rj. The value of the register.

MASKNEZ:

```
    GR[rd] = (GR[rk]!=0) ? 0 : GR[rj]
```

2.2.4 Transfer Command

2.2.4.1BEQ, BNE, BLT[U], BGE[U]

```
Command format: beq     rj, rd, offs16
                see     rj, rd, offs16
                blt     rj, rd, offs16
                bge     rj, rd, offs16
                blue    rj, rd, offs16
                blue    rj, rd, offs16
```

BEQ compares the values of general-purpose register rj and general-purpose register rd. If they are equal, it jumps to the target address; otherwise, it does not jump.

```
BEQ:
    if GR[rj]==GR[rd] :
        PC = PC + SignExtend({offs16, 2'b0}, GRLEN)
```

BNE compares the values of general-purpose register rj and general-purpose register rd. If they are not equal, it jumps to the target address; otherwise, it does not jump.

```
BNE:
    if GR[rj]!=GR[rd] :
        PC = PC + SignExtend({offs16, 2'b0}, GRLEN)
```

BLT compares the values of general-purpose register rj and general-purpose register rd as signed numbers; if the former is less than the latter, it jumps to the target location.

The URL must be provided; otherwise, the user will not be redirected.

```
BLT:
    if signed(GR[rj]) < signed(GR[rd]) :
        PC = PC + SignExtend({offs16, 2'b0}, GRLEN)
```

BGE compares the values of general-purpose register rj and general-purpose register rd as signed numbers; if the former is greater than or equal to the latter, it jumps to the target.

Specify the address; otherwise, do not redirect.

```
BGE:
    if signed(GR[rj]) >= signed(GR[rd]) :
        PC = PC + SignExtend({offs16, 2'b0}, GRLEN)
```

BLTU treats the values of general-purpose register rj and general-purpose register rd as unsigned numbers and compares them; if the former is less than the latter, it jumps to the target.

The address must be displayed; otherwise, the user will not be redirected.

```
BLTU:
    if unsigned(GR[rj]) < unsigned(GR[rd]) :
        PC = PC + SignExtend({offs16, 2'b0}, GRLEN)
```

BGEU compares the values of general-purpose register rj and general-purpose register rd as unsigned numbers; if the former is greater than or equal to the latter, it jumps to...

Target address; otherwise, do not redirect.

BGEU:

```
if unsigned(GR[rj]) >= unsigned(GR[rd]) :
    PC = PC + SignExtend({offs16, 2'b0}, GRLEN)
```

The jump target address for the above six branch instructions is calculated by logically shifting the 16-bit immediate value off16 in the instruction code left by 2 bits before recalculating.

The offset value is extended by the branch instruction, and the resulting offset value is added to the PC of that branch instruction.

However, it should be noted that if the above instructions are written by directly filling in the offset value when writing the assembly code, the immediate value in the assembly representation should be...

Enter the offset value in bytes, which is offs16<<2 in the instruction code.

2.2.4.2BEQZ, BNEZ

Command format: beqz rj, offs21

bnez rj, offs21

BEQZ checks the value of the general-purpose register rj. If the value is 0, it jumps to the target address; otherwise, it does not jump.

BEQZ:

```
if GR[rj]==0 :
    PC = PC + SignExtend({offs21, 2'b0}, GRLEN)
```

BNEZ checks the value of the general-purpose register rj. If the value is not equal to 0, it jumps to the target address; otherwise, it does not jump.

BNEZ:

```
if GR[rj]!=0 :
    PC = PC + SignExtend({offs21, 2'b0}, GRLEN)
```

The jump target address of the two branch instructions mentioned above is obtained by logically shifting the 21-bit immediate value offs21 in the instruction code left by 2 bits and then sign-extending it.

The resulting offset value is added to the PC of the branch instruction.

However, it should be noted that if the above instructions are written by directly filling in the offset value when writing the assembly code, the immediate value in the assembly representation should be...

Enter the offset value in bytes, which is offs21<<2 in the instruction code.

2.2.4.3B

Command format: b offs26

B unconditionally jumps to the target address. The target address is obtained by logically left-shifting the 26-bit immediate value `offs26` in the instruction code by 2 bits.

The sign is extended, and the resulting offset value is added to the PC of the branch instruction.

B:

```
PC = PC + SignExtend({offs26, 2'b0}, GRLEN)
```

It is important to note that if this instruction is written by directly filling in the offset value during assembly, the immediate value in the assembly representation should be filled in with the offset value.

The offset value in bytes, i.e., offs26<<2 in the instruction code.

2.2.4.4BL

Command format: bl offs26

BL jumps unconditionally to the target address and simultaneously writes the PC value of the instruction plus 4 into general-purpose register r1.

The jump target address of this instruction is obtained by logically left-shifting the 26-bit immediate value offs26 in the instruction code by 2 bits and then sign-extending it.

Add the PC value to the branch instruction.

```
BL:
    GR[1] = PC + 4
    PC = PC + SignExtend({offs26, 2'b0}, GRLEN)
```

In the LA ABI, general-purpose register r1 is used as the return address register ra.

It is important to note that if this instruction is written by directly filling in the offset value during assembly, the immediate value in the assembly representation should be filled in with the offset value.

The offset value in bytes, i.e., offs26<<2 in the instruction code.

2.2.4.5JIRL

Command format: jirl rd, rj, offs16

JIRL jumps unconditionally to the target address and simultaneously writes the PC value of the instruction plus 4 into the general-purpose register rd.

The jump target address of this instruction is obtained by logically left-shifting the 16-bit immediate value `offs16` in the instruction code by 2 bits and then sign-extending it.

The value is added to the value in the general-purpose register rj.

```
JIRL:
    GR[rd] = PC + 4
    PC = GR[rj] + SignExtend({offs16, 2'b0}, GRLEN)
```

When rd equals 0, JIRL functions as a regular non-call indirect jump instruction.

JIRLs with rd equal to 0, rj equal to 1, and offs16 equal to 0 are often used as indirect jumps back from calls.

It is important to note that if this instruction is written by directly filling in the offset value during assembly, the immediate value in the assembly representation should be filled in with the offset value.

The offset value in bytes, i.e., offs16<<2 in the instruction code.

2.2.5 Ordinary Memory Access Instructions

2.2.5.1LD.{B[U]/H[U]/W[U]/D}, ST.{B/H/W/D}

Command format:	ld.b	rd, rj, si12
	ld.h	rd, rj, si12
	ld.w	rd, rj, si12
	ld.d	rd, rj, si12
	ld.bu	rd, rj, si12
	ld.hu	rd, rj, si12
	ld.wu	rd, rj, si12
	st.b	rd, rj, si12
	st.h	rd, rj, si12
	st.w	rd, rj, si12
	st.d	rd, rj, si12

LD.{B/H/W} retrieves a byte/half-word/word of data from memory, signs-extends it, and writes it to the general-purpose register rd. LD.D retrieves a...

Double-word data is written to the general-purpose register rd.

LD.B:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
byte = MemoryLoad(paddr, BYTE)
GR[rd] = SignExtend(byte, GRLEN)
```

LD.H:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
halfword = MemoryLoad(paddr, HALFWORD)
GR[rd] = SignExtend(halfword, GRLEN)
```

LD.W:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
word = MemoryLoad(paddr, WORD)
GR[rd] = SignExtend(word, GRLEN)
```

LD.D:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
GR[rd] = MemoryLoad(paddr, DOUBLEWORD)
```

LD.{BU/HU/WU} retrieves one byte/half-word/word of data from memory, zero-extends it, and writes it to the general-purpose register rd.

LD.BU:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
byte = MemoryLoad(paddr, BYTE)
GR[rd] = ZeroExtend(byte, GRLEN)
```

LD.HU:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
halfword = MemoryLoad(paddr, HALFWORD)
GR[rd] = ZeroExtend(halfword, GRLEN)
```

LD.WU:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
word = MemoryLoad(paddr, WORD)
GR[rd] = ZeroExtend(word, GRLEN)
```

ST.{B/H/W/D} writes the data in bits [7:0]/[15:0]/[31:0]/[63:0] from the general-purpose register rd into memory.

ST.B:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][7:0], paddr, BYTE)
```

ST.H:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][15:0], paddr, HALFWORD)
```

ST.W:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][31:0], paddr, WORD)
```

ST.D:

```
vaddr = GR[rj] + SignExtend(si12, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][63:0], paddr, DOUBLEWORD)
```

The memory address of the above instruction is calculated by adding the value in the general-purpose register rj to the sign-extended 12-bit immediate value si12.

For the LD.{H[U]/W[U]/D} and ST.{B/H/W/D} instructions, regardless of the hardware implementation or environment configuration, as long as they access memory... If the memory address is naturally aligned, no unaligned exception will be triggered; however, if the hardware implementation supports unaligned memory access and If the current operating environment is configured to allow unaligned memory access, then the unaligned exception will not be triggered; otherwise, the unaligned exception will be triggered.

2.2.5.2LDX.{B[U]/H[U]/W[U]/D}, STX.{B/H/W/D}

Command format:

```
ldx.b     rd, rj, rk
ldx.h     rd, rj, rk
ldx.w     rd, rj, rk
ldx.d     rd, rj, rk
ldx.bu    rd, rj, rk
ldx.hu    rd, rj, rk
ldx.wu    rd, rj, rk
stx.b     rd, rj, rk
stx.h     rd, rj, rk
stx.w     rd, rj, rk
stx.d     rd, rj, rk
```

LDX.{B/H/W} retrieves one byte/half-word/word of data from memory, signs-extends it, and writes it to the general-purpose register rd. LDX.D retrieves the data from memory... A double word of data is written to the general-purpose register rd.

LDX.B:

```
    vaddr = GR[rj] + GR[rk]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    byte = MemoryLoad(paddr, BYTE)
    GR[rd] = SignExtend(byte, GRLEN)
```

LDX.H:

```
    vaddr = GR[rj] + GR[rk]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    halfword = MemoryLoad(paddr, HALFWORD)
    GR[rd] = SignExtend(halfword, GRLEN)
```

LDX.W:

```
    vaddr = GR[rj] + GR[rk]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    word = MemoryLoad(paddr, WORD)
    GR[rd] = SignExtend(word, GRLEN)
```

LDX.D:

```
vaddr = GR[rj] + GR[rk]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
GR[rd] = MemoryLoad(paddr, DOUBLEWORD)
```

LDX.{BU/HU/WU} retrieves one byte/half-word/word of data from memory, zero-extends it, and writes it to the general-purpose register rd.

LDX.BU:

```
vaddr = GR[rj] + GR[rk]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
byte = MemoryLoad(paddr, BYTE)
GR[rd] = ZeroExtend(byte, GRLEN)
```

LDX.HU:

```
vaddr = GR[rj] + GR[rk]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
halfword = MemoryLoad(paddr, HALFWORD)
GR[rd] = ZeroExtend(halfword, GRLEN)
```

LDX.WU:

```
vaddr = GR[rj] + GR[rk]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
word = MemoryLoad(paddr, WORD)
GR[rd] = ZeroExtend(word, GRLEN)
```

STX.{B/H/W/D} writes the data in bits [7:0]/[15:0]/[31:0]/[63:0] from the general-purpose register rd into memory.

STX.B:

```
vaddr = GR[rj] + GR[rk]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][7:0], paddr, BYTE)
```

STX.H:

```
vaddr = GR[rj] + GR[rk]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][15:0], paddr, HALFWORD)
```

STX.W:

```
vaddr = GR[rj] + GR[rk]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][31:0], paddr, WORD)
```

STX.D:

```
vaddr = GR[rj] + GR[rk]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][63:0], paddr, DOUBLEWORD)
```

The memory access address of the above instruction is calculated by adding the value in general-purpose register rj to the value in general-purpose register rk.

For LDX.{H[U]/W[U]/D} and STX.{B/H/W/D} instructions, regardless of the hardware implementation or environment configuration, as long as their access... If the memory address is naturally aligned, no unaligned exception will be triggered; however, if the hardware implementation supports unaligned access, a memory address that is not naturally aligned will trigger an unaligned exception. If the current operating environment is configured to allow unaligned memory access, then the unaligned exception will not be triggered; otherwise, the unaligned exception will be triggered.

2.2.5.3LDPTR.{W/D}, STPTR.{W/D}

Command format: ldptr.w	rd, rj, si14
ldptr.d	rd, rj, si14
stptr.w	rd, rj, si14
stptr.d	rd, rj, si14

LDPTR.W retrieves a word of data from memory, signs-extends it, and writes it to the general-purpose register rd. LDPTR.D retrieves a double word of data from memory. Data is written to the general-purpose register rd.

LDPTR.W:

```
vaddr = GR[rj] + SignExtend({si14, 2'b0}, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
word = MemoryLoad(paddr, WORD)
GR[rd] = SignExtend(word, GRLEN)
```

LDPTR.D:

```
vaddr = GR[rj] + SignExtend({si14, 2'b0}, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
GR[rd] = MemoryLoad(paddr, DOUBLEWORD)
```

STPTR.{W/D} writes the data in bits [31:0]/[63:0] of the general-purpose register rd into memory.

STPTR.W:

```
vaddr = GR[rj] + SignExtend({si14, 2'b0}, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][31:0], paddr, WORD)
```

STPTR.D:

```
vaddr = GR[rj] + SignExtend({si14, 2'b0}, GRLEN)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][63:0], paddr, DOUBLEWORD)
```

The memory access address of the above instruction is calculated by logically shifting the 14-bit immediate value si14 left by 2 bits, then sign-extending it, and finally multiplying it by the general-purpose register rj.

The values in the above instructions are summed. It's important to note that the immediate address offset values in the assembly representation of the above instructions are in bytes; that is, their values refer to...

In the instruction code, si14<<2.

For the LDPTR.{W/D} and STPTR.{W/D} instructions, regardless of the hardware implementation or environment configuration, as long as their memory access address is...

Naturally aligned memory will not trigger unaligned exceptions; when the memory access address is not naturally aligned, if the hardware implementation supports unaligned memory access and the current...

If the computing environment is configured to allow unaligned memory access, then the unaligned exception will not be triggered; otherwise, the unaligned exception will be triggered.

The LDPTR.{W/D} and STPTR.{W/D} instructions are used in conjunction with the ADDU16I.D instructions to accelerate GOT-based code in position-independent environments.

Access to the table.

2.2.5.4PRELD

Command format: preld hint, rj, si12

PRELD prefetches a cache line of data from memory into the cache. Its memory access address is calculated by adjusting the value in the general-purpose register rj.

The value is summed with the sign-extended 12-bit immediate value si12. The memory access address falls within the cache line to be prefetched.

The `PRETLD` instruction provides hints to the processor about the type of data to prefetch and which cache level the fetched data should be placed in. Hints range from 0 to 31, with 32 possible values.

Select a value. Currently, hint=0 is defined as load prefetching to the first-level data cache, and hint=8 is defined as store prefetching to the first-level data cache. Other hints...

The meaning of the value is not yet defined; the processor will treat it as a NOP instruction during execution.

If the cache attribute of the memory address accessed by the PRELD instruction is not cached, then the instruction cannot perform a memory access operation and is treated as a NOP instruction.

deal with.

The PRELD instruction will not trigger any MMU or address-related exceptions.

2.2.5.5PRELDX

Command format: preldx hint, rj, rk

The PRELDX instruction prefetches data continuously from memory into the cache according to configuration parameters. The data prefetched continuously starts from the specified base address.

The first few data blocks (blocks) are spaced at stride intervals and have a length of block_size. The base address is calculated using a general method.

Bits [63:0] in register rj are added to bits [15:0] in the sign-extended general-purpose register rk. Bit [16] in general-purpose register rk is the address sequence ascending/descending flag, where 0

indicates ascending order and 1 indicates descending order. The value of bits [25:20] in general-purpose register rk is block_size - 1.

The basic unit of block_size is 16 bytes, therefore the maximum length of a single block is 1KB. The values of bits [39:32] in the general-purpose register rk...

The value is block_num-1, therefore a single instruction can prefetch a maximum of 256 blocks. The values in bits [59:44] of the general-purpose register rk are treated as signed numbers.

The stride between adjacent blocks is defined, and the basic unit of the stride is 1 byte.

The `PRELDX` instruction contains hints to the processor about the type of data to prefetch and which cache level the fetched data should be placed in. There are 32 hints, ranging from 0 to 31.

Optional values. Currently, hint=0 defines load prefetching to the level 1 data cache, hint=2 defines load prefetching to the level 3 cache, and hint=8 defines...

This prefetches data into the L1 cache for the store. The meanings of the other hint values are currently undefined; the processor will treat them as NOP instructions during execution.

If the cache attribute of the memory address accessed by the PRELDX instruction is not cached, then the instruction cannot perform a memory access operation and is treated as a NOP instruction.

Order to process.

The PRELDX instruction will not trigger any MMU or address-related exceptions.

2.2.6 Boundary Check Memory Access Command

2.2.6.1LD{GT/LE}.{B/H/W/D}, ST{GT/LE}.{B/H/W/D}

```
Command format: ldgt.b     rd, rj, rk
                ldgt.h     rd, rj, rk
                ldgt.w     rd, rj, rk
                ldgt.d     rd, rj, rk
                ldle.b     rd, rj, rk
                ldle.h     rd, rj, rk
                ldle.w     rd, rj, rk
                ldle.d     rd, rj, rk
                stgt.b     rd, rj, rk
                stgt.h     rd, rj, rk
                stgt.w     rd, rj, rk
                stgt.d     rd, rj, rk
                stle.b     rd, rj, rk
                stle.h     rd, rj, rk
                stle.w     rd, rj, rk
                stle.d     rd, rj, rk
```

LDGT/LDLE.{B/H/W} retrieves a byte/half-word/word of data from memory, sign-extends it, and writes it to the general-purpose register rd.

LDGT/LDLE.D retrieves a double word of data from memory and writes it to the general-purpose register rd.

STGT/STLE.{B/H/W/D} writes the data in bits [7:0]/[15:0]/[31:0]/[63:0] from the general-purpose register rd into memory.

The memory access addresses for the above instructions come directly from the value in the general-purpose register rj. All memory access addresses for the above instructions require natural alignment; otherwise, [the memory will be...].

Triggering an unaligned exception.

When the LDGT.{B/H/W/D} and STGT.{B/H/W/D} instructions are executed, they check whether the value in the general-purpose register rj is greater than that in the general-purpose register rk.

If the condition is not met, the memory access operation is terminated and a boundary check exception is triggered. The instructions LDLE.{B/H/W/D} and STLE.{B/H/W/D}...

During execution, it checks whether the value in general-purpose register rj is less than or equal to the value in general-purpose register rk. If the condition is not met, the memory access operation is terminated.

And trigger a boundary check error exception.

LDGT.B:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    byte = MemoryLoad(paddr, BYTE)
    GR[rd] = SignExtend(byte, GRLEN)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

LDGT.H:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    halfword = MemoryLoad(paddr, HALFWORD)
    GR[rd] = SignExtend(halfword, GRLEN)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

LDGT.W:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    word = MemoryLoad(paddr, WORD)
    GR[rd] = SignExtend(word, GRLEN)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

LDGT.D:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    GR[rd] = MemoryLoad(paddr, DOUBLEWORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

LDLE.B:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    byte = MemoryLoad(paddr, BYTE)
    GR[rd] = SignExtend(byte, GRLEN)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

LDLE.H:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    halfword = MemoryLoad(paddr, HALFWORD)
    GR[rd] = SignExtend(halfword, GRLEN)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

LDLE.W:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    word = MemoryLoad(paddr, WORD)
    GR[rd] = SignExtend(word, GRLEN)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

LDLE.D:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    GR[rd] = MemoryLoad(paddr, DOUBLEWORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

STGT.B:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    MemoryStore(GR[rd][7:0], paddr, BYTE)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

STGT.H:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    MemoryStore(GR[rd][15:0], paddr, HALFWORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

STGT.W:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    MemoryStore(GR[rd][31:0], paddr, WORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

STGT.D:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    MemoryStore(GR[rd][63:0], paddr, DOUBLEWORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

STLE.B:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    MemoryStore(GR[rd][7:0], paddr, BYTE)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

STLE.H:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    MemoryStore(GR[rd][15:0], paddr, HALFWORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

STLE.W:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    MemoryStore(GR[rd][31:0], paddr, WORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

STLE.D:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    MemoryStore(GR[rd][63:0], paddr, DOUBLEWORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

2.2.7.1AM{SWAP/ADD/AND/OR/XOR/MAX/MIN}[_DB].{W/D}, AM{MAX/MIN}[_DB].{WU/DU}

Command format:	amswap.w	rd, rk, rj	amswap_db.w	rd, rk, rj
	amswap.d	rd, rk, rj	amswap_db.d	rd, rk, rj
	amadd.w	rd, rk, rj	amadd_db.w	rd, rk, rj
	paragraph d	rd, rk, rj	amadd_db.d	rd, rk, rj
	amand.w	rd, rk, rj	amand_db.w	rd, rk, rj
	almond.d	rd, rk, rj	amand_db.d	rd, rk, rj
	amor.w	rd, rk, rj	amor_db.w	rd, rk, rj
	love.d	rd, rk, rj	amor_db.d	rd, rk, rj
	amxor.w	rd, rk, rj	amxor_db.w	rd, rk, rj
	amxor.d	rd, rk, rj	amxor_db.d	rd, rk, rj
	ammax.w	rd, rk, rj	ammax_db.w	rd, rk, rj
	ammax.d	rd, rk, rj	ammax_db.d	rd, rk, rj
	admin.w	rd, rk, rj	admin_db.w	rd, rk, rj
	ammin.d	rd, rk, rj	admin_db.d	rd, rk, rj
	ammax.wu	rd, rk, rj	ammax_db.wu	rd, rk, rj
	ammax.du	rd, rk, rj	ammax_db.du	rd, rk, rj
	amen.i	rd, rk, rj	admin_db.wu	rd, rk, rj
	ammyn.du	rd, rk, rj	ammin_db.du	rd, rk, rj

AM* atomic memory access instructions can atomically complete a sequence of "read-modify-write" operations on a specific memory location. Specifically, they atomically access memory... The old value at the specified address is retrieved and written to the general-purpose register rd. Simultaneously, some simple operations are performed between this old value in memory and the value in the general-purpose register rd. The operation is performed, and then the result is written back to the specified memory address. The entire "read-modify-write" process is atomic, meaning that the operation is performed atomically. During instruction execution, from the time the data is returned from the memory access read operation to the time when the effect of the memory access write operation is globally visible, the processor executing the instruction... No other memory access/write operations were performed, no exceptions were triggered, and no other processor cores or cache coherence modules accessed the instruction. The effect of write operations on the cache line containing the image is globally visible.

The memory address accessed by AM* atomic memory access instructions is the value of the general-purpose register rj. The memory address accessed by AM* atomic memory access instructions always requires natural alignment. If this condition is not met, an unaligned exception will be triggered.

Atomic memory access instructions ending in .W and .WU read, write, and perform intermediate operations with a data width of 32 bits. Instructions ending in .D and .DU... Atomic memory access instructions read from and write to memory, as well as perform intermediate operations, using 64-bit data. Regardless of whether the instruction ends in .W or .WU, atomic memory access instructions... Each word of data retrieved from memory is written to the general-purpose register rd after sign extension.

The AMSWAP[_DB].{W/D} instruction writes a new value to memory from the general-purpose register rk. The AMADD[_DB].{W/D} instruction writes to memory. The new value is the result of adding the old value in memory to the value in the general-purpose register rk. The new value written to memory by the AMAND[_DB].{W/D} instruction comes from...

The result of a bitwise logical AND operation between the old value in memory and the value in the general-purpose register rk. The new value written to memory by the AMOR[_DB].{W/D} instruction comes from memory. The result of a bitwise logical OR operation between the old value and the value in the general-purpose register rk. The new value written to memory by the AMXOR[_DB].{W/D} instruction comes from the old value in memory. The value is the result of a bitwise XOR operation between the value in the general-purpose register rk and the value in the register rk. The new value written to memory by the AMMAX[_DB].{W/D} instruction is the old value in memory. The value is compared with the value in the general-purpose register rk, and the result is the maximum value obtained after a signed number comparison. The AMMIN[_DB].{W/D} instruction writes the value to memory. The new value is the minimum value obtained by comparing the old value in memory with the value in the general-purpose register rk as a signed number. The AMMAX[_DB].{WU/DU} instruction writes a new value to memory by treating the old value in memory and the value in the general-purpose register rk as an unsigned number and performing a large operation. The maximum value obtained after a small comparison. The new value written to memory by the AMMIN[_DB].{WU/DU} instruction is the old value in memory compared with the value in the general-purpose register rk. The value is the minimum value obtained after comparing the size of unsigned numbers.

The AM*_DB.W[U]/D[U] instruction, in addition to performing the aforementioned atomic operation sequence, also implements data barrier functionality. That is, when such atoms... Before a memory access instruction is allowed to execute, all memory access operations preceding that atomic memory access instruction in the same processor core must have been completed; only then can execution proceed. Once such an atomic memory access instruction has been executed, all memory access operations following that atomic memory access instruction in the same processor core are allowed to be executed.

AM* atomic memory access instructions will not be triggered unless the register numbers rd and rj are the same.

If the register numbers rd and rk are the same in an AM* atomic memory access instruction, the execution result is uncertain. Please ensure that the software avoids this situation.

2.2.7.2AM{SWAP/ADD}[_DB].{B/H}

Command format: amswap.b	rd, rk, rj	amswap_db.b	rd, rk, rj
amswap.h	rd, rk, rj	amswap_db.h	rd, rk, rj
amadd.b	rd, rk, rj	amadd_db.b	rd, rk, rj
amadd.h	rd, rk, rj	amadd_db.h	rd, rk, rj

LoongArch V1.1 Commands

AM{SWAP/ADD}[_DB].{B/H} and AM{SWAP/ADD}[_DB].{W/D} are both atomic memory access instructions, capable of being completed atomically. The main difference between a "read-modify-write" operation sequence for a memory unit lies in whether the accessed data is a byte/half-word or a word/double word.

AM{SWAP/ADD}[_DB].{B/H} retrieves the old byte/half-word value at the specified memory address, performs sign extension, and writes it to a general-purpose register. rd, and simultaneously swap or add the old value in memory with the byte/half-word value of bits rk[7:0]/[15:0] in the general-purpose register, then... The obtained byte/half-word result is written back to the specified memory address. The entire "read-modify-write" process is atomic, meaning that the instruction executes... During the period from when the data from the memory access read operation is returned to when the global effect of the memory access write operation is visible, the processor executing the instruction neither executes nor performs any other operations. Other memory access and write operations did not trigger any exceptions, and no other processor cores or cache coherence modules were present where the instruction was accessing the target memory. The effects of write operations on cache lines are globally visible.

The memory address accessed by the AM{SWAP/ADD}[_DB].{B/H} atomic memory access instruction is the value of the general-purpose register rj.

The memory access address of the AM{SWAP/ADD}[_DB].H atomic memory access instruction always requires natural alignment; otherwise, it will trigger an error. Non-aligned exceptions.

The AM{SWAP/ADD}_DB.{B/H} instruction, in addition to performing the aforementioned atomic operation sequence, also implements data barrier functionality. That is, when... Before such an atomic memory access instruction is allowed to execute, all memory access operations preceding that atomic memory access instruction in the same processor core must have been completed; Only after such atomic memory access instructions have been executed can all memory access operations following those instructions in the same processor core be allowed.

Xu was instructed to carry out the task.

In the AM{SWAP/ADD}[_DB].{B/H} instruction, if the register numbers of rd and rj are the same, then there are no exceptions to the triggered instruction.

In the instruction AM{SWAP/ADD}[_DB].{B/H}, if the register numbers of rd and rk are the same, the execution result is uncertain. Please avoid this in your software.

This is the current situation.

2.2.7.3AMCAS[_DB].{B/H/W/D}

Command format: amcas.b	rd, rk, rj	amcas_db.b	rd, rk, rj
amcas.h	rd, rk, rj	amcas_db.h	rd, rk, rj
amcas.w	rd, rk, rj	amcas_db.w	rd, rk, rj
amcas.d	rd, rk, rj	amcas_db.d	rd, rk, rj

LoongArch V1.1 Commands

The AMCAS[_DB].{B/H/W/D} instruction performs a byte/half-word/word/double-word Compare-and-Swap operation on a specified memory address.

The byte/half-word/word/double-word value retrieved from memory (old memory value) and the value stored at positions [7:0]/[15:0]/[31:0]/[63:0] in the general-purpose register rd (pre-value)

The values are compared, and only if the comparison results are equal will the values stored in positions [7:0]/[15:0]/[31:0]/[63:0] in the general-purpose register rk be updated (the new values).

Write the value to the same memory location. Regardless of whether the comparison results are equal, the old value in memory is sign-extended and written to the general-purpose register rd.

During the above operations, if a write operation occurs because the old value in memory is equal to the expected value, then the entire "read-modify-write" process is the original...

During the period from the return of data from a memory access read operation to the global visibility of the memory access write operation's execution effect, the processor executing the instruction neither has...

Executing other memory access and write operations did not trigger any exceptions, and no other processor cores or cache coherence modules were involved in the instruction's access to the object.

The effects of write operations on cache lines are globally visible.

The memory access address of the AMCAS[_DB].{H/W/D} instruction is the value of the general-purpose register rj, and the memory access address always requires natural alignment.

If this condition is met, the non-alignment exception will be triggered.

The AMCAS_DB.{B/H/W/D} instruction, in addition to performing the aforementioned atomic operation sequence, also implements data barrier functionality. That is, when this type of instruction...

Before an atomic memory access instruction is allowed to execute, all memory access operations preceding that instruction within the same processor core must have been completed; simultaneously only

Only after such atomic memory access instructions have been executed can all subsequent memory access operations within the same processor core be allowed to execute.

OK.

2.2.7.4LL.{W/D}, SC.{W/D}

Command format: ll.w	rd, rj, si14
ll.d	rd, rj, si14
sc.w	rd, rj, si14
sc.d	rd, rj, si14

The two pairs of instructions, LL.W and SC.W, and LL.D and SC.D, are used to implement the "read-modify-write" memory access sequence of atoms. LL.{W/D} refers to...

This instruction fetches a word/double word of data from a specified memory address, sign-extends it, and writes it to the general-purpose register rd. The corresponding SC.{W/D} instruction then operates accordingly.

Data of the same width accesses the same memory address. The mechanism for maintaining the atomicity of memory access sequence is that LL.{W/D} records the accesses during execution.

The address is set to a flag (LLbit is set to 1). When the SC.{W/D} instruction is executed, it checks LLbit, and a write operation is only actually performed when LLbit is 1.

Actions must be specified; otherwise, no action is required. When software needs to successfully complete an atomic "read-modify-write" memory access sequence, a loop needs to be constructed.

The LL-SC instruction pair is executed repeatedly until SC completes successfully. To construct this loop, the SC.{W/D} instruction will flag whether its execution was successful or not.

The value (which can also be simply understood as the LLbit value seen when the SC instruction is executed) is written into the general-purpose register rd and returned.

During the execution of a paired LL-SC, the following events will cause the LLbit to be cleared to 0:

ÿ The ERTN instruction was executed and the KLO bit in CSR.LLBCTL was not equal to 1 at the time of execution;

ÿ Other processor cores or the cache coherent master complete a store operation on the cache line containing the address corresponding to that LLbit.

operate.

The memory address accessed by the LL-SC instruction always requires natural alignment; if this condition is not met, an unaligned exception will be triggered.

If the storage access attribute of the LL-SC instruction is not consistently cacheable (CC), then the execution result is unpredictable.

It is important to note that when calculating the memory address, the above instruction requires shifting si14 two bits to the left before adding it to the base address.

vaddr = GR[rj] + SignExtend({si14, 2'b0}, GRLEN)

However, the immediate address offset values presented in the assembly representation of these instructions are still in bytes, that is, their value is si14<<2 in the instruction code.

2.2.7.5SC.Q

Command format: SC.Q rd, rk, rj

LoongArch V1.1 Commands

The SC.Q instruction is similar to the SC.D instruction and is used in conjunction with LL.D to implement an atomic "read-modify-write" memory access sequence for 128-bit data.

SC.Q writes the 128-bit data {GR[rk][63:0], GR[rd][63:0]} obtained by concatenating the general-purpose registers rk and rd into memory, and its memory access...

The address is the value of the general-purpose register rj. When the SC.Q instruction is executed, it checks the LLbit and only performs a write operation if the LLbit is 1; otherwise, it does not write.

The SC.Q instruction writes a flag indicating whether its execution was successful (which can be simply understood as the LLbit value seen during the execution of the SC.Q instruction) to the general-purpose key.

Returned in register rd.

The memory address accessed by the SC.Q instruction always requires 16-byte alignment; otherwise, an unaligned exception will be triggered.

If the storage access attribute of the accessed address is not consistently cacheable (CC), then the execution result is unpredictable.

2.2.7.6LL.ACQ.{W/D}, SC.REL.{W/D}

Command format: ll.acq.w rd, rj

ll.acq.d rd, rj

sc.rel.w rd, rj

sc.rel.d rd, rj

LoongArch V1.1 Commands

LL.ACQ.{W/D} is an LL.{W/D} instruction with added read-acquire semantics, meaning it will only acquire the data after LL.ACQ.{W/D} has finished executing (effective).

Once globally visible (i.e., after which all subsequent memory access operations can begin to execute, producing the effect of global visibility); SC.REL.{W/D} is added

The write-release semantics of the SC.{W/D} instruction mean that it will only wait until all memory access operations prior to SC.REL.{W/D} have completed (the effect is global).

Only after it is visible can SC.REL.{W/D} begin to execute (producing a globally visible effect).

The LL.ACQ.{W/D} instruction retrieves a word/double word of data from a specified memory address, sign-extends it, writes it to the general-purpose register rd, and simultaneously records...

The next access address is set and the previous flag is set (LLbit is set to 1). The SC.REL.{W/D} instruction conditionally moves the values in [31:0]/[63:0] of the general-purpose register rd.

A word/double word value is written to a specified memory address. Whether or not a memory is written depends on LLbit. A write operation is only performed when LLbit is 1; otherwise, no write is performed.

Regardless of whether memory is written to, the SC.REL instruction will display a flag indicating whether its execution was successful (which can also be simply understood as what the SC.REL instruction sees during execution).

The LLbit value is written to the general-purpose register rd and returned.

During the execution of a paired LL-SC, the following events will cause the LLbit to be cleared to 0:

ÿ The ERTN instruction was executed and the KLO bit in CSR.LLBCTL was not equal to 1 at the time of execution;

ÿ Other processor cores or the cache coherent master complete a store operation on the cache line containing the address corresponding to that LLbit.

operate.

The memory access addresses of the LL.ACQ and SC.REL instructions always require natural alignment; if this condition is not met, an unaligned exception will be triggered.

If the LL.ACQ and SC.REL instructions do not have a consistent cacheable (CC) memory access attribute for the accessed address, then the execution result will be...

uncertain.

2.2.8 Barrier Commands

2.2.8.1DBAR

Command format: dbar hint

The DBAR instruction is used to establish a barrier between load/store memory access operations. Its 15-bit immediate hint indicates the nature of the barrier.

Synchronization objects and synchronization levels.

A hint value of 0 is mandatory and indicates a fully functional synchronization barrier. This is only possible after all previous load/store memory accesses have been completed.

Only after the bottom layer has finished executing can the "DBAR 0" instruction begin execution; and only after "DBAR 0" has completed execution can all subsequent load/store accesses proceed.

The storage operation can only begin execution after the data is stored. For the DBAR pre-operation, "completely completed" means that the retrieved result is written to the destination register for the load operation.

In memory, a store operation means that the value written to it can be seen by all other observers in the system, which typically include the processor core and...

Other I/O master; DBAR post-operation "start execution" means that the action that can be seen by other observers has begun.

Apart from the mandatory 0-value hint, the specific meanings of the following non-zero-value hints are defined:

ÿ 0x700: Ensures that the globally visible execution effect of load operations at the same address before and after dbar 0x700 conforms to the order in the program.

ÿ 0x1~0x1f: Hint values within this range. Except for the reserved values 0xf and 0x1f, the meaning of each bit is defined as follows:

Bit0 being 1 indicates that all store operations following this dbar instruction are not bound by this barrier.	
Bit1 being 1 indicates that all load operations following this dbar instruction are not bound by this barrier.	
Bit2 being 1 indicates that all store operations prior to this dbar instruction will not affect the execution of this barrier.	
Bit3 being 1 indicates that all load operations prior to this dbar instruction will not affect the execution of this barrier.	
Bit 4 being 1 indicates that only cached load/store memory access operations before and after this dbar instruction are related to this barrier.	The execution effect of the memory access operation determined by the values of hint[1:0] following the dbar instruction is globally visible and must not be performed earlier than [the specified value]. The execution effect of all memory access operations determined by the values of hint[3:2] before the dbar instruction is globally visible. A value of 0 indicates that both cached and uncached load/store memory accesses before and after the dbar instruction are related to this barrier. At this point, the operation is only considered complete if all memory access operations defined by the values of hint[3:2] preceding the dbar instruction have been fully executed. After this is completed, the memory access operation determined by the value of hint[1:0] following the dbar instruction can begin to execute.

Undefined hint values are reserved and will be executed as if hint=0. Any non-reserved hints will be ignored if they are not executed according to the above definitions.

Now, the desired execution effect must be achieved with hint=0.

2.2.8.2IBAR

Command format: ibar hint

The IBAR instruction is used to synchronize the internal store and fetch operations of a single processor core. Its immediate hint is used to specify the instruction.

This indicates the synchronization target and the degree of synchronization for the barrier.

A hint value of 0 is required by default. It ensures that any instruction fetch following the "IBAR 0" instruction will be able to observe the "IBAR 0" instruction.

The results of all previous store operations.

2.2.9 **CRC** Check Command

2.2.9.1CRC[C].W.{B/H/W/D}.W

Command format: crc.wbw	rd, rj, rk
crc.whw	rd, rj, rk
crc.www	rd, rj, rk
crc.wdw	rd, rj, rk
crcc.wbw	rd, rj, rk
crcc.whw	rd, rj, rk
crcc.www	rd, rj, rk
crcc.w.d.w	rd, rj, rk

CRC[C].W.{B/H/W/D}.W is used for CRC-32 checksum calculation, which calculates the 32-bit accumulated CRC checksum stored in the general-purpose register rk.

The checksum and the message stored in bits [7:0]/[15:0]/[31:0]/[63:0] in the general-purpose register rj are used to obtain a new 32-bit checksum according to the CRC-32 checksum generation algorithm.

The CRC checksum, after sign extension, is written to the general-purpose register rd. The difference lies in that CRC.W.{B/H/W/D}.W uses multiple terms from IEEE 802.3.

The formula (polynomial value 0xEDB88320), CRCC.W.{B/H/W/D}.W uses Castagnoli polynomial (polynomial value 0x82F63B78).

The CRC instructions defined in this manual only support the "LSB-first" (little-endian) standard, which means transmitting the least significant bit (little-endian) of the data first.

The least significant bit of the data is mapped to the coefficient of the most significant term of the message polynomial.

```
CRC.WBW:
    chksum = CRC32(GR[rk][31:0], GR[rj][7:0], 8, 0xEDB88320)
    GR[rd] = SignExtend(chksum, GRLEN)
CRC.WHW:
    chksum = CRC32(GR[rk][31:0], GR[rj][15:0], 16, 0xEDB88320)
    GR[rd] = SignExtend(chksum, GRLEN)
CRC.WWW:
    chksum = CRC32(GR[rk][31:0], GR[rj][31:0], 32, 0xEDB88320)
    GR[rd] = SignExtend(chksum, GRLEN)
```

CRC.WDW:

```
chksum = CRC32(GR[rk][31:0], GR[rj][63:0], 64, 0xEDB88320)
GR[rd] = SignExtend(chksum, GRLEN)
```

CRCC.WBW:

```
chksum = CRC32(GR[rk][31:0], GR[rj][7:0], 8, 0x82F63B78)
GR[rd] = SignExtend(chksum, GRLEN)
```

CRCC.WHW:

```
chksum = CRC32(GR[rk][31:0], GR[rj][15:0], 16, 0x82F63B78)
GR[rd] = SignExtend(chksum, GRLEN)
```

CRCC.WWW:

```
chksum = CRC32(GR[rk][31:0], GR[rj][31:0], 32, 0x82F63B78)
GR[rd] = SignExtend(chksum, GRLEN)
```

CRCC.W.D.W:

```
chksum = CRC32(GR[rk][31:0], GR[rj][63:0], 64, 0x82F63B78)
GR[rd] = SignExtend(chksum, GRLEN)
```

2.2.10 Other Miscellaneous Instructions

2.2.10.1 SYSCALL

Command format: syscall code

Executing the SYSCALL instruction will immediately and unconditionally trigger a system call exception.

The information carried in the code field of the instruction code can be used by exception handling routines as parameters passed to them.

2.2.10.2 BREAK

Command format: break code

Executing the BREAK instruction will immediately and unconditionally trigger a breakpoint exception.

The information carried in the code field of the instruction code can be used by exception handling routines as parameters passed to them.

2.2.10.3 ASRT{LE/GT}.D

Command format: asrtle.d rj, rk
asrtgt.d rj, rk

The values in general-purpose register rj and general-purpose register rk are compared as signed numbers. If the comparison condition is not met, an address boundary event is triggered. Check for exceptions. For the ASRTLE.D instruction, an exception is triggered if the value in general-purpose register rj is greater than the value in general-purpose register rk; for The ASRTGT.D instruction triggers an exception if the value in general-purpose register rj is less than or equal to the value in general-purpose register rk.

2.2.10.4 RDTIME{L/H}.W, RDTIME.D

Command format: rdtimel.w rd, rj rdtime.d rd, rj
rdtimeh.w rd, rj

The Loongson instruction set defines a constant-frequency timer, the main body of which is a 64-bit counter called the Stable Counter. The Counter is set to 0 after reset, and then increments by 1 every counting clock cycle. When it reaches all 1s, it automatically wraps back to 0 and continues incrementing. Meanwhile, each... Each timer has a globally unique, software-configurable identifier called a Counter ID. A constant-frequency timer is characterized by its timing frequency changing over time. The bit remains unchanged regardless of how the processor core's clock frequency changes.

The RDTIME{L/H}.W and RDTIME.D instructions are used to read information from a constant frequency timer and write the Stable Counter value to a general-purpose register. In the `rd` instruction, the Counter ID information is written to the general-purpose register `rj`. The difference between the three instructions lies in the different Stable Counter information they read: `RDTIMEL.W` reads bits [31:0] of the Counter, `RDTIMEH.W` reads bits [63:32] of the Counter, and `RDTIME.D` reads the entire Counter. A 64-bit Counter value. On a 64-bit processor, the RDTIME{L/H}.W instruction reads a 32-bit value, sign-extended, and writes it to a general-purpose register. The RDTIME{L/H}.W instruction is defined to allow access to the 64-bit Counter on a 32-bit processor.

2.2.10.5 CPUCFG

Command format: cpucfg rd, rj

The CPUCFG instruction is used by software to dynamically identify which features of the Dragon architecture are implemented in the running processor during execution. The implementation details of these instruction set features are recorded in a series of configuration information words. Each execution of the CPUCFG instruction can read one configuration word. The character ÿ (xÿ).

When using the CPUCFG instruction, the source operand register rj stores the number of the configuration information word to be accessed. After the instruction is executed, the configuration information word read is... The configuration information word is written into the general-purpose register rd. Each configuration information word is 32 bits, and under the LA64 architecture, it is sign-extended before being written to the end. Result register.

The configuration information word contains a series of configuration bits (fields), recorded in the format CPUCFG.<configuration word size>.<configuration information mnemonic name>[bits]

[Subscript], where the bit subscript for a single-bit configuration bit is bitXX, representing the XXth bit of the configuration word; the bit subscript for a multi-bit configuration field is...

bitXX:YY represents the consecutive (XX-YY+1) bits from bit XX to bit YY of the configuration word. For example, bit 0 in configuration word 1 is used to represent...

Whether the LA32 architecture is implemented is recorded as CPUCFG.1.LA32[bit0], where 1 indicates that the word size of the configuration information word is 1.

LA32 indicates that the mnemonic name for this configuration information field is LA32, and bit 0 indicates that the LA32 field is located at bit 0 of the configuration word. (Configuration field #1)

The PALEN field, which records the number of physical address bits supported by bits 11 to 4 in the set word, is denoted as CPUCFG.1.PALEN[bit11:4].

The configuration information accessible by the CPUCFG instruction in the Dragon architecture is listed in Table 2-2. Accessing an undefined configuration word using CPUCFG will read back all zeros.

Value. Undefined fields within a defined configuration word can be read back with arbitrary values when CPUCFG is executed; the software should not interpret them.

Table 2-2 **CPUCFG** Access Configuration Information List

Font size subscript mnemonic name			meaning
0x0	31:0	ADJ	Processor Identifier
0x1	1:0	ARCH	2'b00 indicates the implementation of the LA32 simplified architecture; 2'b01 indicates the implementation of the LA32 architecture. 2'b10 indicates the implementation of the LA64 architecture. 2'b11 is reserved.
	2	PGMMU	A value of 1 indicates that the MMU supports page mapping mode.
	3	IOCSR	A value of 1 indicates support for the IOCSR instruction.
	11:4	POLES	The number of supported physical address bits, PALEN value minus 1.
	19:12	The number of virtual address bits supported by VALEN (VALEN value minus 1)	
	20	UAL	A value of 1 indicates support for unaligned memory access.
	21	RI	A value of 1 indicates support for the "read forbidden" page attribute.
	22	EP	A value of 1 indicates support for the "Execution Protection" page attribute.
	23	RPLV	A value of 1 indicates support for RPLV page attributes.
	24	HP	A value of 1 indicates support for huge page attributes.
	25	CRC	A value of 1 indicates support for CRC check commands.
	26	MSG_INT being 1 indicates support for external interrupts via message interrupt mode.	
0x2	0	FP	A value of 1 indicates support for basic floating-point instructions.
	1	FP_SP	A value of 1 indicates support for single-precision floating-point numbers.
	2	FP_DP	A value of 1 indicates support for double-precision floating-point numbers.
	5:3	FP_ver	The version number of the floating-point arithmetic standard. 1 is the initial version number, indicating compatibility with the IEEE 754-2008 standard.
	6	LSX	A value of 1 indicates support for 128-bit vector extension.
	7	LASX	A value of 1 indicates support for 256-bit vector extension.
	8	COMPLEX set to 1 indicates support for complex vector operation instructions.	
	9	CRYPTO	A value of 1 indicates support for encryption/decryption vector commands.
	10	LVZ	A value of 1 indicates support for virtualization extensions.
	13:11	LVZ_ver is the version number of the virtualization hardware acceleration specification. 1 is the initial version number.	
	14	LLFTP	A value of 1 indicates support for constant frequency timers and timers.
	17:15	LLFTP_ver is the version number of the constant frequency timer and the timer itself. 1 indicates the initial version.	
	18	A value of 1 for LBT_X86 indicates support for the x86 binary translation extension.	

Font size subscript mnemonic name			meaning
	19	A value of 1 for LBT_ARM indicates support for the ARM binary translation extension.	
	20	A value of 1 for LBT_MIPS indicates support for the MIPS binary translation extension.	
	21	LSPW	A value of 1 indicates support for software page table traversal instructions.
	22	LAM	A value of 1 indicates support for AM* atomic memory access instructions.
	24	HPTW	A value of 1 indicates support for hardware page table walker.
	25	FRECIPE	A value of 1 indicates support for the four instructions FRECIPE.{S/D}, FRSQRTE.{S/D}, and if all of them are supported... The 128-bit vector extension supports the four instructions VFRECIPE.{S/D}, VFRSQRTE.{S/D}, and so on. If 256-bit vector extension is also supported, then XVFRECIPE.{S/D} and XVFRSQRTE.{S/D} are supported. These 4 instructions.
	26	DIV32	A value of 1 indicates that on a 64-bit machine, the DIV.W[U] and MOD.W[U] instructions rely solely on the lower 32 bits of the input register. Data calculation
	27	LAM_BH	A value of 1 indicates support for the 8 instructions AM{SWAP/ADD}[_DB].{B/H}.
	28	LAMCAS	A value of 1 indicates support for the 8 instructions AMCAS[_DB].{B/H/W/D}.
	29	A value of 1 for LLACQ_SCREL indicates support for the four commands: LLACQ.{W/D}, SCREL.{W/D}, etc.	
	30	SCQ	A value of 1 indicates support for the SC.Q command.
0x3	0	CCDMA	A value of 1 indicates support for hardware cache coherent DMA.
	1	SFB	A value of 1 indicates support for Store Fill Buffer (SFB).
	2	UCACC	A value of 1 indicates support for ucacc Win
	3	LLEXC	A value of 1 indicates support for the LL instruction to fetch exclusive blocks.
	4	SCDLY	A value of 1 indicates support for the random delay function after SC.
	5	LLDBAR	A value of 1 indicates support for LL automatic bar display functionality.
	6	ITLBHMC	A value of 1 indicates that the hardware maintains consistency between the ITLB and TLB.
	7	ICHMC	A value of 1 indicates that the hardware maintains data consistency between the ICache and DCache within the same processor core.
	10:8	SPW_LVL	The maximum number of directory levels supported by the page walk command.
	11	A value of 1 for SPW_HP_HF indicates that the page walk instruction will halve the page and fill it into the TLB when it encounters a large page.	
	12	RVA	A value of 1 indicates support for the software configuration feature to shorten the virtual address range.
	16:13	RVAMAX-1 is the maximum configurable virtual address shortening bit width - 1.	
	17	A value of 1 for DBAR_hints indicates that the non-zero value of DBAR is implemented according to the manual's recommended meaning.	
	23	A value of 1 for LD_SEQ_SA indicates that the hardware has enabled the function of ensuring sequential execution of load operations at the same address.	
0x4	31:0	CC_FREQ is the crystal frequency corresponding to the constant frequency timer and the clock used by the timer.	
0x5	15:0	CC_MUL is the frequency multiplication factor for a constant frequency timer and the clock used by the timer.	
	31:16	CC_DIV: Constant frequency timer and the frequency division factor corresponding to the clock used by the timer.	
0x6	0	PMP	A value of 1 indicates support for the performance counter.
	3:1	PMVER	In the performance monitor, the architecture defines the version number of the event, with 1 being the initial version.
	7:4	PMNUM	Number of performance monitors - 1
	13:8	PMBITS	Performance monitoring counter bit width -1
	14	UPM	A value of 1 indicates support for user-mode read performance counters.
0x10	0	A L1 IU_Present value of 1 indicates the presence of a Level 1 instruction cache or a Level 1 unified cache.	

Font size subscript mnemonic name			meaning
	1	A value of 1 for L1 IU Unify indicates that the cache shown by L1 IU_Present is a unified cache.	
	2	A L1 D Present value of 1 indicates the existence of a Level 1 data cache.	
	3	A L2 IU Present value of 1 indicates the presence of a Level 2 instruction cache or a Level 2 unified cache.	
	4	A value of 1 for L2 IU Unify indicates that the cache shown by L2 IU_Present is a unified cache.	
	5	A value of 1 for L2 IU Private indicates that the cache shown by L2 IU_Present is private to each core.	
	6	A value of 1 for L2 IU Inclusive indicates that the cache shown by L2 IU_Present is inclusive of the lower-level (L1) cache.	
	7	A L2 D Present value of 1 indicates the existence of a second-level data cache.	
	8	A L2 D Private value of 1 indicates that the L2 data cache is private to each core.	
	9	A value of 1 for L2 D Inclusive indicates that the second-level data cache has an inclusion relationship with the lower level (L1).	
	10	A L3 IU Present value of 1 indicates the presence of a Level 3 instruction cache or a Level 3 unified cache.	
	11	A value of 1 for L3 IU Unify indicates that the cache shown by L3 IU_Present is a unified cache.	
	12	A value of 1 for L3 IU Private indicates that the cache shown by L3 IU_Present is private to each core.	
	13	A value of 1 for L3 IU Inclusive indicates that the cache represented by L3 IU_Present is inclusive of lower levels (L1 and L2).	
	14	A L3 D Present value of 1 indicates the existence of a three-level data cache.	
	15	A L3 D Private value of 1 indicates that the Level 3 data cache is private to each core.	
	16	A value of 1 for L3 D Inclusive indicates that the Level 3 data cache has an inclusion relationship with lower levels (L1 and L2).	
0x11	15:0	Way-1	Number of paths -1 (The cache corresponding to L1 IU_Present in configuration word 10)
	23:16	Index-log2	log2(number of cache lines per path) (Cache corresponding to L1 IU_Present in configuration word 10)
	30:24	Linesize-log2	log2(Cache line bytes) (The cache corresponding to L1 IU_Present in configuration word 10)
0x12	15:0	Way-1	Number of paths -1 (The cache corresponding to L1 D Present in configuration word 10)
	23:16	Index-log2	log2 (number of cache lines per path) (Cache corresponding to L1 D Present in configuration word 10)
	30:24	Linesize-log2	log2(Cache line bytes) (The cache corresponding to L1 D Present in configuration word 10)
0x13	15:0	Way-1	Number of paths -1 (The cache corresponding to L2 IU Present in configuration word 10)
	23:16	Index-log2	log2 (number of cache lines per path) (Cache corresponding to L2 IU Present in configuration word 10)
	30:24	Linesize-log2	log2(Cache line bytes) (The cache corresponding to L2 IU Present in configuration word 10)
0x14	15:0	Way-1	Number of paths -1 (The cache corresponding to L3 IU Present in configuration word 10)
	23:16	Index-log2	log2 (number of cache lines per path) (Cache corresponding to L3 IU Present in configuration word 10)
	30:24	Linesize-log2	log2(Cache line bytes) (The cache corresponding to L3 IU Present in configuration word 10)

3 Basic Floating-Point Instructions

This chapter introduces the floating-point instructions in the non-privileged subset of the Dragon architecture's foundation. The functional definitions of the basic floating-point instructions in the Dragon architecture follow...

IEEE 754-2008 standard.

Basic floating-point instructions cannot be implemented independently of basic integer instructions. Generally, we recommend implementing both basic integer instructions and basic floating-point instructions.

Floating-point instructions. However, for some cost-sensitive embedded applications with extremely low floating-point processing performance requirements, the architectural specifications also allow for...

Implement basic floating-point instructions, or only implement the instructions within the basic floating-point instruction set that operate on single-precision floating-point numbers and word integers (see Table 3-1).

Whether the basic floating-point instructions include instructions for operating on double-precision floating-point numbers and double-word integers is independent of whether the architecture is LA32 or LA64. However...

MOVGR2FR.DÿMOVFR2GR.DÿFSCALEB.S/DÿFLOGB.S/DÿFRINT.S/DÿFRECIPE.S/DÿFRSQRTE.S/Dÿ

The 20 instructions FLD{GT/LE}.{S/D} and FST{GT/LE}.{S/D} only need to be implemented under the LA64 architecture.

Table 3-1 Basic Floating-Point Instructions for Single-Precision Floating-Point Numbers and Word Integers

Floating-point arithmetic instructions	FADD.S, FSUB.S, FMUL.S, FDIV.S, FMADD.S, FMSUB.S, FNMADD.S, FNMSUB.S, FMAX.S, FMIN.S, FMAXA.S, FMINA.S, FABS.S, FNEG.S, FSQRT.S, FRECIP.S, FRSQRT.S, FCOPYSIGN.S, FCLASS.S
Floating-point comparison instructions	FCMP.cond.S
Floating-point conversion instructions	FFINT.SW, FTINT.WS, FTINTRM.WS, FTINTRP.WS, FTINTRZ.WS, FTINTRNE.WS,
Floating-point transfer instructions	FMOV.S, FSEL, MOVGR2FR.W, MOVFR2GR.S, MOVGR2FCSR, MOVFCSR2GR, MOVFR2CF, MOVCF2FR, MOVGR2CF, MOVCF2GR
Floating-point branch instructions	BCEQZ, BCNEZ
Floating-point ordinary memory access instructions	FLD.S, FST.S, FLDX.S, FSTX.S

3.1 Basic Floating-Point Instruction Programming Model

The basic floating-point instruction programming model described in this section only covers the aspects that application software developers need to focus on. Software personnel using...

Programming with basic floating-point instructions builds upon the basic integer instruction programming model and further delves into the content discussed in this section.

3.1.1 Floating-point data type

Floating-point data types include single-precision floating-point numbers and double-precision floating-point numbers, both of which conform to the definitions in the IEEE 754-2008 standard specification.

3.1.1.1 Single-precision floating-point numbers

Single-precision floating-point numbers are 32 bits wide and are organized in the following format:

31	30 23	22 0
S	Exponent	Fraction

The floating-point values represented by the different values of the S, Exponent, and Fraction fields are shown in Table 3-2:

Table 3-2 Methods for Calculating Single-Precision Floating-Point Numbers

Exponent	Fraction	S	bit[22]	In
0	=0	0	0	+0
		1	0	-0
0	!=0	0	Any value	The denormalized number has a value of +2 -126 × (0.Fraction).
		1.	Any value	The denormalized number has a value of -2 -126 × (0.Fraction).
[1, 0xFE]	any value	0	Any value	Normalization number, with a value of +2 (Exponent - 127) × (1.Fraction)
		1.	Any value	Normalization number, with a value of -2 (Exponent-127)×(1.Fraction)
0xFF	=0	0	0	Positive infinity (+ÿ)
		1	0	Negative infinity (-ÿ)
0xFF	!=0	Any value 0		Signaling Not a Number (SNaN)
		any value 1		Quiet Not a Number (QNaN)

For the specific meanings of ±ÿ, SNaN, and QNaN, please refer to the IEEE 754-2008 standard specification.

3.1.1.2 Double-precision floating-point numbers

Double-precision floating-point numbers are 64 bits wide and are organized in the following format:

63	62 52	51 0
S	Exponent	Fraction

The floating-point values represented by the different values of the S, Exponent, and Fraction fields are shown in Table 3-3:

Table 3-3 Methods for Calculating Double-Precision Floating-Point Numbers

Exponent	Fraction	S	bit[51]	In
0	=0	0	0	+0
		1	0	-0
0	!=0	0	Any value	The denormalized number has a value of +2 -1022 × (0.Fraction).
		1.	Any value	The denormalized number has a value of -2 -1022 × (0.Fraction).
[1, 0x7FE]	Any value	0	Any value	Normalization number, with a value of +2 (Exponent - 1023) × (1.Fraction)
		1.	Any value	Normalization number, with a value of -2 (Exponent - 1023) × (1.Fraction)
0x7FF	=0	0	0	Positive infinity (+ÿ)
		1	0	Negative infinity (-ÿ)
0x7FF	!=0	Any value 0		Signaling Not a Number (SNaN)
		any value 1		Quiet Not a Number (QNaN)

For the specific meanings of ±ÿ, SNaN, and QNaN, please refer to the IEEE 754-2008 standard specification.

3.1.1.3 The NOT result produced by the instruction

The non-number result of 1 generated by floating-point instructions either comes from NaN propagation or is generated directly. The situation requiring NaN propagation is...

There are two situations.

Case 1: When an instruction generates an Invalid Operation floating-point exception due to a source operand containing SNaN, but the Invalid Operation floating-point exception...

[1] At this point, the only non-number can be QNaN.

If point exceptions are not enabled, a QNaN result will be generated. The value of this QNaN is the highest priority SNaN among the source operands.

It is then propagated to the corresponding NaN.

The priority rule for source operands is: if there are two source operands fj and fk, then fj has higher priority than fk; if there are three...

If the source operands are fa, fj, and fk, then fa has higher precedence than fj, and fj has higher precedence than fk.

The rules for generating SNaN as QNaN are as follows:

ÿ If the result is the same width as the source operand, then the highest bit of the SNaN mantissa will be set to 1, while the remaining bits will remain unchanged.

ÿ If the result is narrower than the source operand, then retain the high-order bits of the mantissa, discard the low-order bits that exceed the range, and finally set the highest bit of the mantissa to 1.

ÿ If the result is wider than the source operand, then pad the least significant bit of the mantissa with 0s and finally set the most significant bit of the mantissa to 1.

Case 2: If the source operand does not contain SNaN but does contain QNaN, the QNaN with the highest priority is selected as the result of this instruction.

In this case, the method for determining the priority of the source operand is the same as in case one above.

Except for the two cases mentioned above, all other cases requiring a QNaN result will directly set the default QNaN value. The default single...

The value of precision QNaN is 0x7FC00000, and the default value of double precision QNaN is 0x7FF8000000000000.

3.1.2 Fixed-point data types

Some floating-point instructions (such as floating-point conversion instructions) also operate on fixed-point data, including words (abbreviated as W, length 32 bits) and long words (...).

(Abbreviated as L, length 64b).

Both single-character and long-character data types use binary two's complement encoding.

3.1.3 Registers

Floating-point instruction programming involves registers such as the floating-point register (FR) and the condition flag register.

(Condition Flag Register, abbreviated as CFR) and Floating-point Control and Status Register, abbreviated as

FCSRÿÿ

3.1.3.1 Floating-point registers

There are 32 FRs, denoted as f0 to f31, each of which can be read and written. This applies only when implementing floating-point instructions that operate on single-precision floating-point numbers and word integers.

At that time, the bit width of FR is 32 bits. Normally, the bit width of FR is 64 bits, regardless of whether it's LA32 or LA64 architecture. Basic floating-point number.

The instructions and floating-point registers are orthogonal, meaning that from an architectural perspective, any floating-point register operand in these instructions can use 32-bit memory.

Any one of the FRs.

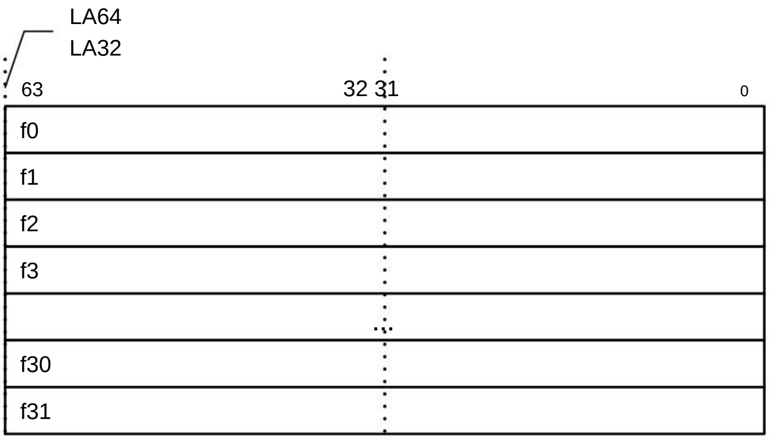

Figure 3-1 Floating-point register

When the floating-point register records a single-precision floating-point number or a word integer, the data always appears in bits [31:0] of the floating-point register.

Bits [63:32] of the floating-point register can have any value.

3.1.3.2 Condition Flag Register

There are 8 CFRs, denoted as fcc0 to fcc7, each of which can be read and written. Each CFR has a bit width of 1 bit. The result of a floating-point comparison will be written to...

In the condition flag register, the flag is set to 1 if the comparison result is true, and set to 0 otherwise. The condition for floating-point branch instructions is determined by the condition flag register.

3.1.3.3 Floating-point control status register

There are four FCSRs, denoted as fcsr0 to fcsr3, each with a bit width of 32 bits. fcsr1 to fcsr3 are aliases for certain fields within fcsr0, i.e., access...

Querying fcsr1~fcsr3 actually involves accessing certain fields of fcsr0. When the software writes to fcsr1~fcsr3, the corresponding fields in fcsr0 are modified while the remaining bits are preserved.

The definitions of the various fields of fcsr0 are unchanged.

Table 3-4 **FCSR0** Register Field Definitions

Bit	Name reading and writing		describe
4:0	Enables	RW	Each floating-point operation VZOUI exception has an enable bit that allows the exception to be triggered. Bit 4 corresponds to V, bit 3 corresponds to Z, bit 2 corresponds to O, bit 1 corresponds to U, and bit 0 corresponds to I.
7:5	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

9:8	RM	RW	Rounding mode control. It includes 4 valid values, each with the following meaning:
			0: RNE, corresponding to Round to Ward Nearest, ties to Even in IEEE 754-2008;
			1: RZ, corresponding to Roundtoward Zero in IEEE 754-2008;
			2: RP, corresponding to RoundtowardsPositive in IEEE 754-2008;
			3: RM, corresponding to RoundtowardsNegative in IEEE 754-2008.

15:10	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	
20:16	Flags	RW	This is the cumulative number of VZOUI exceptions for various floating-point operations that have occurred but not yet trapped since the Flags field was cleared by the software. Bit 20 corresponds to V, bit 19 corresponds to Z, bit 18 corresponds to O, bit 17 corresponds to U, and bit 16 corresponds to I.
23:21	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	
28:24	Cause	RW	VZOUI exceptions resulting from the most recent floating-point operation. Bit 28 corresponds to V, bit 27 corresponds to Z, bit 26 corresponds to O, bit 25 corresponds to U, and bit 24 corresponds to I.

Bit	Name	reading and writing	describe
31:29	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

FCSR1 is an alias for the Enables field in FCSR0. Its location is the same as in FCSR0.

FCSR2 is an alias for the Cause and Flags fields in FCSR0. The positions of the fields are consistent with those in FCSR0.

FCSR3 is an alias for the RM field in FCSR0. Its location is the same as in FCSR0.

3.1.4 Floating-point exception

Floating-point exceptions refer to situations where the floating-point processing unit cannot process operands or the results of floating-point calculations in the usual way, and the floating-point function unit... There will be corresponding exceptions.

The basic floating-point instructions support five floating-point exceptions defined in IEEE 754-2008:

ÿ Inexact (I)

ÿ ÿÿ Underflow (U)

ÿ Overflow (O)

ÿ Division by Zero (Z)

ÿ Invalid Operation (V)

Each bit in the Cause field of FCSR0 corresponds to one of the aforementioned exceptions. After each floating-point instruction is executed, the exception details are updated. The new Cause field is in FCSR0.

FCSR0 also includes an enable bit (Enables field) for each floating-point exception. The enable bit determines the exception generated by the floating-point processing unit. The external flag will either trigger an exception trap or set a status flag. When a floating-point exception occurs, if its corresponding Enable bit is 1, then... This will trigger a floating-point exception trap; if the corresponding Enable bit is 0, then a floating-point exception trap will not be triggered, and FCSR0 will be set instead. The corresponding position 1 in the Flag field.

A single floating-point instruction can generate multiple floating-point exceptions during execution.

When a floating-point exception occurs during the execution of a floating-point instruction but does not trigger a floating-point exception trap, the floating-point processing unit will generate a... Default results. Different exceptions generate default results in different ways; Table 3-5 lists the specific generation rules. Table 3-5 Default results for floating-point exceptions.

Field	description	rounding mode	Default result
I	The result of rounding in any non-precise mode or the result after overflow.		
IN	overflow	RNE	The rounded result could be 0, subnormal, or the normal number with the smallest absolute value (single-precision rounding). Degrees: ±2 -126, Double precision: ±2 -1022)
		RZ	The result after rounding may be 0, or subnormal.
		RP	The rounded result could be 0, subnormal, or the smallest positive normal number (single precision). +2 -126, double precision: +2 -1022)
		RM	The rounded result could be 0, subnormal, or the largest negative normal number (single precision). -2 -126, double precision: -2 -1022)

[1] In fact, only the four exceptions besides underflow strictly conform to this description. Please see the detailed description below for the definition of underflow exceptions.

Field description rounding mode			Default result
THE	overflow	RNE sets the result to +ÿ or -ÿ based on the sign of the intermediate result.	
		RZ	Set the result to the maximum number based on the sign of the intermediate results.
		RP	Correct negative overflow to the smallest negative number, and correct positive overflow to +ÿ.
		RM	Correct positive overflow to the largest positive number, and correct negative overflow to -ÿ.
WITH	Dividing any pattern by zero yields a corresponding signed infinity.		
V. Illegal operation in any mode provides a QNaN.			

3.1.4.1 Illegal Operation Exception (V)

An invalid operation exception signal is issued only if there is no validly defined result. If no exception trap is triggered, then...

Generate a QNaN. For specific details on the determination of illegal operation exceptions, please refer to Section 7.2 of the IEEE 754-2008 specification.

If an exception is allowed to trap: the result register is not modified, and the source register is preserved.

If an exception prevents trapping: If no other exception occurs, QNaN is written to the destination register.

3.1.4.2 Division by zero exception (Z)

In division operations, when the divisor is 0 and the dividend is a finite non-zero number, a signal is issued to indicate division by zero.

If an exception is allowed to trap: the result register is not modified, and the source register is preserved.

If an exception is prohibited from trapping: if no trap occurs, the result is a signed infinity.

3.1.4.3 Overflow Exception (O)

Treating the exponent field as unbounded and rounding intermediate results, when the absolute value of the obtained result exceeds the maximum finite number of the target format,

An overflow exception is signaled. (This exception also sets up both inaccuracy exceptions and a flag.)

If an exception is allowed to trap: the result register is not modified, and the source register is preserved.

If an exception is prohibited from trapping: if no trap occurs, the final result is determined by the rounding mode and the sign of the intermediate result.

3.1.4.4 Underflow Exception (U)

An underflow exception occurs when a non-zero tiny value is detected. The method for detecting non-zero tiny values is to check after rounding.

Test.

Rounding check: For a non-zero result, if the exponent field is considered unbounded, the intermediate result is rounded.

The result is in (-2 Emin) . In the case of Emin , this result is considered a non-zero infinitesimal value. (Single-precision number Emin = -126, double-precision number Emin = ...)

-1022ÿÿ

When FCSR.Enable.U=0, if the detected result is a non-zero tiny value:

(1) If the final result of the floating-point operation is not precise, then both U and I in FCSR.Cause should be set to 1;

(2) If the final rounding result of the floating-point operation is accurate, then neither U nor I in FCSR.Cause should be set to 1.

When FCSR.Enable.U=1, if the detected result is a non-zero tiny value, regardless of whether the final rounded result of the floating-point operation is accurate...

Whether it's exact or inaccurate, it will trigger a floating-point exception trap.

3.1.4.5 Exceptions to Inaccuracy (I)

The FPU generates an inaccuracy exception when the following conditions occur:

ÿ Rounding results are not precise.

ÿ The rounding result overflows, and the enable bit for the overflow exception is not set.

If exception traps are allowed: If an imprecise exception trap is enabled, the result register is not modified and the source register is preserved.

Because this execution mode affects performance, inaccurate exception traps are only enabled when necessary.

If an exception is prevented from trapping: If no other software traps occur, the rounded or overflow result is sent to the destination register.

3.2 Overview of Basic Floating-Point Instructions

3.2.1 Floating-point arithmetic instructions

3.2.1.1F{ADD/SUB/MUL/DIV}.{S/D}

Command format: fadd.s	fd, fj, fk	fadd.d	fd, fj, fk
fsub.s	fd, fj, fk	fsub.d	fd, fj, fk
fmul.s	fd, fj, fk	fmul.d	fd, fj, fk
fdiv.s	fd, fj, fk	fdiv.d	fd, fj, fk

The FADD.{S/D} instruction adds the single-precision/double-precision floating-point number in floating-point register fj to the single-precision/double-precision floating-point number in floating-point register fk.

The single-precision/double-precision floating-point result is written to the floating-point register fd. Floating-point addition operations follow the IEEE 754-2008 standard.

Specifications for the addition(x,y) operation.

FADD.S:

```
FR[fd][31:0] = FP32_addition(FR[fj][31:0], FR[fk][31:0])
```

FADD.D:

```
FR[fd] = FP64_addition(FR[fj], FR[fk])
```

The FSUB.{S/D} instruction subtracts the single-precision/double-precision floating-point number in floating-point register fk from the single-precision/double-precision floating-point number in floating-point register fj.

The single-precision/double-precision floating-point result is written to the floating-point register fd. Floating-point subtraction operations follow the IEEE 754-2008 standard.

Specifications for the subtraction(x,y) operation.

FSUB.S:

```
FR[fd][31:0] = FP32_subtraction(FR[fj][31:0], FR[fk][31:0])
```

FSUB.D:

```
FR[fd] = FP64_subtraction(FR[fj], FR[fk])
```

The FMUL.{S/D} instruction multiplies the single-precision/double-precision floating-point number in floating-point register fj by the single-precision/double-precision floating-point number in floating-point register fk.

The resulting single-precision/double-precision floating-point number is written to the floating-point register fd. Floating-point multiplication operations follow the IEEE 754-2008 standard.

Specifications for the multiplication(x,y) operation.

FMUL.S:

```
FR[fd][31:0] = FP32_multiplication(FR[fj][31:0], FR[fk][31:0])
```

FMUL.D:

```
FR[fd] = FP64_multiplication(FR[fj], FR[fk])
```

The FDIV.{S/D} instruction divides the single-precision/double-precision floating-point number in floating-point register fj by the single-precision/double-precision floating-point number in floating-point register fk.

The single-precision/double-precision floating-point result is written to the floating-point register fd. Floating-point division operations follow the IEEE 754-2008 standard.

Specifications for the division(x,y) operation.

FDIV.S:

FR[fd][31:0] = FP32_division(FR[fj][31:0], FR[fk][31:0])

FDIV.D:

FR[fd] = FP64_division(FR[fj], FR[fk])

3.2.1.2F{MADD/MSUB/NMADD/NMSUB}.{S/D}

Command format: fmadd.s	fd, fj, fk, fa	fmadd.d	fd, fj, fk, fa
fmsub.s	fd, fj, fk, fa	fmsub.d	fd, fj, fk, fa
fnmadd.s	fd, fj, fk, fa	fnmadd.d	fd, fj, fk, fa
fnmsub.s	fd, fj, fk, fa	fnmsub.d	fd, fj, fk, fa

The FMADD.{S/D} instruction modifies the single-precision/double-precision floating-point number in floating-point register fj and the single-precision/double-precision floating-point number in floating-point register fk.

The numbers are multiplied, the result is added to the single-precision/double-precision floating-point number in the floating-point register fa, and the resulting single-precision/double-precision floating-point number is written to [the register name].

In the floating-point register fd.

FMADD.S:

FR[fd][31:0] = FP32_fusedMultiplyAdd(FR[fj][31:0], FR[fk][31:0], FR[fa][31:
0])

FMADD.D:

FR[fd] = FP64_fusedMultiplyAdd(FR[fj], FR[fk], FR[fa])

The FMSUB.{S/D} instruction modifies the single-precision/double-precision floating-point number in floating-point register fj and the single-precision/double-precision floating-point number in floating-point register fk.

Multiply the numbers, subtract the single-precision/double-precision floating-point number in the floating-point register fa from the result, and write the resulting single-precision/double-precision floating-point number to the...

In the floating-point register fd.

FMSUB.S:

FR[fd][31:0] = FP32_fusedMultiplyAdd(FR[fj][31:0], FR[fk][31:0], -FR[fa][3
1:0])

FMSUB.D:

FR[fd] = FP64_fusedMultiplyAdd(FR[fj], FR[fk], -FR[fa])

The FNMADD.{S/D} instruction modifies the single-precision/double-precision floating-point number in floating-point register fj and the single-precision/double-precision floating-point number in floating-point register fk.

Multiply the points, add the result to the single-precision/double-precision floating-point number in the floating-point register fa, and then negative the resulting single-precision/double-precision floating-point number.

Then it is written to the floating-point register fd.

FNMADD.S:

FR[fd][31:0] = -FP32_fusedMultiplyAdd(FR[fj][31:0], FR[fk][31:0], FR[fa][3
1:0])

FNMADD.D:

FR[fd] = -FP64_fusedMultiplyAdd(FR[fj], FR[fk], FR[fa])

The FNMSUB.{S/D} instruction modifies the single-precision/double-precision floating-point number in floating-point register fj and the single-precision/double-precision floating-point number in floating-point register fk.

Multiply the numbers, subtract the single-precision/double-precision floating-point number in the floating-point register fa from the result, and then negate the single-precision/double-precision floating-point result.

Write it to the floating-point register fd.

FNMSUB.S:

FR[fd][31:0] = -FP32_fusedMultiplyAdd(FR[fj][31:0], FR[fk][31:0], -FR[fa]
[31:0])

FNMSUB.D:

FR[fd] = -FP64_fusedMultiplyAdd(FR[fj], FR[fk], -FR[fa])

The above four floating-point fused multiply-add operations follow the specifications of the fusedMultiplyAdd(x,y,z) operation in the IEEE 754-2008 standard.

3.2.1.3F{MAX/MIN}.{S/D}

Command format: fmax.s	fd, fj, fk	fmax.d	fd, fj, fk
fmin.s	fd, fj, fk	fmin.d	fd, fj, fk

The FMAX.{S/D} instruction selects a single-precision/double-precision floating-point number from the floating-point register fj and a single-precision/double-precision floating-point number from the floating-point register fk.

The larger of the two numbers is written to the floating-point register fd. The operation of these two instructions follows the rules of the maxNum(x,y) operation in the IEEE 754-2008 standard.

Fan.

FMAX.S:

FR[fd][31:0] = FP32_maxNum(FR[fj][31:0], FR[fk][31:0])

FMAX.D:

FR[fd] = FP64_maxNum(FR[fj], FR[fk])

The FMIN.{S/D} instruction selects a single-precision/double-precision floating-point number from floating-point register fj and a single-precision/double-precision floating-point number from floating-point register fk.

The smaller of the two numbers is written into the floating-point register fd. The operation of these two instructions follows the rules of the minNum(x,y) operation in the IEEE 754-2008 standard.

Fan.

FMIN.S:

FR[fd][31:0] = FP32_minNum(FR[fj][31:0], FR[fk][31:0])

FMIN.D:

FR[fd] = FP64_minNum(FR[fj], FR[fk])

3.2.1.4F{MAXA/MINA}.{S/D}

Command format: fmaxa.s	fd, fj, fk	fmaxa.d	fd, fj, fk
fmina.s	fd, fj, fk	fmina.d	fd, fj, fk

The FMAXA.{S/D} instruction selects a single-precision/double-precision floating-point number from floating-point register fj and a single-precision/double-precision floating-point number from floating-point register fk.

The larger absolute value of the points is written to the floating-point register fd. The operation of these two instructions follows the IEEE 754-2008 standard for maxNumMag(x,y).

Standard operating procedures.

FMAXA.S:

FR[fd][31:0] = FP32_maxNumMag(FR[fj][31:0], FR[fk][31:0])

FMAXA.D:

FR[fd] = FP64_maxNumMag(FR[fj], FR[fk])

The FMINA.{S/D} instruction selects a single-precision/double-precision floating-point number from the floating-point register fj and a single-precision/double-precision floating-point number from the floating-point register fk.

The smaller absolute value of the points is written into the floating-point register fd. The operation of these two instructions follows the IEEE 754-2008 standard for minNumMag(x,y).

Standard operating procedures.

FMINA.S:

FR[fd][31:0] = FP32_minNumMag(FR[fj][31:0], FR[fk][31:0])

FMINA.D:

FR[fd] = FP64_minNumMag(FR[fj], FR[fk])

3.2.1.5F{ABS/NEG}.{S/D}

Command format: fabs.s	fd, fj	fabs.d	fd, fj
fneg.s	fd, fj	fneg.d	fd, fj

The FABS.{S/D} instruction selects a single-precision or double-precision floating-point number from the floating-point register fj and takes its absolute value (i.e., sets the sign bit to 0).

(It remains partially unchanged) and is written to the floating-point register fd. The operation of these two instructions follows the specification of the abs(x) operation in the IEEE 754-2008 standard.

FABS.S:

FR[fd][31:0] = FP32_abs(FR[fj][31:0])

FABS.D:

FR[fd] = FP64_abs(FR[fj])

The FNEG.{S/D} instruction selects a single-precision/double-precision floating-point number from the floating-point register fj and inverts it (that is, inverts the sign bit and the rest).

(Partially unchanged), written to the floating-point register fd. The operation of these two instructions follows the specifications of the nexteer(x) operation in the IEEE 754-2008 standard.

FNEG.S:

FR[fd][31:0] = FP32_negate(FR[fj][31:0])

FNEG.D:

FR[fd] = FP64_negate(FR[fj])

3.2.1.6F{SQRT/RECIP/RSQRT}.{S/D}

Command format: fsqrt.s	fd, fj	fsqrt.d	fd, fj
frecip.s	fd, fj	frecip.d fd, fj	
frsqrt.s	fd, fj	frsqrt.d	fd, fj

The FSQRT.{S/D} instruction selects a single-precision or double-precision floating-point number from the floating-point register fj, and then takes the square root of the resulting single-precision or double-precision floating-point number.

The number is written to the floating-point register fd. The floating-point square root operation follows the specification of the squareRoot(x) operation in the IEEE 754-2008 standard.

FSQRT.S:

```
FR[fd][31:0] = FP32_squareRoot(FR[fj][31:0]);
```

FSQRT.D:

```
FR[fd] = FP64_squareRoot(FR[fj]);
```

The FRECIP.{S/D} instruction selects a single-precision or double-precision floating-point number from the floating-point register fj, and divides 1.0 by this floating-point number to obtain the single-precision or double-precision floating-point number.

The precision/double-precision floating-point number is written to the floating-point register fd. This is equivalent to the division(1.0,x) operation in the IEEE 754-2008 standard.

FRECIP.S:

```
FR[fd][31:0] = FP32_division(1.0, FR[fj][31:0])
```

FRECIP.D:

```
FR[fd] = FP64_division(1.0, FR[fj])
```

The FRSQRT.{S/D} instruction selects a single-precision/double-precision floating-point number from the floating-point register fj, and then takes the square root of the resulting single-precision/double-precision floating-point number.

Divide the number by 1.0 again, and write the resulting single-precision/double-precision floating-point number into the floating-point register fd. The floating-point square root and inverse operation follows IEEE 754-2008.

The specification of the rSqrt(x) operation in the standard.

FRSQRT.S:

```
FR[fd][31:0] = FP32_division(1.0, FP_squareRoot(FR[fj][31:0]));
```

FRSQRT.D:

```
FR[fd] = FP64_division(1.0, FP_squareRoot(FR[fj]));
```

3.2.1.7F{SCALEB/LOGB/COPYSIGN}.{S/D}

Command format:	fscaleb.s	fd, fj, fk	fscaleb.d	fd, fj, fk
	flogb.s	fd, fj	flogb.d	fd, fj
	fcopysign.s	fd, fj, fk	fcopysign.d	fd, fj, fk

The FSCALEB.{S/D} instruction selects a single-precision/double-precision floating-point number 'a' from floating-point register fj, then retrieves a word/double-word integer 'N' from floating-point register fk, calculates 'a*2N', and writes the resulting single-precision/double-precision floating-point number into floating-point register fd. The operation of these two instructions conforms to IEEE 754-2008.

The specification for the scaleB(x, N) operation in the standard.

FSCALEB.S:

```
FR[fd][31:0] = FP32_scaleB(FR[fj][31:0], FR[fk][31:0])
```

FSCALEB.D:

```
FR[fd] = FP64_scaleB(FR[fj], FR[fk])
```

The FLOGB.{S/D} instruction selects a single-precision/double-precision floating-point number from the floating-point register fj, calculates its logarithm to base 2, and obtains the single-precision/double-precision floating-point number.

The double-precision floating-point number is written to the floating-point register fd. The operation of these two instructions follows the specification of the logB(x) operation in the IEEE 754-2008 standard.

FLOGB.S:

```
FR[fd][31:0] = FP32_logB(FR[fj][31:0])
```

FLOGB.D:

```
FR[fd] = FP64_logB(FR[fj])
```

The FCOPYSIGN.{S/D} instruction selects the single-precision/double-precision floating-point number in the floating-point register fj and changes its sign bit to the floating-point register fk.

The sign bit of the single-precision/double-precision floating-point number is removed, and the resulting new single-precision/double-precision floating-point number is written to the floating-point register fd. (Floating-point copy character)

The sign operation follows the specification of the copySign(x, y) operation in the IEEE 754-2008 standard.

FCOPYSIGN.S:

FR[fd][31:0] = FP32_copySign(FR[fj][31:0], FR[fk][31:0])

FCOPYSIGN.D:

FR[fd] = FP64_copySign(FR[fj], FR[fk])

3.2.1.8FCLASS.{S/D}

Command format: fclass.s fd, fj fclass.d fd, fj

This instruction determines the category of the floating-point number in the floating-point register fj. The result consists of 10 bits of information, each bit...

The meanings are shown in the table below:

Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7	Bit8	Bit9
SNaN	QNaN	negative value				positive value			
		ÿ	normal	subnormal	0	ÿ	normal	subnormal	0

When the data being evaluated meets the condition corresponding to a certain bit, the corresponding bit in the result information vector will be set to 1. This instruction corresponds to...

The class(x) function in the IEEE-754-2008 standard.

FCLASS.S:

FR[fd][31:0] = FP32_class(FR[fj][31:0])

FCLASS.D:

FR[fd] = FP64_class(FR[fj])

3.2.1.9F{RECIPE/RSQRTE}.{S/D}

Command format: frecipe.s fd, fj frecipe.d fd, fj

frsqrte.s fd, fj frsqrte.d fd, fj

The FRECIPE.{S/D} instruction selects a single-precision or double-precision floating-point number from the floating-point register fj, and calculates the result by dividing 1.0 by this floating-point number.

The approximate results for single-precision/double-precision floating-point numbers are written to the floating-point register fd. The relative error of the obtained approximate results is less than 2ÿ¹ÿ .

When the input value is 2N , the output value is 2 ^(-N). The results for inputs of QNaN, SNaN, ±ÿ, ±0, and the conditions for generating floating-point exceptions are also provided.

The default result when a floating-point exception is generated but not triggered is the same as that of the FRECIP.{S/D} instruction.

FRECIPE.S:

FR[fd][31:0] = FP32_reciprocal_estimate(FR[fj][31:0])

FRECIPE.D:

FR[fd] = FP64_reciprocal_estimate(FR[fj])

The FRSQRTE.{S/D} instruction selects a single-precision/double-precision floating-point number from the floating-point register fj, and then takes the square root of the resulting single-precision/double-precision floating-point number.

The approximate result of dividing the point by 1.0 is written into the floating-point register fd to obtain a single-precision/double-precision floating-point number. The relative error of the obtained approximation result...

The difference is less than 2 -14.

When the input value is 2 ^2N , the output value is 2 ^-N. The results for inputs of QNaN, SNaN, ±ÿ, ±0, and the conditions for generating floating-point exceptions are as follows.

The default result when a floating-point exception is generated but not triggered is the same as that of the FRSQRT.{S/D} instruction.

```
FRSQRTE.S:
        FR[fd][31:0] = FP32_reciprocal_squareroot_estimate(FR[fj][31:0]);
FRSQRTE.D:
        FR[fd] = FP64_reciprocal_squareroot_estimate(FR[fj]);
```

3.2.2 Floating-point comparison instructions

3.2.2.1FCMP.cond.{S/D}

Command format: fcmp.cond.s cc, fj, fk fcmp.cond.d cc, fj, fk

This is a floating-point comparison instruction that stores the comparison result in the specified status code (CC). This instruction has 22 possible conditions (cond), which compare...

The comparison conditions and judgment criteria are listed in the table below.

mnemonic	cond	meaning	True Condition	QNaN whether Report exceptions	Corresponding to IEEE 754-2008 functions
CAF	0x0	no	none	no	
WITH	0x8	Incomparable	AND		compareQuietUnordered
CEQ	0x4	equal	EQ		compareQuietEqual
CUEQ	0xC	Equal or incomparable	EQ		
CLT	0x2	Less than	LT		compareQuietLess
CULT	0xA	Smaller than or cannot be compared	AND LT		compareQuietLessUnordered
CLE	0x6	Less than or equal to	LT EQ		compareQuietLessEqual
vesicles	0xE	Less than or equal to or cannot be compared	UN LT EQ		compareQuietNotGreater
CNE	0x10	Unequal	GT LT		
COR	0x14	orderly	GT LT EQ		
CUTE	0x18	Unable to be compared or unequal	UN GT LT		compareQuietNotEqual
SAF	0x1	no	none	yes	
SUN	0x9	is not greater than, less than, or equal to.	AND		
SEQ	0x5	equal	EQ		compareSignalingEqual
SUEQ	0xD	Not greater than or less than	EQ		
SLT	0x3	Less than	LT		compareSignalingLess
SULT	0xB	Not greater than or equal to	AND LT		compareSignalingLessUnordered
SLE	0x7	Less than or equal to	LT EQ		compareSignalingLessEqual
SULE	0xF	Not greater than	A LT EQ		compareSignalingNotGreater
SNOW	0x11	Unequal	GT LT		
BEER	0x15	orderly	GT LT EQ		
THEY ARE	0x19	Unable to be compared or unequal	UN GT LT		

Note: UN indicates that they cannot be compared, EQ indicates that they are equal, and LT indicates that they are less than. When two operands contain at least one NaN, these two...

Numbers cannot be compared.

3.2.3 Floating-point conversion instructions

3.2.3.1FCVT.S.D, FCVT.D.S

Command format: fcvt.sd	fd, fj	fcvt.d.s	fd, fj

The FCVT.SD instruction selects a double-precision floating-point number in the floating-point register fj, converts it to a single-precision floating-point number, and writes the resulting single-precision floating-point number to fj.

In the floating-point register fd.

FCVT.S.D:

```
FR[fd][31:0] = FP32_convertFormat(FR[fj], FP64)
```

The FCVT.DS instruction selects a single-precision floating-point number in the floating-point register fj, converts it to a double-precision floating-point number, and writes the resulting double-precision floating-point number to fj.

In the floating-point register fd.

FCVT.D.S:

```
FR[fd] = FP64_convertFormat(FR[fj][31:0], FP32)
```

Floating-point format conversion operations follow the specifications of the convertFormat(x) operation in the IEEE 754-2008 standard.

3.2.3.2FFINT.{S/D}.{W/L}, FTINT.{W/L}.{S/D}

Command format: ffint.sw fd, fj		ftint.w.s fd, fj	
ffint.s.l	fd, fj	ftint.ls	fd, fj
ffint.d.w fd, fj		ftint.w.d fd, fj	
ffint.d.l	fd, fj	ftint.ld	fd, fj

The FFINT.{S/D}.{W/L} instruction selects the integer/long integer fixed-point number in the floating-point register fj and converts it to a single-precision/double-precision floating-point number.

The obtained single-precision/double-precision floating-point number is written to the floating-point register fd. This floating-point format conversion operation conforms to the IEEE 754-2008 standard.

Specifications for the convertFromInt(x) operation.

FFINT.S.W:

```
FR[fd][31:0] = FP32_convertFromInt(FR[fj][31:0], SINT32)
```

FFINT.S.L:

```
FR[fd][31:0] = FP32_convertFromInt(FR[fj], SINT64)
```

FFINT.D.W:

```
FR[fd] = FP64_convertFromInt(FR[fj][31:0], SINT32)
```

FFINT.D.L:

```
FR[fd] = FP64_convertFromInt(FR[fj], SINT64)
```

The FTINT.{W/L}.{S/D} instruction selects the single-precision/double-precision floating-point number in the floating-point register fj and converts it to an integer/long integer fixed-point number.

The resulting integer/long integer fixed-point number is written to the floating-point register fd. Depending on the state in the FCSR, this floating-point format conversion operation follows...

The operation in accordance with the IEEE 754-2008 standard is shown in the table below.

Rounding mode	Operations in the IEEE 754-2008 standard
Round to the nearest integer (eventh integer).	convertToIntegerExactTiesToEven(x)
Rounding to zero	convertToIntegerExactTowardZero(x)
Rounding to positive infinity	convertToIntegerExactTowardPositive(x)
Rounding to negative infinity	convertToIntegerExactTowardNegative(x)

FTINT.W.S:

FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], FCSR.RM)

FTINT.WD:

FR[fd][31:0] = FP64convertToSint32(FR[fj], FCSR.RM)

FTINT.LS:

FR[fd] = FP32convertToSint64(FR[fj][31:0], FCSR.RM)

FTINT.LD:

FR[fd] = FP64convertToSint64(FR[fj], FCSR.RM)

3.2.3.3FTINT{RM/RP/RZ/RNE}.{W/L}.{S/D}

Command format:	ftintrm.ws	fd, fj	ftintrp.ws	fd, fj
	ftintrm.w.d	fd, fj	ftintrp.wd	fd, fj
	ftintrm.l.s	fd, fj	ftintrp.l.s	fd, fj
	ftintrm.l.d	fd, fj	ftintrp.l.d	fd, fj
	ftintrz.ws	fd, fj	ftintrne.w.s	fd, fj
	ftintrz.wd	fd, fj	ftintrne.w.d	fd, fj
	ftintrz.l.s	fd, fj	ftintrne.l.s	fd, fj
	ftintrz.ld	fd, fj	ftintrne.l.d	fd, fj

These instructions convert floating-point numbers to fixed-point numbers using a specified rounding mode.

The FTINTRM.{W/L}.{S/D} instruction selects the single-precision/double-precision floating-point number in the floating-point register fj and converts it to an integer/long integer fixed-point number.

The integer/long integer fixed-point number obtained is written into the floating-point register fd, using the "rounding towards negative infinity" method.

FTINTRM.W.S:

FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], 3)

FTINTRM.W.D:

FR[fd][31:0] = FP64convertToSint32(FR[fj], 3)

FTINTRM.L.S:

FR[fd] = FP32convertToSint64(FR[fj][31:0], 3)

FTINTRM.L.D:

FR[fd] = FP64convertToSint64(FR[fj], 3)

The FTINTRP.{W/L}.{S/D} instruction selects the single-precision/double-precision floating-point number in the floating-point register fj and converts it to an integer/long integer fixed-point number.

The obtained integer/long integer fixed-point number is written into the floating-point register fd, using the "rounding towards positive infinity" method.

FTINTRP.WS:

FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], 2)

FTINTRP.WD:

FR[fd][31:0] = FP64convertToSint32(FR[fj], 2)

FTINTRP.L.S:

FR[fd] = FP32convertToSint64(FR[fj][31:0], 2)

FTINTRP.L.D:

FR[fd] = FP64convertToSint64(FR[fj], 2)

The FTINTRZ.{W/L}.{S/D} instruction selects the single-precision/double-precision floating-point number in the floating-point register fj and converts it to an integer/long integer fixed-point number.

The obtained integer/long integer fixed-point number is written into the floating-point register fd, using the "rounding towards zero" method.

FTINTRZ.WS:

FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], 1)

FTINTRZ.WD:

FR[fd] = FP64convertToSint32(FR[fj], 1)

FTINTRZ.L.S:

FR[fd][31:0] = FP32convertToSint64(FR[fj][31:0], 1)

FTINTRZ.LD:

FR[fd] = FP64convertToSint64(FR[fj], 1)

The FTINTRNE.{W/L}.{S/D} instruction selects the single-precision/double-precision floating-point number in the floating-point register fj and converts it to an integer/long integer fixed-point number.

The integer/long integer fixed-point number obtained is written into the floating-point register fd, using the "rounding to the nearest (eventh)" method.

FTINTRNE.W.S:

FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], 0)

FTINTRNE.W.D:

FR[fd][31:0] = FP64convertToSint32(FR[fj], 0)

FTINTRNE.L.S:

FR[fd] = FP32convertToSint64(FR[fj][31:0], 0)

FTINTRNE.L.D:

FR[fd] = FP64convertToSint64(FR[fj], 0)

The operations in the IEEE 754-2008 standard followed by the above four floating-point format conversion operations are shown in the table below.

	IEEE 754-2008
FTINTRNE.{W/L}.{S/D}	convertToIntegerExactTiesToEven(x)
FTINTRZ.{W/L}.{S/D}	convertToIntegerExactTowardZero(x)
FTINTRP.{W/L}.{S/D}	convertToIntegerExactTowardPositive(x)
FTINTRM.{W/L}.{S/D}	convertToIntegerExactTowardNegative(x)

3.2.3.4FRINT.{S/D}

Command format: frint.s fd, fj frint.d fd, fj

The FRINT.{S/D} instruction selects a single-precision/double-precision floating-point number in the floating-point register fj and converts it to an integer single-precision/double-precision floating-point number.

The resulting single-precision/double-precision floating-point number is written to the floating-point register fd. This floating-point format conversion operation follows the IEEE 754-2008 standard.

The instructions are shown in the table below.

Rounding mode	Operations in the IEEE 754-2008 standard
Round to the nearest integer (eventh integer).	roundToIntegralExact(x)
Rounding to zero	
Rounding to positive infinity	
Rounding to negative infinity	

FRINT.Sÿ

FR[fd][31:0] = FP32_roundToInteger(FR[fj])

FRINT.Dÿ

FR[fd] = FP64_roundToInteger(FR[fj])

3.2.4 Floating-point transport instructions

3.2.4.1FMOV.{S/D}

Command format: fmov.s fd, fj fmov.d fd, fj

FMOV.{S/D} writes the value of floating-point register fj to floating-point register fd in single-precision/double-precision floating-point format. If the value of fj is not...

If it is a single-precision/double-precision floating-point number format, the result is uncertain.

FMOV.Sÿ

FR[fd][31:0] = FR[fj][31:0]

FMOV.dÿ

FR[fd] = FR[fj]

The above instruction operation is non-arithmic, will not trigger an IEEE 754 exception, and will not modify the Cause and Flags of the floating-point control status register.

domain.

3.2.4.2FSEL

Command format: fsel fd, fj, fk, ca

The FSEL instruction performs a conditional assignment operation. When FSEL is executed, if the value of the condition flag register ca is equal to 0, then the value of the floating-point register fj is incremented.

The value is written to the floating-point register fd; otherwise, the value of the floating-point register fk is written to the floating-point register fd.

FSELÿ

FR[fd] = CFR[ca] ? FR[fk] : FR[fj]

3.2.4.3MOVGR2FR.{W/D}, MOVGR2FRH.W

Command format: movgr2fr.w fd, rj movgr2fr.d fd, rj

movgr2frh.w fd, rj

MOVGR2FR.W writes the lower 32 bits of the general-purpose register rj into the lower 32 bits of the floating-point register fd. If the floating-point register has a bit width of 64...

If the higher 32 bits of fd are not specified, then the value of fd is uncertain.

MOVGR2FR.Wÿ

FR[fd][31:0] = GR[rj][31:0]

MOVGR2FRH.W writes the lower 32 bits of the general-purpose register rj into the higher 32 bits of the floating-point register fd.

The bit value remains unchanged.

MOVGR2FRH.Wÿ

FR[fd][63:32] = GR[rj][31:0]

FR[fd][31: 0] = FR[fd][31:0]

MOVGR2FR.D writes the 64-bit value of the general-purpose register rj to the floating-point register fd.

MOVGR2FR.Dÿ

FR[fd] = GR[rj]

3.2.4.4MOVFR2GR.{S/D}, MOVFRH2GR.S

Command format: movfr2gr.s rd, fj movfr2gr.d rd, fj

movfrh2gr.s rd, fj

MOVFR2GR/MOVFRH2GR.S writes the sign-extended value of the lower 32 bits/higher 32 bits of the floating-point register fj to the general-purpose register rd.

MOVFR2GR.Sÿ

GR[rd] = SignExtend(FR[fj][31:0], GRLEN)

MOVFRH2GR.Sÿ

GR[rd] = SignExtend(FR[fj][63:32], GRLEN)

MOVFR2GR.D writes the 64-bit value of the floating-point register fj to the general-purpose register rd.

MOVFR2GR.Dÿ

GR[rd] = FR[fj]

3.2.4.5MOVGR2FCSR, MOVFCSR2GR

Command format: movgr2fcsr fcsr, rj

movfcsr2gr rd, fcsr

MOVGR2FCSR modifies the software-writable field corresponding to the floating-point control status register indicated by fcsr based on the value of the lower 32 bits of the general-purpose register rj.

The value. If the MOVGR2FCSR instruction modifies FCSR0 so that the bit in its Cause field and the corresponding Enables bit are both 1, or modifies...

The Enables field of FCSR1 and the Cause field of FCSR2 are set so that both the Cause bit and the corresponding Enables bit are 1, as specified in the MOVGR2FCSR instruction.

It will not trigger a floating-point exception on its own.

MOVGR2FCSRÿ

FCSR[fcsr] = GR[rj][31:0]

MOVFCSR2GR writes the 32-bit value of the floating-point control status register indicated by fcsr to the general-purpose register rd after sign extension.

MOVFCSR2GRÿ

GR[rd] = SignExtend(FCSR[fcsr], GRLEN)

If the floating-point control status register indicated by fcsr in the above instruction does not exist, the result is uncertain.

3.2.4.6MOVFR2CF, MOVCF2FR

Command format: movfr2cf cd, fj

movcf2fr fd, cj

MOVFR2CF writes the value of the least significant bit of the floating-point register fj to the condition flag register cd.

MOVFR2CFÿ

CFR[cd] = FR[fj][0]

MOVCF2FR writes the value of the condition flag register cj to the lowest bit of the floating-point register fd, and pads the remaining bits of fd with 0.

MOVCF2FRÿ

FR[fd] = ZeroExtend(CFR[cj], 64)

3.2.4.7MOVGR2CF, MOVCF2GR

Command format: movgr2cf cd, rj

movcf2gr rd, cj

MOVGR2CF writes the value of the least significant bit of the general-purpose register rj to the condition flag register cd.

MOVGR2CFÿ

CFR[cd] = GR[rj][0]

MOVCF2GR writes the value of the condition flag register cj to the general-purpose register rd, and fills the remaining bits of rd with 0.

MOVCF2GRÿ

GR[rd] = ZeroExtend(CFR[cj], GRLEN)

3.2.5 Floating-point branch instructions

3.2.5.1BCEQZ, BCNEZ

Command format:	bceqz	cj, offs21
	bcnez	cj, offs21

BCEQZ checks the value of the condition flag register cj. If it is equal to 0, it jumps to the target address; otherwise, it does not jump.

BCNEZ checks the value of the condition flag register cj. If the value is not equal to 0, it jumps to the target address; otherwise, it does not jump.

```
BCEQZ:
    if CFR[cj]==0 :
        PC = PC + SignExtend({offs21, 2'b0}, GRLEN)
BCNEZ:
    if CFR[cj]!=0 :
        PC = PC + SignExtend({offs21, 2'b0}, GRLEN)
```

The jump target address of the two branch instructions mentioned above is obtained by logically shifting the 21-bit immediate value offs21 in the instruction code left by 2 bits and then sign-extending it.

The resulting offset value is added to the PC of the branch instruction.

However, it should be noted that if the above instructions are written by directly filling in the offset value when writing the assembly code, the immediate value in the assembly representation should be...

Enter the offset value in bytes, which is offs21<<2 in the instruction code.

3.2.6 Floating-point ordinary memory access instructions

3.2.6.1FLD.{S/D}, FST.{S/D}

Command format:	fld.s	fd, rj, si12	fld.d	fd, rj, si12
	fst.s	fd, rj, si12	fst.d	fd, rj, si12

FLD.S retrieves a word of data from memory and writes it to the lower 32 bits of the floating-point register fd. If the floating-point register is 64 bits wide, then the higher 32 bits of fd...

The 32-bit value is uncertain. FLD.D retrieves a double word of data from memory and writes it to the floating-point register fd.

```
FLD.S:
    vaddr = GR[rj] + SignExtend(si12, GRLEN)
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    word = MemoryLoad(paddr, WORD)
    FR[fd][31:0] = word
FLD.D:
    vaddr = GR[rj] + SignExtend(si12, GRLEN)
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    doubleword = MemoryLoad(paddr, DOUBLEWORD)
    FR[fd] = doubleword
```

FST.S writes the lower 32 bits of the floating-point register fd into memory. FST.D writes a double word of data from the floating-point register fd into memory.

middle.

FST.S:

```
    vaddr = GR[rj] + SignExtend(si12, GRLEN)
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    MemoryStore(FR[fd][31:0], paddr, WORD)
```

FST.D:

```
    vaddr = GR[rj] + SignExtend(si12, GRLEN)
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    MemoryStore(FR[fd][63:0], paddr, DOUBLEWORD)
```

The memory address of the above instruction is calculated by adding the value in the general-purpose register rj to the sign-extended 12-bit immediate value si12.

For the FLD.{S/D} and FST.{S/D} instructions, regardless of the hardware implementation or environment configuration, as long as their memory access address is naturally aligned...

Naturally aligned memory addresses will not trigger unaligned exceptions; however, when the memory access address is not naturally aligned, if the hardware implementation supports unaligned memory access and the current operation cycle...

If the environment is configured to allow unaligned memory access, then the unaligned exception will not be triggered; otherwise, the unaligned exception will be triggered.

3.2.6.2FLDX.{S/D}, FSTX.{S/D}

Command format: fldx.s	fd, rj, rk	fldx.d	fd, rj, rk
fstx.s	fd, rj, rk	fstx.d	fd, rj, rk

FLDX.S retrieves a word of data from memory and writes it to the lower 32 bits of the floating-point register fd. If the floating-point register is 64 bits wide, then fd...

The value of the high 32 bits is uncertain. FLDX.D retrieves a double word of data from memory and writes it to the floating-point register fd.

FLDX.S:

```
    vaddr = GR[rj] + GR[rk]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    word = MemoryLoad(paddr, WORD)
    FR[fd][31:0] = word
```

FLDX.D:

```
    vaddr = GR[rj] + GR[rk]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    doubleword = MemoryLoad(paddr, DOUBLEWORD)
    FR[fd] = doubleword
```

FSTX.S writes the lower 32 bits of the floating-point register fd to memory. FSTX.D writes a double-word of data from the floating-point register fd to memory.

In memory.

FSTX.S:

```
vaddr = GR[rj] + GR[rk]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(FR[fd][31:0], paddr, WORD)
```

FSTX.D:

```
vaddr = GR[rj] + GR[rk]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(FR[fd][63:0], paddr, DOUBLEWORD)
```

The memory access address of the above instruction is calculated by adding the value in general-purpose register rj to the value in general-purpose register rk.

For the FLDX.{S/D} and FSTX.{S/D} instructions, regardless of the hardware implementation or environment configuration, as long as their memory access address is...

Naturally aligned memory will not trigger unaligned exceptions; however, when the memory access address is not naturally aligned, if the hardware implementation supports unaligned memory access and the current operation...

If the computing environment is configured to allow unaligned memory access, then the unaligned exception will not be triggered; otherwise, the unaligned exception will be triggered.

3.2.7 Floating-point boundary check memory access instructions

3.2.7.1FLD{GT/LE}.{S/D}, FST{GT/LE}.{S/D}

Command format:	fldgt.s	fd, rj, rk	fldgt.d	fd, rj, rk
	fldle.s	fd, rj, rk	fldle.d	fd, rj, rk
	fstgt.s	fd, rj, rk	fstgt.d	fd, rj, rk
	fstle.s	fd, rj, rk	fstle.d	fd, rj, rk

FLD{GT/LE}.{S/D} checks if the valid address is out of bounds, and retrieves the value from memory and writes it to the floating-point register.

FLD{GT/LE}.S checks if the value in general-purpose register rj is greater than or less than or equal to the value in general-purpose register rk. If the condition is met, it proceeds from...

A word of data is retrieved from memory and written to the lower 32 bits of the floating-point register fd. If the floating-point register is 64 bits wide, the higher 32 bits of fd are undefined.

Certainly.

FLD{GT/LE}.D checks if the value in general-purpose register rj is greater than or less than or equal to the value in general-purpose register rk. If the condition is met, it proceeds from...

Retrieve a double word of data from memory and write it to the floating-point register fd.

FLDGT.S:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    word = MemoryLoad(paddr, WORD)
    FR[fd][31:0] = word
```

```
else :
    RaiseException(BCE) #Bound Check Error Exception
```

FLDGT.D:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    FR[fd] = MemoryLoad(paddr, DOUBLEWORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

FLDLE.S:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    word = MemoryLoad(paddr, WORD)
    FR[fd][31:0] = word
else :
    RaiseException(BCE) #Bound Check Error Exception
```

FLDLE.D:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    FR[fd] = MemoryLoad(paddr, DOUBLEWORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

FST{GT/LE}.{S/D} checks if the valid address is out of bounds and writes the value of the floating-point register to memory.

FST{GT/LE}.S checks if the value in general-purpose register rj is greater than or less than or equal to the value in general-purpose register rk. If the condition is met, then...

The lower 32 bits of the floating-point register fd are written into memory.

FST{GT/LE}.D checks if the value in general-purpose register rj is greater than or less than or equal to the value in general-purpose register rk. If the condition is met, then...

The double-word data in the floating-point register fd is written into memory.

FSTGT.S:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    MemoryStore(FR[fd][31:0], paddr, WORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

FSTGT.D:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]>GR[rk] :
    MemoryStore(FR[fd][63:0], paddr, DOUBLEWORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

FSTLE.S:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    MemoryStore(FR[fd][31:0], paddr, WORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

FSTLE.D:

```
vaddr = GR[rj]
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
if GR[rj]<=GR[rk] :
    MemoryStore(FR[fd][63:0], paddr, DOUBLEWORD)
else :
    RaiseException(BCE) #Bound Check Error Exception
```

The memory access addresses for the above instructions come directly from the value in the general-purpose register rj. All memory access addresses for the above instructions require natural alignment; otherwise, [the memory will be...].

Trigger an unalignment exception. If the above instruction fails to meet the check conditions, it terminates the memory access operation and triggers a boundary check error exception.

4. Overview of Privileged Resource Architecture

4.1 Privilege Levels

In the Dragon architecture, processor cores are divided into four privilege levels (PLVs): PLV0 to PLV3.

The privilege level previously held is uniquely determined by the value of the PLV field in CSR.CRMD.

Of all privilege levels, PLV0 is the highest privilege level, and the only one that can use privileged instructions and access all privileged resources.

The privilege levels PLV1 through PLV3 are not allowed to execute privileged instructions or access privileged resources. However, these three privilege levels are configured in the MMU.

Different access permissions are used when using the mapped address translation mode.

For Linux systems, only PLV0 level in the architecture corresponds to kernel mode, while it is recommended to use PLV3 level to correspond to user mode.

4.2 Overview of Privileged Instructions

All privileged instructions are accessible only at the PLV0 privilege level. There is only one exception. , When in CSR.MISC

When RPCNTL1/RPCNTL2/RPCNTL3 is configured to 1, CSRRD instruction reads can be executed under the PLV1/PLV2/PLV3 privilege levels.

Energy counter.

4.2.1 CSR Access Commands

4.2.1.1CSRRD, CSRWR, CSRXCHG

Command format: csrrd	rd, csr_num
creator	rd, csr_num
csrxchg	rd, rj, csr_num

The CSRRD, CSRWR, and CSRXCHG instructions are used for software access to CSRs. The CSRRD instruction writes the value of the specified CSR to a general-purpose register.

The CSRWR instruction writes the old value from the general-purpose register rd to the specified CSR, and simultaneously updates the old value of the specified CSR to the general-purpose register rd.

The CSRXCHG instruction, based on the write mask information stored in the general-purpose register rj, writes the old value from the general-purpose register rd to the bits in the specified

CSR where the write mask is 1. The remaining bits in the CSR remain unchanged, and the old value of the CSR is updated in the general-purpose register rd.

In the device rd.

All CSR registers use an independent address space. In the above instructions, the address value of the CSR comes from the 14-bit immediate value csr_num in the instruction.

The addressing unit of a CSR is a CSR register, that is, the csr_num of CSR 0 is 0, the csr_num of CSR 1 is 1, and so on.

All CSR registers are either 32 bits wide or the same width as the GR registers in the architecture; therefore, CSR access instructions are not distinguishable by bit width.

In the LA32 architecture, all CSRs are naturally 32 bits wide. In the LA64 architecture, CSRs with a fixed width of 32 bits in the definition are always sign-expanded.

The result is written to the general-purpose register rd after the expansion.

When a CSR access instruction accesses a CSR that is not defined in the architecture or not implemented in the hardware, the read operation can return any value (returning all zeros is recommended).

The write operation does not modify any software-visible state of the processor. It is important to note that the CSRWR and CSRXCHG instructions not only contain...

The write action to update the CSR also includes the read action to read the old value of the CSR.

4.2.2 IOCSR Access Commands

4.2.2.1IOCSR{RD/WR}.{B/H/W/D}

Command format: iocsrrd.b	rd, rj
iocsrrd.h	rd, rj
iocsrrd.w	rd, rj
iocsrrd.d	rd, rj
yoxrwr.b	rd, rj
yoxrwr.h	rd, rj
iocsrwr.w	rd, rj
iocsrwr.d	rd, rj

The IOCSR{RD/WR}.{B/H/W/D} instruction is used to access the IOCSR.

All IOCSR registers use independent addressing spaces, with the basic addressing unit being the byte. All data in the IOCSR space is little-endian.

Storage format. The IOCSR space uses direct address mapping, where the physical address is directly equal to the logical address. IOCSR{RD/WR}.{B/H/W/D}

The instruction's IOCSR address comes from the general-purpose register rj.

The IOCSRRD.{B/H/W} instruction retrieves a byte/half-word/word of data from the specified address in the IOCSR space, sign-extends it, and writes it to the specified address.

Using register rd, the IOCSRRD.D instruction retrieves a double-word length of data from the specified address in the IOCSR space, sign-extends it, and writes it to the general purpose register.

In register rd.

The IOCSRWR.{B/H/W/D} instruction writes bits [7:0]/[15:0]/[31:0]/[63:0] from the general-purpose register rd to the IOCSR space.

The starting point of the address.

The IOCSRRD.D and IOCSRWR.D instructions only appear in the LA64 architecture.

The IOCSR register can typically be accessed simultaneously by multiple processor cores. The execution of IOCSR access instructions on multiple processor cores must satisfy a sequential order.

Coherence conditions.

4.2.3 Cache Maintenance Instructions

4.2.3.1CACOP

Command format: cacop code, rj, si12

The CACOP instruction is primarily used for cache initialization and cache consistency maintenance.

Adding the value of the general-purpose register rj to the sign-extended 12-bit immediate value si12 will yield the virtual address VA used by the CACOP instruction, which will...

Used to indicate the location of the cache line being operated on.

The CACOP instruction determines which cache it accesses and what cache operation it performs, determined by the 5-bit code in the instruction. code[2:0] indicates the operation.

The Cache object, code[4:3] indicates the operation type.

The cache object indicated by code[2:0] is consistent with the cache order identified in CPUCFG10. For example, when CPUCFG10=0x02C3D

At that time, code[2:0]=0 indicates operation on the first-level private instruction cache, code[2:0]=1 indicates operation on the first-level private data cache, and code[2:0]=2 indicates...

The operation of the second-level private hybrid cache, code[2:0]=3 indicates the operation of the third-level shared hybrid cache.

`code[4:3]=0` indicates that this is used for cache initialization (Store Tag), setting the tag of the specified cache line to all zeros. Assuming the accessed cache...

There are (1<<Way) paths, each path has (1<<Index) cache lines, and each cache line is (1<<Offset) bytes in size. Then, the...

Using direct address indexing means operating on the VA[Index+Offset-1:Offset]th cache line of the VA[Way-1:0]th path of this cache.

code[4:3]=1 indicates that the cache consistency is maintained by direct address indexing (Index Invalidate / Invalidate and Writeback).

Please refer to the previous paragraph for the definition of direct address indexing. Maintaining consistency involves invalidating and writing back a specified cache entry.

If the operation is on the instruction cache, then only an invalidation operation is needed; it is not necessary to write back the data in the cache line. The written-back data then...

The storage level into which data is stored is determined by the specific cache hierarchy implemented and the inclusion or mutual exclusion relationships between each level. For data caches or hybrid caches,

The specific implementation determines whether to write back cached line data only when the cache line data is dirty.

code[4:3]=2 indicates that the cache consistency is maintained using a query index method (Hit Invalidate / Invalidate and Writeback).

The operations for maintaining cache consistency are the same as described in the previous paragraph. The so-called lookup index method treats the VA of the CACOP instruction as a regular...

The `load` instruction accesses the cache to be operated on. If a cache hit occurs, the operation is performed on the hit cache line; otherwise, no operation is performed. Because of this...

The query process may involve virtual-to-physical address translation, so in this case, the CACOP instruction may trigger TLB-related exceptions. However, because CACOP...

The instruction operates on cache lines, so address alignment is not a concern in this case.

code[4:3]=3 is a custom cache operation that is not explicitly defined in the architecture specification.

4.2.4 TLB Maintenance Commands

4.2.4.1TLBSRCH

Command format: tlbsrch

This section provides the functional definition of the TLBSRCH instruction when the LVZ extension is not implemented.

Use the CSR.ASID and CSR.TLBEHI information to query the TLB. If a match is found, write the index value of the match to...

The index field of CSR.TLBIDX is set, and the NE position of CSR.TLBIDX is set to 0; if no item is found, then the index field of CSR.TLBIDX is set to 0.

The NE position is 1.

The index value calculation rule for each item in the TLB is to start from 0 and increment sequentially, first for STLB then for MTLB. In STLB, the index value starts from the 0th path.

From row 0 to the last row, then from row 0 to the last row of the first path, until the last row of the last path, MTLB is from row 0 to the last row.

4.2.4.2TLBRD

Command format: tlbrd

This section provides the functional definition of the TLBRD instruction when the LVZ extension is not implemented.

The value of the Index field of CSR.TLBIDX is used as the index to read the specified item in the TLB. If the specified position is a valid TLB...

If an item is specified, then the page table information of that TLB item is written to CSR.TLBEHI, CSR.TLBELO0, CSR.TLBELO1, and CSR.TLBIDX.PS

In the middle, set the NE bit of CSR.TLBIDX to 0; if the specified position is an invalid TLB entry, the NE bit of CSR.TLBIDX must be set to 0.

Set to 1, and it is recommended to mask and protect the read content, such as CSR.ASID.ASID, CSR.TLBEHI, CSR.TLBELO0, CSR.TLBELO1.

Neither CSR.TLBIDX.PS is updated or all are set to 0.

It is important to note that valid/invalid TLB entries and valid/invalid page table entries in the TLB are two different concepts.

If the index value used for access exceeds the range of the TLB, the processor's behavior is unpredictable.

4.2.4.3TLBWR

Command format: tlbwr

This section provides the functional definition of the TLBWR instruction when the LVZ extension is not implemented.

The TLBWR instruction writes page table entry information stored in the relevant CSR in the TLB to a specified entry in the TLB. The page table entry information being filled in comes from...

From CSR.TLBEHI, CSR.TLBELO0, CSR.TLBELO1, and CSR.TLBIDX.PS. If CSR.TLBRERA.IsTLBR=1 at this time,

If the TLB is in the process of being refilled as an exception, then a valid entry will always be filled into the TLB (i.e., the E bit of the TLB entry will be 1). Otherwise,

This requires checking the value of the CSR.TLBIDX.NE bit. If CSR.TLBIDX.NE = 1, then an invalid TLB entry will be filled into the TLB.

A valid TLB entry will only be filled into the TLB when CSR.TLBIDX.NE=0.

The location where page table entries are written to the TLB is specified by the value of the Index field of CSR.TLBIDX. For specific rules, please refer to TLBSRCH.

The instructions specify the calculation rules for each index value in the TLB. If a page table entry is to be written to the STLB, but the Index field of CSR.TLBIDX...

If there is a conflict between the value of VPPN and CSR.TLBEHI and CSR.TLBIDX.PS, then the processor's behavior is uncertain.

4.2.4.4TLBFILL

Command format: tlbfill

This section defines the functionality of the TLBFILL instruction when LVZ extensions are not implemented.

The TLBFILL instruction fills the TLB with page table entry information stored in the relevant CSR. The page table entry information being filled comes from...

CSR.TLBEHI, CSR.TLBELO0, CSR.TLBELO1, and CSR.TLBIDX.PS. If CSR.TLBRERA.IsTLBR=1 at this time, that is, at...

During TLB refill exception handling, a valid entry is always filled into the TLB (i.e., the E bit of the TLB entry is 1). Otherwise, it is necessary to...

You need to check the value of the CSR.TLBIDX.NE bit. If CSR.TLBIDX.NE = 1, then an invalid TLB entry will be filled into the TLB; only when...

A valid TLB entry will only be filled into the TLB when CSR.TLBIDX.NE=0.

When page table entries are populated, the first step is to determine whether to write to the STLB or MTLB based on the page size of the page table entry being populated.

Page entries will be populated in the STLB if the page size is equal to the page size configured in the STLB (CSR.STLBPS); otherwise, they will be populated in the MTLB.

Which path in the STLB or which item in the MTLB is entered is randomly selected by the hardware.

4.2.4.5TLBCLR

Command format: tlbclr

The content in the TLB is invalidated based on the information in the TLB-related CSR to maintain the consistency of page table data between the TLB and memory. This is given here.

The function definition of the TLBCLR instruction without implementing LVZ extension.

When CSR.TLBIDX.Index falls within the MTLB range (greater than or equal to the STLB number of items), TLBCLR is executed, removing all items from the MTLB.

Page table entries with G=0 and ASID equal to CSR.ASID.ASID are invalidated.

When CSR.TLBIDX.Index falls within the STLB range (less than the number of STLB items), execute a TLBCLR statement to change the STLB value from the specified value.

All page table entries in the group indicated by the low bit of CSR.TLBIDX.Index that are equal to G=0 and have ASID equal to CSR.ASID.ASID are invalidated.

4.2.4.6TLBFLUSH

Command format: tlbflush

The content in the TLB is invalidated based on the information in the TLB-related CSR to maintain the consistency of page table data between the TLB and memory. This is given here.

The function definition of the TLBFLUSH instruction without implementing LVZ extension.

When CSR.TLBIDX.Index falls within the MTLB range (greater than or equal to the STLB number of items), execute TLBFLUSH to move all items in the MTLB.

Page table entries are invalidated.

When CSR.TLBIDX.Index falls within the STLB range (less than the number of STLB items), execute a TLBFLUSH statement to move items from the STLB.

The page table entries in all paths of the group indicated by the low bit of CSR.TLBIDX.Index are invalidated.

4.2.4.7INVTLB

Command format: invtlb up, rj, rk

The INVTLB instruction is used to invalidate the contents of the TLB to maintain page table data consistency between the TLB and memory. Here is an example of a non-implemented LVZ.

In the extended case, the function definition of the INVTLB instruction.

Of the three source operands of the instruction, op is a 5-bit immediate value used to indicate the operation type.

Bits [9:0] of the general-purpose register rj store the ASID information required for invalid operations (called the "register-specified ASID"). The remaining bits must be filled.

0. When the operation indicated by op does not require ASID, the general-purpose register rj should be set to r0.

The general-purpose register rk stores the virtual address information (called the "register specification VA") required for invalid operations. When op indicates...

When the operation does not require virtual address information, the general-purpose register rk should be set to r0.

The operations corresponding to each op are shown in the table below. Ops that do not appear in the table will trigger reserved instruction exceptions.

on	operate
0x0	Clear all page table entries.
0x1	Clears all page table entries. This operation has the same effect as op=0.
0x2	Clear all page table entries where G=1.
0x3	Clear all page table entries where G=0.
0x4	Clears all page table entries where G=0 and ASID equals the ASID specified in the register.
0x5	Clears page table entries where G=0, ASID equals the ASID specified in the register, and VA equals the VA specified in the register.
0x6	Clears all page table entries where G=1 or ASID equals the ASID specified in the register and VA equals the VA specified in the register.

4.2.5 Software Page Table Traversal Instructions

4.2.5.1LDDIR

Command format: lddir rd, rj, level

The LDDIR instruction is used to access directory entries during software page table traversal.

The 8-bit immediate value `level` in the `LDDIR` instruction indicates which level of page table is being accessed. `level=1` corresponds to the value in `CSR.PWCL`.

Dir1, level=2 corresponds to Dir2 in CSR.PWCL; level=3 corresponds to Dir3 in CSR.PWCH; level=4 corresponds to Dir3 in CSR.PWCH.

Dir4.

If bit [6] of the general-purpose register rj is 0, it indicates that the value of the general-purpose register rj is the physical address of the base address of the level-1 page table.

When the LDDIR instruction is executed, it will access the level-1 page table based on the currently processed TLB refill address and retrieve the base address of the next-level page table.

Write to the general-purpose register rd. Note that the next level page table after the level 1 page table is not limited to the level-1 page table.

If bit [6] of the general-purpose register rj is 1, it indicates that the value of the general-purpose register rj is a page table entry of a HugePage.

Step by step, if bits [14:13] of the general-purpose register rj are equal to 0, it indicates that the page table entry for that large page has not yet been marked with the corresponding page table level.

In this case, replace bits [14:13] of the general-purpose register rj with level[1:0] and then write the entire value into the general-purpose register rd; if the general-purpose register

If bits [14:13] of rj are not equal to 0, it indicates that the page table entry for that large page has been marked with the corresponding page table level. In this case, the general-purpose register rj will be...

The value is written directly into the general-purpose register rd.

4.2.5.2LDPTE

Command format: ldpte rj, seq

The LDPTE instruction is used to access page table entries during software page table traversal.

The immediate value `seq` in the LDPTE instruction indicates whether the page being accessed is even or odd. When accessing an even page, the result will be written to...

CSR.TLBRELO0 means that when accessing odd-numbered pages, the results will be written to CSR.TLBRELO1.

If bit [6] of the general-purpose register rj is 0, it indicates that the content in rj is the physical address of the base address of the page table at the PTE level.

When the LDPTE instruction is executed, it accesses the PTE-level page table based on the currently processed TLB refill address, retrieves the page table entry, and writes it to the corresponding address.

CSR.

If bit [6] of the general-purpose register rj is 1, it indicates that the content of rj is a HugePage page table entry. In this case,

Bits [14:13] of the general-purpose register rj should be a non-zero value 'n', indicating that it is a large page table entry corresponding to the nth level page table. Set the general-purpose register rj...

The values in the table are converted into the final page table entry format and then written into the corresponding CSR.

4.2.6 Other Miscellaneous Instructions

4.2.6.1ERTN

Command format: ertn

The ERTN instruction is used to return from exception handling.

If the exception being handled is a Debug exception, clear the DS bit in CSR.DBG and jump to the address stored in CSR.DERA.

The finger is taken from the starting point.

If the exception being handled is not a debug exception, update the corresponding PPLV, PIE, PWE, and other information to [the relevant information].

In CSR.CRMD, the program simultaneously jumps to the return address corresponding to the exception to begin fetching instructions.

If the exception being handled is an Error class exception, the corresponding PPLV, PIE, and PWE information comes from CSR.MERRCTL.

The corresponding return address comes from CSR.MERRERA. In addition, the Error class exception also requires the PDA, PPG, and other files in CSR.MERRCTL to be returned.

PDCAF and PDCAM information are updated in CSR.CRMD.

If the exception being processed is a TLB refill exception, the corresponding PPLV, PIE, and PWE information comes from CSR.TLBRSAVE, for example...

The corresponding return address comes from CSR.TLBRERA. In addition, the DA bit in CSR.CRMD must be cleared to 0 and the PG bit set to 1.

If the exception being handled is not a Debug exception, an Error class exception, or a TLB refill exception, then the corresponding PPLV, PIE, and PWE...

The information comes from CSR.PRMD, and the return address corresponding to the exception comes from CSR.ERA.

When executing the ERTN instruction, if the KLO bit in CSR.LLBCTL is not equal to 1, then LLbit is set to 0; otherwise, LLbit is not modified.

4.2.6.2DBCL

Command format: dbcl code

Executing the DBCL command will immediately enter debug mode.

4.2.6.3IDLE

Command format: idle level

After executing the IDLE instruction, the processor core will stop fetching instructions and enter a wait state until it is woken up by an interrupt or reset.

After a pause, the first instruction executed by the processor core is the one following IDLE.

5 Storage Management

5.1 Physical Address Space

The physical address space range of memory is 0~ 2PALEN-1.

In the LA32 architecture, PALEN is theoretically a positive integer not exceeding 36, with its specific value determined by the implementation, but 32 is usually recommended.

In the LA64 architecture, PALEN is theoretically a positive integer not exceeding 60, with its specific value determined by the implementation.

The system software can determine the specific value of PALEN by reading the PALEN field of configuration word 0x1 through CPUCFG.

5.2 Virtual Address Space and Address Translation Mode

In the Dragon architecture, the virtual address space is linearly flat. For PLV0 level, the virtual address space size under the LA32 architecture is 2^ 32 bytes.

The virtual address space in the LA64 architecture is 2 ^64 bytes. However, for the LA64 architecture, not all 2 ^64 bytes of virtual address space are...

It is legal, and it can be assumed that some virtual address gaps exist. A legal virtual address space is closely related to the address mapping pattern, which will be discussed later.

This will be introduced in conjunction with the definition of address mapping modes.

The Dragon architecture's MMU supports two virtual and physical address translation modes: direct address translation mode and mapped address translation mode.

When DA=1 and PG=0 in CSR.CRMD, the processor core's MMU is in direct address translation mode. In this mapping mode,

The physical address is by default directly equal to the [PALEN-1:0] bits of the virtual address (padded with 0s if necessary), unless a higher priority is used in the specific implementation.

Virtual and physical address translation rules. As you can see, the entire virtual address space is valid at this point. After the processor resets, it will enter the direct address translation phase.

model.

When DA=0 and PG=1 in CSR.CRMD, the processor core's MMU is in mapped address translation mode. This is further divided into direct mapping...

There are two types of address translation mode (referred to as "direct mapping mode") and page table mapping address translation mode (referred to as "page table mapping mode"). Translation

When translating addresses, the system will first check if they can be translated using the direct mapping mode; if not, it will then proceed with the page table mapping mode. (Regarding direct mapping...)

For a detailed explanation of mapping modes, please refer to Section 5.2.1; for a detailed explanation of page table mapping modes, please refer to Section 5.4. This section focuses on LA64.

Under this architecture, when using page table mapping mode, the rule for determining the validity of the virtual address space is: the [63:VALEN] bits of a valid virtual address must be equal to...

The [VALEN-1] bits must be identical, meaning all bits above [VALEN-1] are its sign extension; otherwise, an Address Error (ADE) exception will be triggered. However, in direct...

In the mapping mode, there is no need to perform this address invalidity check.

5.2.1 Direct Mapping Address Translation Mode

When the processor core's MMU is in mapped address mode, direct mapping between virtual and physical addresses can also be achieved through the direct mapping configuration window mechanism.

The direct mapping configuration window has four windows. The first two windows can be used for both instruction fetching and load/store operations simultaneously, while the last two windows are only used for...

Load/store operations.

The system software configures four direct mapping configuration windows by configuring the CSR.DMW0~CSR.DMW3 registers. Each window, in addition to...

In addition to the address range information, you can also configure under which privilege levels this window is available, and the storage of memory access operations where virtual addresses fall on this window.

Access type.

Under the LA64 architecture, each direct mapping configuration window can be configured with a fixed-size virtual address space of 2 PALEN bytes. When the virtual address space...

When a physical address hits a valid directly mapped configuration window, its physical address is directly equal to the [PALEN-1:0] bits of the virtual address. The hit determination method is:

The highest 4 bits of the virtual address (bits [63:60]) are equal to the VSEG field in the configuration window register, and the current privilege level is allowed in this configuration window.

May.

For example, with PALEN equal to 48, by configuring DMW0 to 0x9000000000000011, then at PLV0 level...

The virtual address space from 0x900000000000000 to 0x9000FFFFFFFFFFFF, which was originally invalid in page mapping mode, will be mapped.

It is mapped to the physical address space 0x0 ~ 0xFFFFFFFFFFFF, and the storage access type is consistent and cacheable.

In the LA32 architecture, each direct-mapped configuration window can be configured with a fixed-size virtual address space of 229 bytes. When the virtual address...

When a valid direct mapping configuration window is hit, its physical address is directly equal to the [28:0] bits of the virtual address, appended with the physical address configured in that mapping window.

The high-order bits of the virtual address. The hit detection method is: the highest 3 bits of the virtual address (bits [31:29]) are equal to [31:29] in the configuration window register, and when...

The previous privilege level is allowed in this configuration window.

For example, by configuring DMW0 to 0x80000011, then at PLV0 level, the range 0x80000000 ~ 0x9FFFFFFF...

The address will be directly mapped to the physical address space 0x0 ~ 0x1FFFFFFF, and its storage access type is consistent and cacheable.

5.2.2 32 -bit address mode under the LA64 architecture

When application software binaries based on the LA32 architecture are run on processors implementing the LA64 architecture, in order to obtain the same results, it is necessary to...

Special handling is required for address calculations within instructions; this is the 32-bit address mode control unique to the LA64 architecture. When CSR.MISC...

When VA32L1/VA32L2/VA32L3 is set to 1, software running at PLV1/PLV2/PLV3 levels will run in 32-bit address mode.

At this point, the hardware will use the value after zero-extending the lower 32 bits of the virtual address of the memory access (including instruction fetch memory access) to 64 bits as the virtual address of the memory access, and BL and JIRL...

Instructions like PCADD will also sign-extend the lower 32 bits of the result to 64 bits before writing it back to the result register.

5.2.3 Virtual Address Reduction Mode under LA64 Architecture

To reduce page table levels in certain applications, the LA64 architecture also provides a virtual address reduction mode. When system software...

When the RDVA in the CSR.RVACFG register is configured to a value between 1 and 8, the valid bits of the virtual address in the mapped address translation mode will be determined according to...

(VALEN-RDVA) requires so many bits for processing. For example, on a processor with VALEN=48, when RDVA is configured to 8, the valid address...

The [63:40]th bit needs to be the sign extension of the [39]th bit.

5.3 Storage Access Types

As mentioned in Section 2.1.7 above, the Dragon architecture supports three storage access types: Coherent Cached (or simply Cached).

CC, Strongly-ordered Uncached (SUC), and Weakly-ordered Uncached (WC)

(referred to as WUC).

When the processor core MMU is in direct address translation mode, all memory access types for instruction fetching are determined by CSR.CRMD.DATF.

The storage access type for load/store operations is determined by the CSR.CRMD.DATM domain.

When the processor core MMU is in mapped address translation mode, the determination of the memory access type falls into two categories. If it's instruction fetch or load/store...

If the address of the operation falls on a direct mapping configuration window, then the storage access type of the fetch or load/store operation is determined by the configuration of that window.

The MAT field in the CSR register determines the memory access type. If instruction fetching or load/store can only be mapped through the page table, then the memory access type is determined by the page table.

The MAT field in the item determines this.

Regardless of the specific circumstances, the definition of the storage access type control value remains the same: 0 – strong order, non-cached; 1 – consistent, cached.

Store, 2 - weak order, no cache, 3 - keep.

5.4 Page Table Mapping Storage Management

In mapped address translation mode, all valid addresses, except those falling within the direct mapping configuration window, must be translated through the page table.

The mapping completes the virtual-to-physical address translation. The TLB, acting as a temporary cache in the processor storing operating system page table information, is used to accelerate the address mapping process.

The virtual-to-physical address translation process for instruction fetch and load/store operations in translation mode.

5.4.1 TLB Organizational Structure

In the Dragon architecture, the TLB is divided into two parts: a single-page-size TLB where all entries have the same page size.

The other is the Multiple-Page-Size TLB (TLB), which supports different page sizes for different entries.

MTLB).

Whether a page table entry with the same page size as the STLB can enter the MTLB is determined by the implementation and is not restricted by the architecture specification.

During the virtual-to-physical address translation process, both the STLB and MTLB are looked up simultaneously. Accordingly, the software must ensure that the MTLB and STLB are not accessed concurrently.

The hit occurs when the processor is active; otherwise, the processor's behavior will be unpredictable.

MTLB uses a fully associative lookup table organization, while STLB uses a multi-way set-associative organization. For STLB, if it has 2 INDEX...

If the group is configured with a page size of 2 PS bytes, then during the hardware STLB lookup process, the [PS+INDEX:PS+1] bits of the virtual address are used as the index.

Use reference values to access various information.

5.4.2 TLB Entries

The table entry formats of STLB and MTLB are basically the same, the only difference being that each entry in MTLB includes page size information, while STLB...

Since it's the same page size, the TLB entry doesn't need to store the page size information again. For STLB, the page size of the page table entry it stores...

It is configured in the PS field of the CSR.STLBPS register by the system software.

The format of each TLB entry is shown in Figure 5-1, which contains two parts: a comparison part and a physical conversion part.

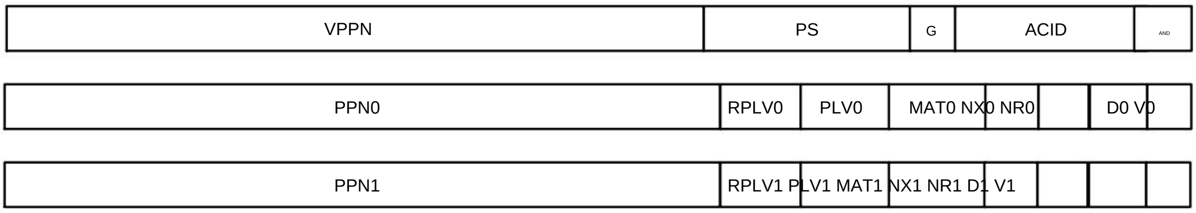

Figure 5-1 **TLB** Entry Format

The comparison section of the TLB entries includes:

ÿ Existence bit (E), 1 bit. A value of 1 indicates that the corresponding TLB entry is not empty and can participate in the search and matching.

ÿ Address Space Identifier (ASID), 10 bits. The address space identifier is used to distinguish the same virtual address in different processes, avoiding process switching.

The performance penalty of clearing the entire TLB during swapping. The operating system assigns a unique ASID to each process, and the TLB performs lookups...

In addition to verifying that the address information matches, it is also necessary to verify the ASID information.

ÿ Global Flag (G), 1 bit. When this bit is 1, no ASID consistency check is performed during the lookup. This is necessary when the operating system requires...

When all processes share the same virtual address, the G bit in the TLB page table entry can be set to 1.

ÿ Page Size (PS), 6 bits. Appears only in MTLB. Used to specify the page size stored in this page table entry. The value is 2 times the page size.

The power of the power. That is, for a 16KB page, PS=14.

ÿ Virtual Double Page Number (VPPN), (VALEN-13) bits. In the Dragon architecture, each page table entry stores a pair of adjacent odd-even adjacent page tables.

Therefore, the virtual page number stored in the TLB page table entry is the virtual page number in the system divided by 2, meaning the least significant bit of the virtual page number does not need to be stored.

In the TLB, when searching the TLB, the least significant bit of the virtual page number being searched determines whether to select an odd-numbered or even-numbered page for physical transfer.

Change information.

The physical translation section of the table entry stores the physical translation information for a pair of odd-even adjacent page tables. The translation information for each page includes:

ÿ Valid bit (V), 1 bit. A value of 1 indicates that the page table entry is valid and has been accessed.

ÿ Dirty bit (D), 1 bit. A value of 1 indicates that there is dirty data in the address range corresponding to this page table entry.

ÿ Unreadable bit (NR), 1 bit. A value of 1 indicates that a load operation is not allowed in the address space containing this page table entry. This control bit is only used to specify the access level.

It is based on the LA64 architecture.

ÿ Non-executable bit (NX), 1 bit. A value of 1 indicates that instruction fetching is not allowed in the address space containing this page table entry. This control bit only determines...

It is based on the LA64 architecture.

ÿ Memory Access Type (MAT), 2 bits. Controls the memory access type of memory access operations falling within the address space of this page table entry. (Each number...)

For the specific meaning of the value, see Section 5.3 .

ÿ Privilege Level (PLV), 2 bits. The privilege level corresponding to this page table entry. When RPLV=0, this page table entry can be accessed by any privileged user.

Programs with a privilege level of at least PLV can access this page table entry; when RPLV=1, this page table entry can only be accessed by programs with a privilege level equal to PLV.

ÿ Restricted Privilege Level Enable (RPLV), 1 bit. A control bit indicating whether a page table entry is accessed only by programs of the corresponding privilege level. Please refer to [link/reference].

See the contents of PLV above. This control bit is only defined in the LA64 architecture.

ÿ Physical Page Number (PPN), (PALEN-12) bits. When the page size is greater than 4KB, the PPN stored in the TLB is [PS-1:12] bits.

The bit can be any value.

5.4.3 TLB Software Management

In the Dragon architecture, TLB management involves software aspects. In version 1.0x of this architecture specification, TLB refilling and the relationship between TLB and memory pages...

Maintaining consistency between tables is still entirely handled by the software.

5.4.3.1 TLB -related exceptions

The TLB performs virtual-to-physical address translation automatically in hardware. However, this can happen when there is no matching entry in the TLB, or when a match is found but the page table entry is invalid.

If access is illegal, an exception needs to be triggered, handing the matter over to the operating system kernel or other monitoring programs for further processing by the software, including the contents of the TLB.

To perform maintenance or make a final ruling on the legality of program execution. Exceptions related to TLB management in the Dragon architecture include:

ÿ TLB Refill Exception: This exception is triggered when no match is found in the TLB for the virtual address accessed in the memory access operation, notifying the system software to proceed.

The TLB is refilled. This exception has its own exception entry point, a separate CSR for maintaining the exception context, and a separate set of procedures.

The TLB access interface CSR means that this exception can be triggered during the handling of other exceptions. The TLB refill exception trap is the same...

When this happens, the hardware will automatically set DA to 1 and PG to 0 in CSR.CRMD, thus automatically entering direct address translation mode and avoiding...

If the exception handler that was previously exempt from TLB re-filling triggers a TLB re-filling exception again, the exception context will not be saved or restored. This is to prevent...

The TLB refills the CSR used after an exception trap and the CSRs available for other exceptions. Simultaneously with the TLB refilling of the exception trap, the hardware...

It will also automatically set CSR.TLBRERA.ISTLBR to position 1.

ÿ Load operation page invalid exception: If a match is found in the TLB for the virtual address of the load operation, but the matching page table entry has V=0, this will trigger an exception.

This is an exception.

ÿ Store operation page invalid exception: If a match is found in the TLB for the virtual address of the store operation, but the V=0 of the matching page table entry, this will trigger an exception.

This is an exception.

ÿ Instruction fetch page invalid exception: If a match is found in the TLB for the virtual address of the instruction fetch operation, but the matching page table entry has V=0, this will trigger an exception.

This is an exception.

ÿ Page privilege level non-compliance exception: The virtual address of the memory access operation found a matching entry with V=1 in the TLB, but the access privilege level...

This exception will be triggered if the privilege level is non-compliant. Non-compliance is manifested as RPLV=0 and CSR.CRMD.PLV value being large for the page table entry.

The PLV in the page table entry; or the RPLV of the page table entry is 1 and CSR.CRMD.PLV is not equal to the PLV in the page table entry.

ÿ Page Modification Exception: The virtual address of the store operation is matched in the TLB, V=1, and the privilege level is compliant, but the page...

This exception will be triggered if the D bit of the entry is 0.

ÿ Page Unreadable Exception: The virtual address of the load operation is found to match in the TLB, V=1, and the privilege level is compliant, but the...

This exception will be triggered if the NR bit of a page table entry is 1.

ÿ Page Non-Executable Exception: A matching entry was found in the TLB for the virtual address of the instruction fetch operation, where V=1 and the privilege level is compliant, but...

This exception will be triggered if the NX bit of this page table entry is 1.

5.4.3.2 TLB- related instructions

TLB-related instructions mainly involve operations such as searching, reading, writing, and invalidating the TLB, and are used for TLB filling, updating, and consistency.

Maintenance. For specific instruction definitions, please refer to sections 4.2.4 and 4.2.5 of this manual.

5.4.3.3 TLB- related CSR

TLB-related CSRs are mainly divided into three categories according to their functions. The first category is for the interaction interface of the TLB in non-TLB refill exception cases.

The first class is used for traversing hardware and software page tables, and the third class is used for TLB refill exceptions.

The first category includes:

ÿ BADV

ÿ OBSERVATION

ÿ EXPECTATION0

ÿ TLBELO1

ÿ TLBIDX

ÿ ACID

ÿ STLBPS

The second category includes:

ÿ PGDL

ÿ PGDH

ÿ PGD

ÿ PWCL

ÿ PWCH

The third category includes:

ÿ TLBRENTRY

ÿ TLBRERA

ÿ TLBRBADV

ÿ TLBREHI

ÿ TLBRELO0

ÿ TLBRELO1

ÿ TLBRPRMD

ÿ TLBRSAVE

For details on the interaction between the above CSR registers and the TLB, please refer to the detailed definitions of each CSR in Section 7.4 .

5.4.3.4 TLB Initialization

The Dragon architecture allows for hardware initialization of the TLB without implementation, allowing the startup software to accomplish this function by executing "INVTLB 0, r0, r0".

5.4.4 TLB -based virtual-physical address translation process

This section describes the virtual-to-physical address translation process based on the TLB. The following description uses pseudocode to illustrate the process of first checking the STLB and then the MTLB.

The process described here is for ease of description only; in the processor hardware implementation, both STLB and MTLB can be searched simultaneously.

```
# va: Virtual address to be found
# mem_type: Memory access operation type, FETCH is instruction fetch, LOAD is load, STORE is store # plv: Current privilege level,
i.e., the value of CSR.CRMD.PLV # pa: Translated physical address
# mat: Translated memory access
type # VALEN: Effective number of bits in the
virtual address # PALEN: Effective number
of bits in the physical address # STLB[][]:
STLB[N][M] represents the Mth item of the Nth path in the STLB #
STLB_WAY: Number of paths in the
STLB # STLB_INDEX: The number of groups in each path of the STLB raised to the power of 2, i.e., each path
has 2STLB_INDEX groups # MTLB[]: MTLB[N] represents the
Nth item in the MTLB # MTLB_ENTRIES: Number of items in the MTLB


# Find STLB
stlb_found = 0
stlb_ps = CSR.STLBPS.PS
stlb_idx = and[stlb_ps+STLB_INDEX:stlb_ps+1]
for way in range(STLB_WAY)ÿ
    if (STLB[way][stlb_idx].E==1) and
      ((STLB[way][stlb_idx].G==1) or (STLB[way][stlb_idx].ASID==CSR.ASID.ASID))
 and
      (STLB[way][stlb_idx].VPPN[VALEN-1:stlb_ps+1]==va[VALEN-1:stlb_ps+1]) :
        if (stlb_found==0) :
            stlb_found = 1
            if (va[stlb_ps]==0) :
                sfound_v = STLB[way][stlb_idx].V0
                sfound_d = STLB[way][stlb_idx].D0
                sfound_nr = STLB[way][stlb_idx].NR0
                sfound_nx = STLB[way][stlb_idx].NX0
                sfound_mat = STLB[way][stlb_idx].MAT0
                sfound_plv = STLB[way][stlb_idx].PLV0
                sfound_rplv = STLB[way][stlb_idx].RPLV0
                sfound_ppn = STLB[way][stlb_idx].PPN0
            else :
                sfound_v = STLB[way][stlb_idx].V1
                sfound_d = STLB[way][stlb_idx].D1
                sfound_nr = STLB[way][stlb_idx].NR1
                sfound_nx = STLB[way][stlb_idx].NX1
                sfound_mat = STLB[way][stlb_idx].MAT1
                sfound_plv = STLB[way][stlb_idx].PLV1
                sfound_rplv = STLB[way][stlb_idx].RPLV1
                sfound_ppn = STLB[way][stlb_idx].PPN1
        else :
            # Multiple hits occurred, and the processor's execution result is uncertain.
# Find MTLB
mtlb_found = 0
for i in range(MTLB_ENTRIES) :
    if (MTLB[i].E==1) and
      ((MTLB[i].G==1) or (MTLB[i].ACID==CSR.ACID.ACID)) and
      (MTLB[i].VPPN[VALEN-1:MTLB[i].PS+1]==va[VALEN-1: MTLB[i].PS+1]) :
        if (mtlb_found==0) :
            mtlb_found = 1
            mfound_ps = MTLB[i].PS
            if (va[mfound_ps]==0) :
                mfound_v = MTLB[i].V0
                mfound_d = MTLB[i].D0
                mfound_nr = MTLB[i].NR0
                mfound_nx = MTLB[i].NX0
                mfound_mat = MTLB[i].MAT0
                mfound_plv = MTLB[i].PLV0
                mfound_rplv = MTLB[i].RPLV0
                mfound_ppn = MTLB[i].PPN0
            else :
                mfound_v = MTLB[i].V1
                mfound_d = MTLB[i].D1
                mfound_nr = MTLB[i].NR1
                mfound_nx = MTLB[i].NX1
                mfound_mat = MTLB[i].MAT1
                mfound_plv = MTLB[i].PLV1
                mfound_rplv = MTLB[i].RPLV1
                mfound_ppn = MTLB[i].PPN1

        else: # Multiple hits occurred, and the processor's execution result is uncertain.

if (stlb_found==1) and (mtlb_found==1) :
    #Multiple hits occurred, and the processor's execution result is uncertain.
elif (stlb_found==1) :
    found_v = sfound_v
    found_d = sfound_d
    found_nr = sfound_nr
    found_nx = sfound_nx
    found_mat = sfound_mat
    found_plv = sfound_plv
    found_rplv = sfound_rplv
    found_ppn = sfound_ppn
    found_ps = stlb_ps
```

```
elif (mtlb_found==1):
        found_v = mfound_v
        found_d = mfound_d
        found_nr = mfound_nr
        found_nx = mfound_nx
        found_mat = mfound_mat
        found_plv = mfound_plv
        found_rplv = mfound_rplv
        found_ppn = mfound_ppn
        found_ps = mfound_ps
else :
        SignalException(TLBR)                    #Report TLB re-entry exception

if (found_v==0) :
        case mem_type :
                FETCH : SignalException(PIF) # Reports invalid fetch operation page # Reports
                LOAD : SignalException(PIL)              invalid load operation page #
                STORE : SignalException(PIS)             Reports invalid store operation page
elif (mem_type==FETCH) and (found_nx==1) :
        SignalException(PNX)                     #Exception that the page cannot be executed
elif ((found_rplv==0) and (plv > found_plv)) or
            ((found_rplv==1) and (plv!= found_plv)) :
        SignalException(PPI)                     #Exceptions to non-compliant newspaper page privilege levels
elif (mem_type==LOAD) and (found_nr==1) :
        SignalException(PNR)                     #Unreadable page exception
elif (mem_type==STORE) and (found_d==0)
            and ((plv==3) or (CSR.MISC[16+plv]==0)): #Write is disabled, check function is not enabled#Page
        SignalException(PME)                     modification exception
else :
        pa = {found_ppn[PALEN-13:found_ps-12], va[found_ps-1:0]}
        mat = found_mat
ÿ
```

5.4.5 Multi-level page table structures supported by the page table traversal process

Whether it's software page table traversal implemented using LDDIR and LDPTE instructions or hardware page table traversal, it supports multi-level page table structures.

They are the same, as shown in Figure 5-2.

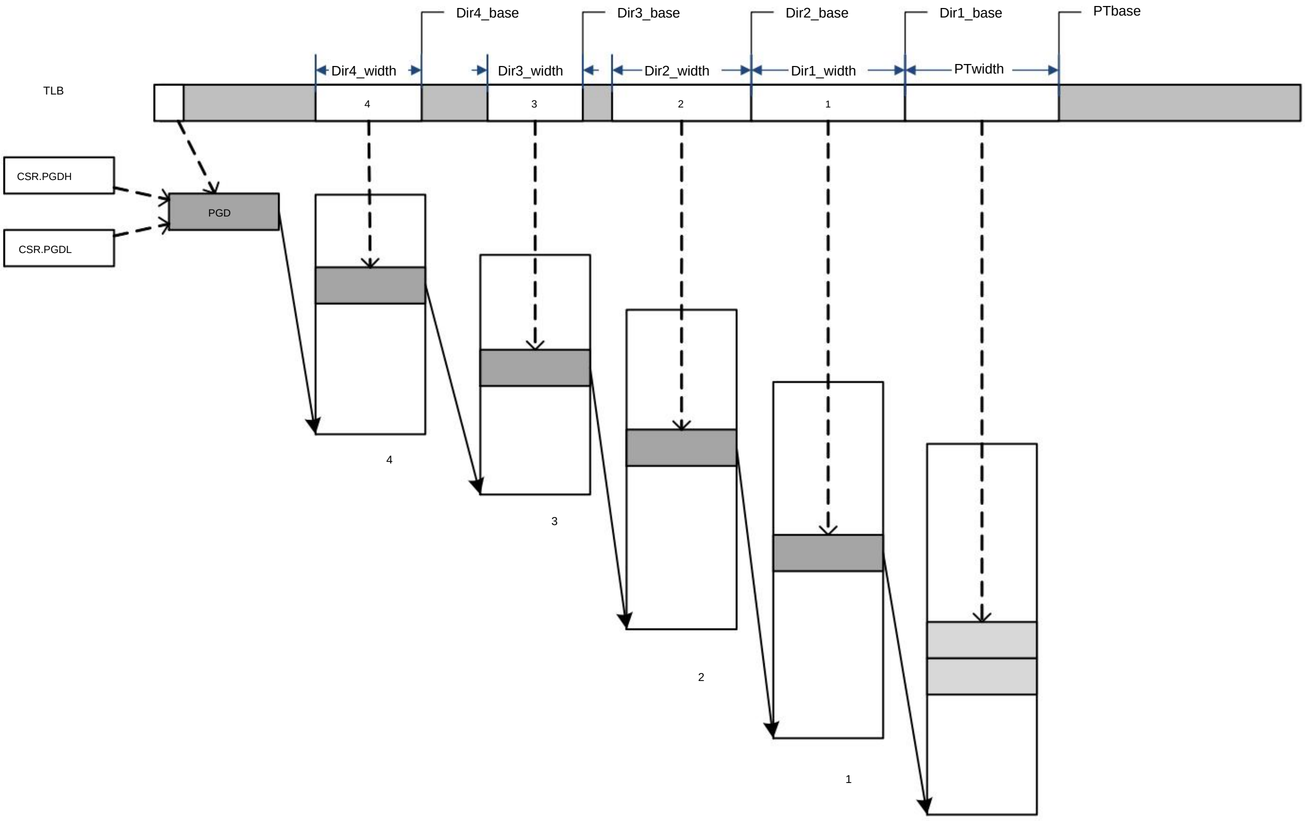

Figure 5-2 shows the multi-level page table structures supported by the page table traversal process.

The base address PGD of the top-level directory (Global Directory) of the page table being traversed needs to be determined based on the (VALEN-1)th digit of the virtual address being queried.

This is determined by a specific bit. When this bit is 0, PGD comes from CSR.PGDL; when this bit is 1, PGD comes from CSR.PGDH. This means that the entire...

The page table structure is (VALEN-1) bits.

The specifications of directory entries and page table entries at all levels are configured by the system software in CSR.PWCL and CSR.PWCH.

Whether page table traversal is performed using LDDIR and LDPTE instructions or hardware, the system software needs to follow these guidelines:

The format defines page table entries.

Basic page table item format:

63	62 61	POLES-1		12	8 7 6 5 4 3 2 1 0
RPLV NX NR		PA[PALEN-1:12]			WPG MAT PLV DV

Large page table item format:

63	62 61	POLES-1	log2PageSize		12		8 7 6 5 4 3 2 1 0
RPLV NX NR		NOT[PALEN-1:log2PageSize]			G		W P H MAT PLV D V

In the above definition of page table entry format, the main difference in format between page table entries for large pages and page table entries for basic pages is: (1) the first page table entry of the table of contents...

6 bits are the large page table entry flag H, and a value of 1 indicates that the directory entry at this time actually stores the page table entry information of a large page; (2) Basic page table

The G position for the item is in position 6, while the G position for the large page table item is in position 12.

Bits not explicitly defined in the above two formats will be automatically ignored by the LDDIR, LDPTE instructions or hardware page table traversal logic.

In the page table entry format described above, the "P" and "W" fields represent whether the physical page exists and whether the page is writable, respectively. These information...

Although the information is not filled into the TLB table entries, it is used in the page table traversal process.

Because the TLB entries use a double-page storage structure, for large-page page table entries (which have only one entry), the hardware page table refill logic or software...

The LDPTE instruction for a component will automatically split the page table entry information of the large page into two half-sized page table entries and fill them into the TLB. For example, the standard

If the page size is 16KB, then the size of the first-level big page is typically 32MB. After the software page table traversal process completes "LDPTE rj, 0" and "LDPTE

After the "rj, 1" instruction, two page table entries of 16MB each will be filled into the TLB, and no special software intervention is required.

Because address mapping is in direct address translation mode during TLB refill exception (TLBR) processing, PGD and memory...

The address configured in the directory entry of the page table must be a physical address.

6 Exceptions and Interruptions

Line 6.1 interrupted

The Dragon architecture supports two types of interrupts: line interrupts and message interrupts. Line interrupts are mandatory, while message interrupts are optional.

The implemented interrupt is an extension of the online interrupt. This section will introduce the specifications of online interrupts.

6.1.1 Line Interruption Types

Under the Dragon architecture, each processor core can record 13 line interrupts: 1 inter-core interrupt (IPI) and 1 timer interrupt (TI).

1 Performance Monitor Count Overflow Interrupt (PMI), 8 hard interrupts (HWI0~HWI7), and 2 soft interrupts (SWI0~SWI1). All lines

All interrupts are level interrupts, and all are active high.

Inter-core interrupts are input from the external interrupt controller and are sampled and recorded by the processor core in the CSR.ESTAT.IS[12] bit.

The timer interrupt originates from the internal constant-frequency timer. This interrupt is triggered when the constant-frequency timer counts down to all zeros.

Enabled. Once enabled, the timer interrupt is sampled and recorded by the processor core in the CSR.ESTAT.IS bit[11]. Clearing the timer interrupt requires software intervention.

This is accomplished by writing 1 to the TI bit of the CSR.TICLR register.

The performance counter overflow interrupt originates from the performance counter within the kernel. When any interrupt enables the performance counter, the counting process resumes.

When bit [63] of the value is 1, the interrupt will be enabled. The enabled performance counter overflow interrupt is sampled and recorded by the processor core.

CSR.ESTAT.IS[10] bit. Clearing a performance counter overflow interrupt requires setting bit[63] of the performance counter that caused the interrupt to 0, or

Disable the interrupt enable for this performance counter.

The interrupt source for hardware interrupts originates outside the processor core, typically from an external interrupt controller. The eight hardware interrupts HWI[7:0] are...

The processor core sampling record is in bits CSR.ESTAT.IS[9:2].

The interrupt source for a software interrupt originates from within the processor core. Software enables a software interrupt by writing 1 to CSR.ESTAT.IS[1:0] using the CSR instruction.

0 clears the soft interrupt.

The index value of the location of the interrupt recorded in the CSR.ESTAT.IS field is also called the interrupt number (Int Number). The interrupt number for SWI0 is equal to 0.

The interrupt number for SWI1 is 1, ..., and the interrupt number for IPI is 12.

6.1.2 Line interrupt priority

The response to multiple interrupts simultaneously employs a fixed-priority arbitration mechanism, with higher interrupt numbers having higher priority. Therefore, IPI has the highest priority.

TI is next, ..., SWI0 has the lowest priority.

6.1.3 Line Interruption Entry

Once an interrupt is marked as an instruction by the processor hardware, it is treated as an exception; therefore, the calculation of the interrupt entry point follows the same rules as ordinary exceptions.

Calculation rules for entry points. For calculation rules regarding ordinary exception entry points, please refer to Section 6.2.1 . It should be noted that, in calculating the entry point location...

When addressing an interrupt, the exception number corresponding to that interrupt is its own interrupt number plus 64. That is, the exception number corresponding to interrupt 0 (SWI0) is 64, and the exception number corresponding to interrupt 1 (SWI1) is 64.

The corresponding exception numbers are 65, ..., and so on.

6.1.4 Processing procedure for interrupts on the processor hardware response line

Interrupt signals from each interrupt source are sampled by the processor and stored in the CSR.ESTAT.IS field. This information, along with software configuration, is stored in CSR.ECFG.LIE.

The local interrupt enable information in the domain is bitwise ANDed to obtain a 13-bit interrupt vector int_vec. This is achieved when CSR.CRMD.IE = 1 and int_vec is not all zeros.

When the processor determines that there is an interrupt that needs to be responded to, it selects an instruction from the executed instruction stream and marks it as a special exception.

— Interruption exception.

The subsequent processing by the processor hardware is the same as that for ordinary exceptions; please refer to the description in Section 6.2.3 .

6.2 Message Interruption

6.2.1 Message Interruption Types

In the Dragon architecture, each logical processor core can record 256 message interrupts internally, including message-type inter-core interrupts input from outside the processor core.

Hardware interrupts include both breakpoint and message-based interrupts. Each processor core internally has four 64-bit CSRs (CSR.MSGIS0~CSR.MSGIS3) that sequentially record values from 0 to 255.

Whether message interrupts are triggered. The specific sources of the 256 message interrupts within each processor core are determined by the implementation, and software developers need to consult [the relevant documentation].

Refer to the specific chip user manual for relevant information.

6.2.2 Message Interruption Priority

In the Dragon architecture, each logical processor core has 256 message interrupts with a fixed priority. The higher the interrupt number, the higher the priority; for example, the 255th interrupt has the highest priority.

Message No. 1 has the highest interrupt priority, followed by Message No. 254, and so on, with Message No. 0 having the lowest interrupt priority.

Each logic processor core internally records message interrupts only if their priority is not lower than the message interrupt enable priority threshold (recorded in...).

Only when the CSR.MSGIE.PT domain is reached can the hardware further select and initiate a message interrupt request.

When a processor core has both a message interrupt request and a line interrupt request triggered simultaneously, the message interrupt request has higher priority than the line interrupt request.

ask.

6.2.3 Message Interruption Entry Point

All message interrupts use a unified entry point, and the "entry page number" used to calculate the entry address is the same as that of line interrupts, originating from CSR.EENTRY.

Its calculated entry address “page offset” is equal to 2 (CSR.ECFG.VS+2)×78 (0x4E).

6.2.4 Message Interruption Response Process

Once a message interrupt is routed to the designated processor core, that core will internally process CSR.MSGIS0~CSR.MSGIS3 according to its interrupt number.

The corresponding status position is 1, and this process is for recording message interrupts. Subsequently, the processor core records interrupts with interrupt numbers no lower than message interrupt enable priority.

From the message interrupts with the highest priority (recorded in the CSR.MSGIE.PT field), select the one with the highest priority and record its message interrupt number.

In the CSR.MSGIR.IntNum field, simultaneously clear the CSR.MSGIR.Null bit to 0 and set the CSR.ESTAT.MsgInt bit to 1. This process interrupts the message.

The selection and triggering of message interrupt requests. When the CSR.ESTAT.MsgInt bit is 1, CSR.CRMD.IE masking can only be enabled via global interrupts.

The selected and triggered message interruption request.

When the CSR.ESTAT.MsgInt bit is 1, if the software reads the CSR.MSGIR register, the hardware will automatically adjust the current record accordingly.

For the message interruption number in the CSR.MSGIR.IntNum field, clear the corresponding status bits of CSR.MSGIS0~CSR.MSGIS3 to 0. If the message interruption status...

After the status bit is cleared to 0, if there are no more selectable message interrupts in CSR.MSGIS0~CSR.MSGIS3, then in the next processor core internal clock cycle...

The CSR.ESTAT.MsgInt bit will be cleared to 0 by hardware, while the CSR.MSGIR.Null bit will be set to 1. Software developers are especially advised to take note, because...

The CSR.MSGIR register has a "read clear" feature, so it is recommended to read CSR.ESTAT.MsgInt when you need to check for pending message interrupts.

Bit.

6.3 Exceptions

6.3.1 Exception Entrance

The entry point for TLB refill exceptions comes from CSR.TLBRENTRY.

The entry point for the machine error exception comes from CSR.MERRENTRY.

Exceptions other than the two types mentioned above are called ordinary exceptions, and their entry addresses are calculated using the formula "entry page number | offset within page".

The "|" operator is used for bitwise OR operations.

All ordinary exception entries have the same entry page number, and all originate from CSR.EENTRY.

The offset of a normal exception entry is determined by both CSR.ECFG.VS and the exception number (ecode). When CSR.ECFG.VS = 0, all normal exception entries...

When the exception entry offsets are the same, the software needs to determine the specific exception type using the Ecode and IS fields in CSR.ESTAT.

When CSR.ECFG.VS != 0, the entry offset for each normal exception is equal to 2 (CSR.ECFG.VS+2)×ecode, and the software does not need to access CSR.ESTAT.

To confirm the exception type, see the Ecode column in Table 7-8 for the ecode values of ordinary exceptions, excluding interrupts; the ecode of an interrupt is the interrupt number plus 64.

Since the exception entry is an offset value obtained by bitwise ORing the entry page number, when CSR.ECFG.VS != 0, the software allocates the exception entry base address.

It is necessary to ensure that all possible offset values do not exceed the boundary alignment space corresponding to the low bit of the entry base address.

6.3.2 Exception Priority

Exception priority follows two basic principles: first, interrupts have higher priority than exceptions; second, for exceptions, those detected during the instruction fetch phase have higher priority.

The highest priority is detected during the decoding stage, followed by the next highest priority, and then the lowest priority is detected during the execution stage.

For exceptions detected during the instruction fetch phase: fetch Watch exceptions have the highest priority, followed by fetch address errors, and then fetch TLB exceptions.

The priority of the off exception is next, and the priority of the machine error exception is the lowest.

The exceptions that can be detected during the decoding phase are mutually exclusive, so there is no need to consider their priority.

During the execution phase, if only memory access instructions are executed or multiple exceptions are triggered simultaneously, their priorities from highest to lowest are: Address Error Exception (ADE) > Address Request Exception ...

Address alignment error exception (ALE) caused by address misalignment in memory access instructions > Boundary constraint check exception 1 (BCE) > TLB related

Exception 2 > Address alignment errors caused by memory access instructions that allow unaligned addresses spanning two pages of different memory access types.

(ALE). However, there is a detail that needs special explanation: for memory access instructions that require address alignment, if the address is not aligned...

If the instruction happens to cross two access regions with different attributes, then if the instruction satisfies the ADE exception condition within the access region containing the lower address...

If the condition is met, then the instruction triggers an ADE exception. However, if the instruction only meets the ADE exception criteria within the access region containing the higher address, then...

This instruction then triggers an ALE exception instead of an ADE exception.

6.3.3 General Process for Handling Common Exceptions in Hardware

Different common exceptions may have some subtle differences in how the processor hardware handles them. Here, we discuss the general principles common to all common exceptions.

The processing procedure is described.

When a normal exception is triggered, the processor hardware performs the following operations:

ÿ Store the PLV and IE of CSR.CRMD into the PPLV and PIE of CSR.PRMD respectively, and then set the PLV of CSR.CRMD to

If the value is 0, IE will set it to 0;

ÿ For implementations supporting the Watch function, the WE of CSR.CRMD should be stored in the PWE of CSR.PRMD, and then...

The WE property of CSR.CRMD is set to 0;

ÿ Record the PC value that triggered the exception instruction in CSR.ERA;

ÿ Jump to the exception entry point to retrieve the pointer.

When the software executes the ERTN instruction and returns from a normal exception execution, the processor hardware performs the following operations:

ÿ Restore the PPLV and PIE values from CSR.PRMD to the PLV and IE values from CSR.CRMD;

ÿ For implementations that support the Watch function, the PWE value in CSR.PRMD must also be restored to the WE value in CSR.CRMD;

ÿ Jump to the address recorded in CSR.ERA to fetch the instruction.

For the hardware implementation described above, if the software needs to enable interrupts during the exception handling process, it needs to save the PPLV in CSR.PRMD.

The system stores information such as PIE and restores the saved information to CSR.PRMD before the exception returns.

6.3.4 **TLB** Refill Exception Hardware Processing

When a TLB refill exception is triggered, the processor hardware performs the following operations:

ÿ Store the PLV and IE of CSR.CRMD into the PPLV and PIE of CSR.TLBRPRMD respectively, and then store the PLV of CSR.CRMD.

[1] This exception only occurs with bound memory access instructions. Except for AM*

[2] atomic memory access instructions, all other memory access instructions will generate only one type of TLB-related exception, but AM* atomic memory access instructions may simultaneously detect both page unreadable and page modified exceptions.

In addition, in this case, the page unreadable exception takes precedence over the page modified exception.

Set IE to 0, DA to 1, and PG to 0;

ÿ For implementations supporting the Watch function, the WE of CSR.CRMD should be stored in the PWE of CSR.TLBRPRMD, and then...

The WE property of CSR.CRMD is set to 0;

ÿ Record the [GRLEN-1:2] bits of the PC that triggered the exception instruction into the ERA field of CSR.TLBRERA, and then record the ERA field of CSR.TLBRERA.

Set IsTLBR to 1;

ÿ Record the virtual memory address that triggered the exception (or PC if it was triggered by instruction fetch) in CSR.TLBRBADV, and then record the virtual address...

The [VALEN-1:13] bits of the address are recorded in the VPPN field of CSR.TLBREHI;

ÿ Jump to the exception entry configured in CSR.TLBRENTTRY to fetch the pointer.

When the software executes the ERTN instruction and returns from the TLB refill exception execution, the processor hardware performs the following operations:

ÿ Restore the PPLV and PIE values from CSR.TLBRPRMD to the PLV and IE values from CSR.CRMD;

ÿ For implementations that support the Watch function, the PWE value in CSR.TLBRPRMD must also be restored to the WE value in CSR.CRMD;

ÿ Set DA to 0 and PG to 1 in CSR.CRMD;

ÿ Set IsTLBR of CSR.TLBRERA to 0;

ÿ Jump to the address recorded in CSR.TLBRERA to fetch the instruction.

6.3.5 Hardware Handling Procedures for Machine Error Exceptions

When a machine error exception is triggered, the processor hardware will perform the following operations:

ÿ Store PLV, IE, DA, PG, DATF, and DATM from CSR.CRMD into PPLV, PIE, PDA, and PDA from CSR.MERRCTL, respectively.

In PPG, PDATF, and PDATM, then set PLV of CSR.CRMD to 0, IE to 0, DA to 1, and PG to 0.

Set DATF to 0 and DATM to 0;

ÿ For implementations supporting the Watch function, the WE of CSR.CRMD should also be stored in the PWE of CSR.MERRCTL, and then...

The WE property of CSR.CRMD is set to 0;

ÿ Record the PC that triggered the exception instruction in CSR.MERRERA;

ÿ Set the IsMERR bit of CSR.MERRCTL to 1;

ÿ Record the specific error information of the verification into CSR.MERRINFO1 and CSR.MERRINFO2;

ÿ Jump to the exception entry configured in CSR.MERRENTRY to fetch the instruction.

When the software executes the ERTN instruction and returns from a machine error exception, the processor hardware performs the following operations:

ÿ Restore the PPLV, PIE, PDA, PPG, PDATF, and PDATM values in CSR.MERRCTL to the PLV values in CSR.CRMD.

IE, DA, PG, DATF, DATM;

ÿ For implementations that support the Watch function, the PWE value in CSR.MERRCTL must also be restored to the WE value in CSR.CRMD;

Set the IsMERR bit in CSR.MERRCTL to 0;

ÿ Jump to the address recorded in CSR.MERRERA to fetch the pointer.

6.4 Reset

A reset will reset all logic in the processor core, placing the circuitry into a defined state. The definition of the processor's state after a reset will be given here.

The PC of the first instruction after reset is 0x1C000000. Since the MMU will definitely be in direct address translation mode after the reset is reversed, the reset... The physical address of the first instruction fetched after the bit is also 0x1C000000.

After the reset is canceled, the contents of the registers in the determined state are:

ÿ CSR.CRMD ÿ PLV=0ÿIE=0ÿDA=1ÿPG=0ÿDATF=0ÿDATM=0ÿWE=0ÿ

ÿ CSR.EUEN's FPUen, VPUen, XVPUen, and BTUen are all 0;

ÿ All configurable bits in CSR.MISC are set to 0;

ÿ In CSR.ECFG, both VS and LIE are 0;

ÿ In CSR.ESTAT, IS[1:0] are all 0;

ÿ RDVA=0 in CSR.RVACFG;

ÿ CSR.TCFG's En=0;

ÿ CSR.LLBCTL's KLO=0;

ÿ CSR.TLBRERA ÿ IsTLBR=0ÿ

ÿ IsMERR=0 for CSR.ERRCTL;

ÿ In all implemented CSR.DMWs, PLV0~PLV3 are all 0;

ÿ In all implemented CSR.PMCFGs, all configurable bits except EvCode are 0;

ÿ All configurable bits in all implemented data breakpoint control CSRs are set to 0;

ÿ All configurable bits in all implemented instruction breakpoint control CSRs are 0;

ÿ DS=0 in CSR.DBG.

In addition to the above-specified content, after a reset is reversed, the values of other software-visible registers in the processor are uncertain, and the software... Before use, its state must be set to a defined state.

Whether the TLB and Cache undergo a hardware reset during reset is determined by the implementation; the startup software can use the configuration information provided by the processor. Decide whether a software reset is needed.

7. Control Status Register

7.1 Overview of Control Status Registers

Table 7-1 Overview of Control Status Registers

address	name	
0x0	Current mode information	CRMD
0x1	Exception Pre-Mode Information	PRMD
0x2	Extended component enable	EU
0x3	Miscellaneous Controls	MISC
0x4	Exception Configuration	ECFG
0x5	Exceptional state	STATE
0x6	Exception return address	ERA
0x7	Error virtual address	BADV
0x8	Error command	BADI
0xc	Exception Entry Address	EENTRY
0x10	TLB Index	TLBIDX
0x11	TLB High Level	TLBEHI
0x12	TLB entry low 0	EXPECTATION0
0x13	TLB Low Item 1	TLBELO1
0x18	Address space identifier	ACID
0x19	Global directory base address in the lower half of the address space	PGDL
0x1A	High half-address space global directory base address	PGDH
0x1B	Global directory base address	PGD
0x1C	Page table traversal controls the lower half of the process.	PWCL
0x1D	Page table traversal control of the high half	PWCH
0x1E	STLB page size	STLBPS
0x1F	Reduce virtual address configuration	RVACFG
0x20	Processor number	CPUID
0x21	Privileged resource allocation information 1	PRCFG1
0x22	Privileged resource allocation information 2	PRCFG2
0x23	Privileged resource allocation information 3	PRCFG3

address	name	
0x30+n (0ÿnÿ15) Data storage		SAVE
0x40	Timer number	TIME
0x41	Timer configuration	TCFG
0x42	Timer value	TVAL
0x43	Timer compensation	CNTC
0x44	Timed interrupt clear	TICLR
0x60	LLBit control	LLBCTL
0x80	Implement relevant control 1	IMPCTL1
0x81	Implement relevant control 2	IMPCTL2
0x88	TLB Re-entry Exception Address	TLBRENTRY
0x89	TLB refill exception virtual address error	TLBRBADV
0x8A	TLB Refill Exception Return Address	TLBRERA
0x8B	TLB Refill Exception Data Saving	TLBRSAVE
0x8C	TLB Refill Exception Entries Low Bit 0	TLBRELO0
0x8D	TLB Refill Exception Entry Low 1	TLBRELO1
0x8E	TLB Refill Exception Entries High Bit	TLBREHI
0x8F	TLB Refill Exception Pre-Mode Information	TLBRPRMD
0x90	Machine error control	MERRCTL
0x91	Machine error message 1	GETINFO1
0x92	Machine error message 2	GETINFO2
0x93	Machine error exception entry address	MERRENTRY
0x94	Machine error exception return address	MERRA
0x95	Machine error exception data saving	MERRSAVE
0x98	Cache tags	CTAG
0xa0	Message interruption status 0	MSGIS0
0xa1	Message interruption status 1	MSGIS1
0xa2	Message interruption status 2	MSGIS2
0xa3	Message interruption status 3	MSGIS3
0xa4	Message interruption request	MSGIR
0xa5	Message interrupt enable	MSGIE
0x180+n (0ÿnÿ3) Directly maps configuration window n		DMWn
0x200+2n (0ÿnÿ31) Performance monitoring configuration n		PMCFGn
0x201+2n (0ÿnÿ31) Performance monitoring counter n		PMCNTn

address	name	
0x300	Overall control of load/store monitoring points	MWPC
0x301	Overall status of load/store monitoring points	MWPS
0x310+8n (0ÿnÿ13)	load/store monitoring point n configuration 1	MWPnCFG1
0x311+8n (0ÿnÿ13)	load/store monitoring point n configured 2	MWPnCFG2
0x312+8n (0ÿnÿ13)	Load/store monitoring point n configured 3	MWPnCFG3
0x313+8n (0ÿnÿ13)	Load/store monitoring point n configured 4	MWPnCFG4
0x380	Overall control of monitoring points	FWPC
0x381	Overall status of the monitoring point	FWPS
0x390+8n (0ÿnÿ13)	Indicates monitoring point n, configured 1	FWPnCFG1
0x391+8n (0ÿnÿ13)	Indicates monitoring point n, configured 2	FWPnCFG2
0x392+8n (0ÿnÿ13)	Indicates monitoring point n, configured with 3.	FWPnCFG3
0x393+8n (0ÿnÿ13)	Indicates monitoring point n, configured with 4.	FWPnCFG4
0x500	Debug Register	DBG
0x501	Debug exception return address	AREA
0x502	Debug data saving	DSAVE

7.2 Description of Control Status Register Access Characteristics

7.2.1 Read/Write Attributes

The "read/write" attribute of each field will be defined later in this manual in the section on the definition of control status register fields. This "read/write" attribute primarily...

From the perspective of software access, it can be defined into four types:

ÿ RW—Software readable and writable. Except for illegal values explicitly stated in the definition that would lead to indeterminate processor execution results, the software can...

You can write any value to these fields. Normally, software performs a write-then-read operation on these fields, and the value read should be the one written. However,

An error may occur if the accessed domain can be updated by the hardware, or if an interrupt occurs between two instructions performing a read or write operation.

The current situation is that the read value and the written value are inconsistent.

ÿ R — Read-only. Writing to these fields by software will not update their contents and will not produce any other side effects.

ÿ R0 — The software always returns 0 when reading these fields. However, the software must also ensure that, either by setting the CSR write mask, it prevents further...

These new fields must either be written with a value of 0 when updating them. This requirement is to ensure backward compatibility in the software. For hardware...

In practice, fields marked with this attribute will prevent software from writing to them.

ÿ W1 — Software write of 1 is valid. Writing 0 to these fields will not clear them to 0 and will not produce any other side effects. Also, define...

The read values of the fields for this attribute have no software meaning and the software should ignore them.

7.2.2 Differences and similarities in the bit width of the control status register under the LA32 and **LA64** architectures

The bit width of all control status registers is either fixed at 32 bits, or depends on whether an LA32 or LA64 architecture is being implemented. For the first...

For registers of a certain type, when accessed by the CSR instruction under the LA64 architecture, the read return value is the sign-extended value to 64 bits, and the write return value is the high-order value.

32-bit values are automatically ignored by the hardware. For the second type, the definition will explicitly specify the differences between the LA32 and LA64 architectures.

7.2.3 Effects of accessing undefined and unimplemented control status registers

When the software accesses a CSR object using a CSR directive that is not defined in the architecture specification, or is an implementable item defined in the architecture specification but...

If the specific hardware does not implement this, a read operation can return any value, but a write operation should not change the processor state visible to the software.

Although the software uses the CSRWR or CSRXCHG instructions to write these undefined or unimplemented control status registers, in addition to changing the general-purpose register rd...

Setting it to any meaningless value will not change the processor state visible to other software, but if backward compatibility is desired, the software should not actively...

Write to these registers.

7.3 Conflicts caused by control status registers

Conflicts caused by control status registers are handled by the hardware; software does not need to add barrier instructions to avoid these conflicts.

7.4 Basic Control Status Register

7.4.1 Current Mode Information (CRMD)

The information in this register is used to determine the processor core's current privilege level, global interrupt enable, watchpoint enable, and address translation mode.

Mode.

Table 7-2 Current Mode Information Register Definition

Bit	Name reading and writing		describe
1:0	POS	RW	Current privilege level. Its valid value range is 0 to 3, where 0 represents the highest privilege level and 3 represents the lowest. Authority level. When an exception is triggered, the hardware sets the value of this field to 0 to ensure that the user is in the highest privilege level after a trap. When the execution of the ERTN instruction returns from the exception handler, If CSR.ERRCTL.IsMERR=1, the hardware restores the value of the PPLV field of CSR.ERRCTL to this value; Otherwise, if CSR.TLBRERA.IsTLBR=1, the hardware restores the value of the PPLV field of CSR.TLBRPRMD to 1. here; Otherwise, the hardware will restore the value of the PPLV field of CSR.PRMD to this location.

Bit	Name reading and writing		describe
2	IE	RW	Global interrupts are currently enabled, active high. When an exception is triggered, the hardware sets the value of this field to 0 to ensure that interrupts are masked after a trap. The exception handler determines... To re-enable interrupt response, this bit must be explicitly set to 1. When the execution of the ERTN instruction returns from the exception handler, If CSR.ERRCTL.IsMERR=1, the hardware restores the value of the PIE field of CSR.ERRCTL to this value; Otherwise, if CSR.TLBRERA.IsTLBR=1, the hardware restores the value of the PIE field of CSR.TLBRPRMD to 1. here; Otherwise, the hardware will restore the value of the PIE field of CSR.PRMD to this location.
3	AND	RW	Enable direct address translation mode, highly effective. When a TLB refill exception or machine error exception is triggered, the hardware sets this field to 1. When the execution of the ERTN instruction returns from the exception handler, If CSR.ERRCTL.IsMERR=1, the hardware restores the value of the PDA field of CSR.ERRCTL to this value; Otherwise, if CSR.TLBRERA.IsTLBR=1, the hardware sets this field to 0. The valid combinations of the DA and PG bits are 0 and 1 or 1 and 0. When the software is configured with other combinations, the result will not be valid. Sure.
4	PG	RW	Enabled for mapped address translation mode, highly active. When a TLB refill exception or machine error exception is triggered, the hardware sets this field to 0. When the execution of the ERTN instruction returns from the exception handler, If CSR.ERRCTL.IsMERR=1, the hardware restores the value of the PPG field of CSR.ERRCTL to this value; Otherwise, if CSR.TLBRERA.IsTLBR=1, the hardware sets this field to 1. The valid combinations of the PG and DA bits are 0 and 1 or 1 and 0. When the software is configured with other combinations, the result will not be valid. Sure.
6:5	DATF	RW	The memory access type for instruction fetch operations in direct address translation mode. When a machine error exception is triggered, the hardware sets this field to 0. When the execution of the ERTN instruction returns from the exception handler and CSR.ERRCTL.IsMERR=1, the hardware will... The value of the PDATF field of CSR.ERRCTL is restored here. When using software to handle TLB refills, it is recommended to set the DATF field to 1 simultaneously when the software sets PG to 1. It is 0b01, which means it is a consistent cacheable type.
8:7	DATM	RW	Storage access type for load and store operations in direct address translation mode. When a machine error exception is triggered, the hardware sets this field to 0. When the execution of the ERTN instruction returns from the exception handler and CSR.ERRCTL.IsMERR=1, the hardware will... The value of the PDATM field of CSR.ERRCTL is restored here. When using software to handle TLB refills, it is recommended to set DATM to 1 when the software sets PG to 1. 0b01, which is a consistent cacheable type.

Bit	Name reading and writing		describe
9	WE	RW	Enable bit for command and data monitoring points, active high. When an exception is triggered, the hardware sets the value of this field to 0. When the execution of the ERTN instruction returns from the exception handler, If CSR.ERRCTL.IsMERR=1, the hardware restores the value of the PWE field of CSR.ERRCTL to this value; Otherwise, if CSR.TLBRERA.IsTLBR=1, the hardware restores the value of the PWE field of CSR.TLBRPRMD to 1. here; Otherwise, the hardware will restore the value of the PWE field of CSR.PRMD to this location.
31:10	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

7.4.2 Pre-Exception Mode Information (PRMD)

When an exception is triggered, if the exception type is not a TLB refill exception or a machine error exception, the hardware will adjust the privilege level of the processor core at that time.

The global interrupt enable and watchpoint enable bits are saved to the pre-exception mode information register, which is used to restore the processor core's context when the exception returns.

Table 7-3 Definition of Exception Mode Information Register

Bit	Name reading and writing		describe
1:0	PPLV	RW	When an exception is triggered, if the exception type is not a TLB refill exception or a machine error exception, the hardware will call CSR.CRMD. The old value of the PLV field is recorded in this field. When the exception being handled is neither a TLB refill exception (CSR.TLBRERA.IsTLBR=0) nor a machine error exception When CSR.ERRCTL.IsMERR=0, the hardware will return from the exception handler after executing the ERTN instruction. The value of this field is restored to the PLV field of CSR.CRMD.
2	ABOUT	RW	When an exception is triggered, if the exception type is not a TLB refill exception or a machine error exception, the hardware will call CSR.CRMD. The old value of the IE field is recorded in this field. When the exception being handled is neither a TLB refill exception (CSR.TLBRERA.IsTLBR=0) nor a machine error exception When CSR.ERRCTL.IsMERR=0, the hardware will return from the exception handler after executing the ERTN instruction. The value of this field is restored to the IE field of CSR.CRMD.
3	WEIGHT	RW	When an exception is triggered, if the exception type is not a TLB refill exception or a machine error exception, the hardware will call CSR.CRMD. The old value of the WE field is recorded in this field. When the exception being handled is neither a TLB refill exception (CSR.TLBRERA.IsTLBR=0) nor a machine error exception When CSR.ERRCTL.IsMERR=0, the hardware will return from the exception handler after executing the ERTN instruction. The value of this field is restored to the WE field of CSR.CRMD.
31:4	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

7.4.3 Extended Component Enable (EUEN)

In addition to the basic integer instruction set and privileged instruction set, there are also the basic floating-point instruction set, the binary translation extended instruction set, and the 128-bit vector extended instruction set.

Both the standard instruction set and the 256-bit vector extension instruction set have software-configurable enable bits. When these enable bits are disabled, the corresponding instruction is executed.

This will trigger a corresponding instruction unavailability exception. Software can use this mechanism to determine the scope of context saving. Hardware implementations can also utilize this.

The control bit at the location enables circuit power consumption control.

Table 7-4 Extended Instruction Enable Register Definitions

Bit	Name reading and writing		describe
0	FPE	RW	Basic floating-point instruction enable bit. When this bit is 0, execution of the basic floating-point instructions described in Section 3.2 will be triggered. Floating-point instruction not enabled exception (FPD).
1	SXE	RW	128-bit vector extension instruction enable control bit. When this bit is 0, the 128-bit vector extension described in Volume 2-a is executed. The instruction will trigger a 128-bit vector extension instruction not enabled exception (SXD).
2	ASXE	RW	256-bit vector extension instruction enable control bit. When this bit is 0, the 256-bit vector extension described in Volume 2-a is executed. The instruction will trigger a 256-bit vector extension instruction not enabled exception (ASXD).
3	BTE	RW	Binary translation extension instruction enable control bit. When this bit is 0, the binary translation extension instructions described in Volume 3 are executed. This will trigger a Binary Translation Extensions Not Enabled (BTD) exception.
31:4	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

7.4.4 Miscellaneous (MISC)

This register contains control bits for processor core behavior at different privilege levels, including whether to enable 32-bit address mode.

Controls whether partial privileged instructions are allowed to be used at non-privileged levels, and controls address non-alignment checks and page table write-allowed checks.

Table 7-5 Miscellaneous Register Definitions

Bit	Name reading and writing		describe
0	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	
1	VA32L1	RW	At PLV1 privilege level, whether to enable 32-bit address mode. 0—Disabled, 1—Enabled. This bit is readable and writable only in the LA64 architecture. In the LA32 architecture privilege level, the read/write attribute of this bit is R0.
2	VA32L2	RW	Under PLV2 privilege level, whether to enable 32-bit address mode. 0—Disabled, 1—Enabled. This bit is readable and writable only in the LA64 architecture. In the LA32 architecture privilege level, the read/write attribute of this bit is R0.
3	VA32L3	RW	At PLV3 privilege level, whether to enable 32-bit address mode. 0—Disabled, 1—Enabled. This bit is readable and writable only in the LA64 architecture. In the LA32 architecture privilege level, the read/write attribute of this bit is R0.
4	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	
5	DRDTL1	RW	Under PLV1 privilege level, whether to disable RDTIME type instructions. When this bit is 1, instructions executed under PLV1 privilege level... RDTIME type instructions will trigger an instruction privilege level error exception (IPE).
6	DRDTL2	RW	Under PLV2 privilege level, whether to disable RDTIME type instructions. When this bit is 1, instructions executed under PLV2 privilege level... RDTIME type instructions will trigger an instruction privilege level error exception (IPE).
7	DRDTL3	RW	Under PLV3 privilege level, whether to disable RDTIME type instructions. When this bit is 1, instructions executed under PLV3 privilege level... RDTIME type instructions will trigger an instruction privilege level error exception (IPE).
8	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

Bit	Name	reading and writing	describe
9	RPCNTL1	RW	At PLV1 privilege level, this determines whether software is allowed to read performance counters. When this bit is 1, at PLV1 privilege level... Accessing any implemented performance counter using the CSRRD instruction will not trigger an instruction privilege level exception. (CALL).
10	RPCNTL2	RW	Under PLV2 privilege level, this determines whether software is allowed to read the read performance counter. When this bit is 1, the PLV2 privilege level... Using the CSRRD instruction to access any implemented performance counter PCNT will not trigger an instruction privilege level error. External (IPE).
11	RPCNTL3	RW	At PLV3 privilege level, this determines whether software is allowed to read the read performance counter. When this bit is 1, the PLV3 privilege level... Using the CSRRD instruction to access any implemented performance counter PCNT will not trigger an instruction privilege level error. External (IPE).
12	ALCL0	RW	At PLV0 privilege level, whether to perform an alignment check on non-vector load/store instructions that are allowed to be unaligned . The value is 1. This indicates that an inspection will be performed, and if a violation is found, an address alignment error exception will be triggered. This bit is readable and writable only if the hardware implementation supports non-vector load/store instruction address unalignment; otherwise... This bit is read-only and is always 1.
13	ALCL1	RW	Under PLV1 privilege level, whether to perform an alignment check on non-vector load/store instructions that are allowed to be unaligned . (Set to 1) This indicates that an inspection will be performed, and if a violation is found, an address alignment error exception will be triggered. This bit is readable and writable only if the hardware implementation supports non-vector load/store instruction address unalignment; otherwise... This bit is read-only and is always 1.
14	ALCL2	RW	Under PLV2 privilege level, whether to perform an alignment check on non-vector load/store instructions that are allowed to be unaligned . (Set to 1) This indicates that an inspection will be performed, and if a violation is found, an address alignment error exception will be triggered. This bit is readable and writable only if the hardware implementation supports non-vector load/store instruction address unalignment; otherwise... This bit is read-only and is always 1.
15	ALCL3	RW	At PLV3 privilege level, whether to perform an alignment check on non-vector load/store instructions that are allowed to be unaligned . (Set to 1) This indicates that an inspection will be performed, and if a violation is found, an address alignment error exception will be triggered. This bit is readable and writable only if the hardware implementation supports non-vector load/store instruction address unalignment; otherwise... This bit is read-only and is always 1.
16	DWPL0	RW	Under PLV0 privilege level, should the check of the page table entry write enable bit be disabled during TLB virtual-to-physical address translation? When the bit is 1, the store instruction will not trigger a page modification exception even if it accesses a page table entry with D=0.
17	DWPL1	RW	Under PLV1 privilege level, does it disable the check of page table entry write enable bits during TLB virtual-to-physical address translation? When the bit is 1, the store instruction will not trigger a page modification exception even if it accesses a page table entry with D=0.
18	DWPL2	RW	Under PLV2 privilege level, should the check of the page table entry write enable bit be disabled during TLB virtual-to-physical address translation? When the bit is 1, the store instruction will not trigger a page modification exception even if it accesses a page table entry with D=0.
31:19	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

[1] The instructions affected by this control bit are: LD[X].{H[U]/W[U]/D}, ST[X].{H/W/D}, LDPTR.{W/D}, STPTR.{W/D}, FLD[X].{S/D}, FST[X].{S/D}, LDPTE, LDDIR, IOCSRRD.{H/W/D}ÿIOCSRWR.{H/W/D}ÿ

7.4.5 Exceptional Configuration (ECFG)

This register is used to control the entry calculation method for exceptions and interrupts, as well as the local enable bits for each interrupt.

Table 7-6 Exception Configuration Register Definitions

Bit	Name	reading and writing	describe
12:0	LIE	RW	Local interrupt enable bits, active high. These local interrupt enable bits correspond to the 13 bits recorded in the IS field of CSR.ESTAT. Each interrupt source is matched one-to-one, with each bit controlling one interrupt source.
15:13	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.
18:16	VS	RW	Configure the spacing between exception and interrupt entries. When VS=0, all exceptions and interrupts share the same entry address. When VS!=0, the entry address spacing between each exception and interrupt is 2VS instructions. Because TLB refill exceptions and machine error exceptions have independent entry base addresses, their exception entries are not affected by VS. The impact of the domain.
31:19	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.4.6 Exception Status (ESTAT)

This register records the status information of the exception, including the first and second level codes of the triggered exception, as well as the status of each interrupt.

Table 7-7 Exception Status Register Definitions

Bit	Name	reading and writing	describe
1:0	IS[1:0]	RW	Two software interrupt status bits. Bits 0 and 1 correspond to SWI0 and SWI1, respectively. Software interrupt settings are also accomplished using these two bits: 1 for software write, 0 for write clear interrupt.
12:2	IS[12:2]	R	Interrupt status bit. A value of 1 indicates that the corresponding interrupt is enabled. There is one inter-core interrupt (IPI) and one timer interrupt (TI). One performance counter overflow interrupt (PMI), and eight hardware interrupts (HWI0~HWI7). In online interrupt mode, the hardware simply samples each interrupt source on a clock cycle and records its state. At this time, for all... The requirement that interrupts must be level interrupts is guaranteed by the interrupt source and is not maintained here.
13	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.
14	0	R0	When the implementation does not support external interrupts and uses message interrupt mode (CPUCFG.1.MSG_INT[bit26]=0), read return. 0, and the software does not allow its value to be changed.
14	MsgInt	R	When the implementation does not support external interrupts and uses message interrupt mode (CPUCFG.1.MSG_INT[bit26]=0), this bit is... 1 indicates that a message interruption has occurred.
15	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.
21:16	Ecode	R	Exception type first-level encoding. When an exception is triggered: If it is a TLB refill exception or a machine error exception, the field remains unchanged; Otherwise, the hardware will write the value defined in the Ecode column of Table 7-8 into this field, depending on the exception type.
30:22	EsubCode	R	Exception type two-level encoding. When an exception is triggered: If it is a TLB refill exception or a machine error exception, the field remains unchanged; Otherwise, the hardware will write the value defined in the EsubCode column of Table 7-8 into this field, depending on the exception type.

Bit	Name reading and writing		describe
31	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

Table 7-8 Exception Code Table

Ecode	EsubCode Exception Code		Exception types
0x0		INT	An interrupt is indicated only when CSR.ECFG.VS=0.
0x1		PIL	Load operation page invalid exception
0x2		PIS	Store action page invalid exception
0x3		PIF	Invalid Fetch Operation Page Exception
0x4		SMEs	Page modification exceptions
0x5		PNR	Page unreadable exception
0x6		PNX	Page non-executable exception
0x7		PPI	Page privilege level non-compliance exception
0x8	0	ADEF	Fetch Address Error Exception
	1	ADEM	memory access instruction address error exception
0x9		BUT	Exceptions to address misalignment
0xA		BCE	Boundary check error exception
0xB		SYS	System call exceptions
0xC		BRK	Breakpoint Exception
0xD		I HAVE	The instruction has no exceptions
0xE		CALL	Command privilege level error exception
0xF		FPD	Floating-point instruction not enabled exception
0x10		SXD	128-bit vector extension instruction not enabled exception
0x11		ASXD	256-bit vector extension instruction not enabled exception
0x12	0	FPE	Exceptions to basic floating-point instructions
	1	VFPE	Vector floating-point instruction exceptions
0x13	0	WPEF	finger retrieval monitoring point exception
	1	WPEM	Load/store operation monitoring point exception
0x14		BTD	Binary translation extension instructions not enabled exception
0x15		BTE	Exceptions related to binary translation
0x16		GSPR	Client Sensitive Privileged Resource Exception
0x17		HVC	Virtual machine monitoring call exception
0x18	0	GCSC	Client CSR software modification exception
	1	GCHC	Client CSR Hardware Modification Exception

Ecode	EsubCode Exception	Code	Exception types
0x1A-0x3E			Reserved code

7.4.7 Exception Return Address (ERA)

This register records the return address after a normal exception has been handled. When an exception is triggered, if the exception type is not TLB, the exception is refilled.

If it is not a machine error exception, the PC of the instruction that triggered the exception will be recorded in this register.

Table 7-9 Exception Program Counter Register Definitions

Bit	Name reading	and writing	describe
GRLEN-1:0	PC	RW	When an exception is triggered: If it is a TLB refill exception or a machine error exception, the field remains unchanged; Otherwise, the hardware will record the PC that triggered the exception here. For the LA64 architecture, in this case, If the privilege level that triggered the exception is in 32-bit address mode, then the high 32 bits of the recorded PC value are forcibly set to 0. 0ÿ

7.4.8 Error Virtual Address (BADV)

This register is used to record the virtual address of the error when an address error-related exception is triggered. Such exceptions include:

ÿ Instruction Fetch Error (ADEF) exception: In this case, the PC of the instruction is recorded.

ÿ Load/store operation address error exception (ADEM)

ÿ Address alignment misalignment exception (ALE)

ÿ Boundary constraint check error exception (BCE)

ÿ Load operation page invalid exception (PIL)

ÿ Store Action Page Invalid Exception (PIS)

ÿ Invalid Fetch Page Exception (PIF)

ÿ Page Modification Exception (PME)

ÿ Page Unreadable Exception (PNR)

ÿ Page Non-Executable Exception (PNX)

ÿ Page Privilege Level Non-Compliance Exception (PPI)

Table **7-10** Error Virtual Address Register Definitions

Bit	Name reading	and writing	describe
GRLEN-1:0	VAddr RW		When an address error-related exception is triggered, the hardware records the erroneous virtual address here. For the LA64 architecture, this... In this case, if the privilege level that triggers the exception is in 32-bit address mode, then the high 32 bits of the recorded virtual address... The bit is forcibly set to 0.

7.4.9 Error Instruction (BADI)

This register is used to record the instruction code of the instruction that triggers a synchronization class exception. Synchronization class exceptions refer to exceptions other than interrupts (INT) and guest exceptions.

All exceptions except CSR Hardware Modification Exception (GCHC) and Machine Error Exception (MERR).

Table **7-11** Error Instruction Register Definitions

Bit	Name	reading and writing	describe
31:0	Inst	R	When triggers a synchronization class exception, the hardware records the instruction code that triggered the exception here.

7.4.10 Exception Entry Address (EENTRY)

This register is used to configure the entry address for normal exceptions and interrupts.

Table **7-12** Exception Entry Page Number Register Definitions

Bit	Name	reading and writing	describe
11:0	0	R	is always 0 when read-only, and writes are ignored.
GRLEN-1:12	VPN	RW	is the page number of the page containing the entry address for normal exceptions and interrupts.

7.4.11 Reduce Virtual Address Configuration (RVACFG)

This register is used to control the reduced address width in virtual address reduction mode.

Table **7-13** Reduced Virtual Address Register Definitions

Bit	Name reading and writing		describe
3:0	RBits	RW	In virtual address reduction mode, this refers to the number of bits in the high-order address space that are reduced. It can be configured to a value between 0 and 8. 0 is a special configuration value that means virtual address reduction mode is not enabled. If the configured value is greater than 8, the processor behavior is unpredictable.
31:4	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.4.12 Processor ID (CPUID)

This register contains processor core number information.

Table **7-14** Processor Number Register Definition

Bit	Name reading and writing		describe
8:0	CoreID	R	The processor core number. This information is used by software to distinguish each processor core in a multi-core system. During system integration, each... The processor core number information for each processor core is set by the hardware based on the specific implementation. It is recommended that the processor in the system... The numbering starts from 0 and increments.
31:9	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.4.13 Privileged Resource Configuration Information 1 (PRCFG1)

This register contains configuration information for some privileged resources.

Table **7-15** Privileged Resource Configuration Information **1.** Register Definitions

Bit	Name reading and writing		describe
3:0	SAVENum		R SAVE controls the number of status registers.
11:4	TimerBits		Decrement the effective number of bits of the R timer by 1.
14:12	VSMax		The maximum value that can be set for the R exception and interrupt vector entry spacing (CSR.ECFG.VS domain).
31:15	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.4.14 Privileged Resource Configuration Information 2 (PRCFG2)

This register contains configuration information for some privileged resources.

Table **7-16** Privileged Resource Configuration Information **2** Register Definitions

Bit	Name reading and writing		describe
GRLEN-1:0	PSAVL		R indicates the page size that the TLB can support. When the i-th bit is 1, it indicates that pages of size 2i bytes are supported.

7.4.15 Privileged Resource Configuration Information 3 (PRCFG3)

This register contains configuration information for some privileged resources.

Table **7-17** Privileged Resource Configuration Information **3** Register Definitions

Bit	Name	reading and writing	describe
3:0	TLBType	R	Instructions for TLB organization: 0: No TLB; 1: A fully associative multi-page size TLB (MTLB) 2: A fully associative multi-page-size TLB (MTLB) + a group-associative single-page-size TLB (STLB). Other values: Reserved.
11:4	MTLBEntries	R	When TLBType=0, this field is read-only and always equal to 0; When TLBType=1 or 2, the value of this field is the number of entries in the fully associative multi-page size TLB minus 1.
19:12	STLBWays	R	When TLBType=0 or 1, this field is read-only and always equal to 0; When TLBType=2, the value of this field is the number of paths in a group-associative single page size TLB minus 1.
25:20	STLBSets	R	When TLBType=0 or 1, this field is read-only and always equal to 0; When TLBType=2, the value of this field is a power of the number of items per way in a group-associative single-page-size TLB, i.e., per There are 2 on the waySTLBSets items.
31:26	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

7.4.16 Data Saving (SAVE)

The data storage control status register is used to temporarily store data for system software. Each data storage register can store data from one general-purpose register. data.

The minimum number of data storage registers implemented is 1, and the maximum is 16. The specific number can be determined by software configuration using CSR.PRCFG1.SAVENum. We learned that, starting from SAVE0, the addresses of each SAVE register are 0x30, 0x31, ..., 0x30+SAVENum-1.

All data storage control status registers use the same format, as shown in Table 7-18.

Table **7-18** Data Storage Register Definitions

Bit	Name	reading and writing	describe
GRLEN-1:0	Data RW is data that can only be read and written by software. The hardware will not modify the contents of this field except when executing CSR instructions.		

7.4.17 LLBit Control (LLBCTL)

This register is used for access control operations on LLBit.

Table **7-19 LLBit** Register Definition

Bit	Name	reading and writing	describe
0	ROLLB	R is a read-only bit that returns the current value of LLBit.	
1	WCLLB	Writing a 1 to this bit in the W1 software will clear LLBit to 0. Writing a 0 to this bit in the software will be ignored by the hardware.	
2	AT	RW	Used to control the operation of LLBit when the ERTN instruction is executed. When this bit is equal to 1, the LLBit bit is not cleared to 0 when the ERTN instruction is executed, but the bit will be automatically cleared by the hardware. Setting KLO to 0 means that each time KLO is set to 1, it can only affect the execution of the ERTN instruction once.

Bit	Name reading and writing		describe
31:3	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.4.18 Implement related control **1 (IMPCTL1)**

This register contains control information related to the microarchitectural characteristics of the specific implementation. Its format and the specific meaning of each field are defined by the specific implementation.

7.4.19 Implement Related Control **2 (IMPCTL2)**

This register contains control information related to the microarchitectural characteristics of the specific implementation. Its format and the specific meaning of each field are defined by the specific implementation.

7.4.20 Cache Tag (CTAG)

This register is used by the CACOP instruction to directly access the cache, storing content read from the cache tag or content to be written to the cache tag. The content, format, and specific meaning of each field are defined by the implementation.

7.5 Mapped Address Translation Related Control Status Registers

7.5.1 TLB Index (TLBIDX)

This register contains information such as index values related to TLB instruction operations. The bit width of the Index field in Table 7-20 is implementation-dependent, however... The allowed index bit width in this architecture is no more than 16 bits.

This register also contains information related to the PS and E fields in the TLB entry during TLB instruction operations.

Table **7-20 TLB** Index Register Definitions

Bit	Name reading and writing		describe
n-1:0	Index	RW	When executing the TLBRD and TLBWR instructions, the index value for accessing TLB entries comes from this. When the TLBSRCH instruction is executed, if a hit occurs, the index value of the hit item is recorded here. For information on the correspondence between index values and TLB entries, please refer to the relevant content in Section 4.2.4.1.
15:n	0	R	is always 0 when read-only, and writes are ignored.
23:16	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.
29:24	PS	RW	When the TLBRD instruction is executed, the value of the PS field of the read TLB entry is recorded here. When CSR.TLBRERA.IsTLBR=0, the TLBWR and TLBFILL instructions are executed, writing the PS of the TLB table entry. The value of the field comes from this.
30	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

Bit	Name	reading and writing	describe
31	IS	RW	A value of 1 indicates that the TLB entry is empty (invalid TLB entry), and a value of 0 indicates that the TLB entry is not empty (valid). TLB entries). When executing TLBSRCH, if there is a hit, this bit is recorded as 0; otherwise, it is recorded as 1. When executing TLBRD, the E bit information of the read TLB entry is inverted and recorded here. When executing the TLBWR or TLBFILL instruction, if CSR.TLBRERA.IsTLBR=0, the value of this bit is inverted and then written. Move to bit E of the TLB entry being written; if CSR.TLBRERA.IsTLBR=1 at this time, then bit E of the TLB entry being written... It is always set to 1, regardless of the value of that bit.

7.5.2 TLB High Bit (TLBEHI)

This register contains information related to the virtual page number (VPPN) in the high-order part of the TLB entry during TLB instruction operations.

The bit width of a field is related to the effective virtual address range supported by the implementation, so the definitions of register fields are described separately.

Table **7-21 TLB** Page Table High-Level Register Definitions (LA64 Architecture)

Bit	Name	reading and writing	describe
12:0	0	R	is always 0 when read-only, and writes are ignored.
VALEN-1:13	VPPN	RW	When the TLBRD instruction is executed, the value of the VPPN field of the read TLB entry is recorded here. When CSR.TLBRERA.IsTLBR=0, the TLBSRCH instruction queries the VPPN value used by the TLB, and executes... The value of the VPPN field written to the TLB entry when executing the TLBWR and TLBFILL instructions comes from this. When the following exceptions are triggered: load operation page invalidity exception, store operation page invalidity exception, fetch operation page invalidity exception, page modification... When an exception occurs, such as an unreadable page exception, a non-executable page exception, or a non-compliant page privilege level exception, the virtual space that triggers the exception is... The [VALEN-1:13] bit of the address is recorded here.
63:VALEN	Sign_Ext	R	The read return value is a sign extension of the highest bit of the VPPN field; these bits are ignored when writing.

Table **7-22 TLB** Page Table High-Level Register Definitions (LA32 Architecture)

Bit	Name	reading and writing	describe
12:0	0	R	is always 0 when read-only, and writes are ignored.
31:13	VPPN	RW	When the TLBRD instruction is executed, the value of the VPPN field of the read TLB entry is recorded here. When CSR.TLBRERA.IsTLBR=0, the TLBSRCH instruction queries the VPPN value used by the TLB, and executes... The value of the VPPN field written to the TLB entry when executing the TLBWR and TLBFILL instructions comes from this. When the following exceptions are triggered: load operation page invalidity exception, store operation page invalidity exception, fetch operation page invalidity exception, page modification... When an exception occurs, such as an unreadable page exception, a non-executable page exception, or a non-compliant page privilege level exception, the virtual space that triggers the exception is... The [31:13]th position of the address is recorded here.

7.5.3 Low bits of TLB entries (TLBELO0, TLBELO1)

The TLBELO0 and TLBELO1 registers contain information such as the physical page number of the lower-order part of the TLB entry when the TLB instruction is executed.

Because the TLB in the Dragon architecture uses a two-page structure, the low-order bits of the TLB entry correspond to the odd and even physical page entries, with the even-numbered page information in...

In TLBELO0, odd-numbered page information is stored in TLBELO1. The format definitions of the TLBELO0 and TLBELO1 registers are identical, and their respective fields...

The definitions are in Tables 7-23 and 7-24.

When CSR.TLBRERA.IsTLBR=0, the TLBWR and TLBFILL instructions are executed, writing to the G, PPN0, V0, PLV0, and PLV0 entries in the TLB table.

The values of the fields MAT0, D0, NR0, NX0, RPLV0, PPN1, V1, PLV1, MAT1, D1, NR1, NX1, and RPLV1 come from...

For TLBELO0 and TLBELO1.

When the TLBRD instruction is executed, the information read from the TLB entries is written one by one into the TLBEL0 and TLBEL01 registers.

In the corresponding domain.

Table **7-23 TLB** Entries Low-order Register Definitions (LA64 Architecture)

Bit	Name reading	and writing	describe
0	In	The valid bit (V) of the RW page table entry.	
1	D	Dirty position (D) of RW page table entries.	
3:2	POS	Privilege Level (PLV) of RW page entries.	
5:4	ALONG WITH	Storage Access Type (MAT) for RW page table entries.	
6	G	RW	Global flags (G) for page table entries. When executing the TLBFILL and TLBWR instructions, the fill is only performed if the G bits in both TLBELO0 and TLBELO1 are 1. The G bit in the page table entry in the TLB is 1. When the TLBRD instruction is executed, if the G bit of the read TLB entry is 1, then the entries in TLBLO0 and TLBLO1... The G bit is simultaneously set to 1.
11:7	0	R is always 0 when read-only, and writes are ignored.	
POLES-1:12	PPN	Physical page number (PPN) of the RW page table.	
60:POLES	0	R is always 0 when read-only, and writes are ignored.	
61	No.	Unreadable bit (NR) of RW page table entries.	
62	NX	The non-executable bit (NX) of the RW page table entry.	
63	RPLV	RW	Restricted Privilege Level (RPLV) enable for page tables. When RPLV=0, the page table entry can be accessed by any privilege level. Programs with a privilege level of at least PLV can access this page table entry; when RPLV=1, this page table entry can only be accessed by programs with a privilege level equal to PLV. access.

Table **7-24 TLB** Entries Low-order Register Definitions (LA32 Architecture)

Bit	Name reading	and writing	describe
0	In	The valid bit (V) of the RW page table entry.	
1	D	Dirty position (D) of RW page table entries.	

Bit	Name	reading and writing	describe
3:2	POS		Privilege Level (PLV) of RW page entries.
5:4	ALONG WITH		Storage Access Type (MAT) for RW page table entries.
6	G	RW	Global flags (G) for page table entries. When executing the TLBFILL and TLBWR instructions, the fill is only performed if the G bits in both TLBELO0 and TLBELO1 are 1. The G bit in the page table entry in the TLB is 1. When the TLBRD instruction is executed, if the G bit of the read TLB entry is 1, then the entries in TLBLO0 and TLBLO1... The G bit is simultaneously set to 1.
7	0		R is always 0 when read-only, and writes are ignored.
POLES-5:8	PPN		Physical page number (PPN) of the RW page table.
31:POLES-4	0		R is always 0 for read-only operations; write operations are ignored. This field does not exist when PALEN=36.

7.5.4 Address Space Identifier (ASID)

This register contains the address space identifier (ASID) information used for memory access operations and TLB instructions. The bit width of the ASID varies depending on the architecture.

The specification may evolve further, and to make it easier for software to clearly define the bit width of the ASID, this information will be provided directly.

Table **7-25** Address Space Identifier Register Definitions

Bit	Name	reading and writing	describe
9:0	ACID	RW	The address space identifier corresponding to the currently executing program. It is used as the ASID key value information for querying the TLB when fetching instructions and executing load/store instructions. When executing the TLBSRCH and TLBCLR instructions, the ASID key value information of the TLB is used to query the TLB. When the TLBWR or TLBFILL instruction is executed, the value written to the ASID field of the TLB entry comes from this. When the TLBRD instruction is executed, the contents of the ASID field of the TLB entry are recorded here.
15:10	0		R is always 0 when read-only, and writes are ignored.
23:16	ASIDBITS		The bit width of the R ASID field. It is directly equal to the value of this field.
31:24	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.5.5 Global Directory Base Address in the Lower Half-Address Space (PGDL)

This register is used to configure the base address of the global directory in the lower half of the address space. The base address of the global directory must be aligned to a 4KB boundary address.

Therefore, the lowest 12 bits of this register are not configurable by software and are always 0 (read-only).

Table **7-26** Definitions of Global Directory Base Address Registers in the Lower Half-Address Space

Bit	Name	reading and writing	describe
11:0	0		R is always 0 when read-only, and writes are ignored.
GRLEN-1:12	Base	RW	The base address of the global directory in the lower half of the address space. The so-called lower half-address space refers to the virtual address where the [VALEN-1]th bit is equal to 0.

7.5.6 Global Directory Base Address in High Half-Space (PGDH)

This register is used to configure the base address of the global directory in the high half-address space. The base address of the global directory must be aligned to a 4KB boundary address. Therefore, the lowest 12 bits of this register are not configurable by software and are always 0 (read-only).

Table **7-27** Definitions of Global Directory Base Address Registers in the High Half-Address Space

Bit	Name	reading and writing	describe
11:0	0	R	is always 0 when read-only, and writes are ignored.
GRLEN-1:12	Base	RW	The base address of the global directory in the high half-address space. The so-called high half-address space refers to the virtual address where the [VALEN-1]th bit is equal to 1.

7.5.7 Global Directory Base Address (PGD)

This register is a read-only register, and its content is the global directory base address information corresponding to the virtual address of the error in the current context. The read-only information of the device is used not only for the read return value of CSR-type instructions, but also for the base address information required by the LDDIR instruction when accessing the global directory.

Table **7-28** Global Directory Base Address Register Definitions

Bit	Name	reading and writing	describe
11:0	0	R	is always 0 when read-only, and writes are ignored.
GRLEN-1:12	Base	R	If the highest bit of the error virtual address in the current context is 0, the read return value is equal to the Base field of CSR.PGDL; otherwise... Then, the read return value is equal to the Base field of CSR.PGDH. When CSR.TLBRERA.IsTLBR=0, the error virtual address information in the current context is located in CSR.BADV; otherwise... The error virtual address information is located in CSR.TLBRBADV.

7.5.8 Page Table Traversal Control of Lower Half (PWCL)

Together with the information in this register and the CSR.PWCH register, it defines the page table structure used by the operating system. This information will be used for... Instructs the software or hardware to perform page table traversal. See Section 5.4.5 for a diagram of the page table structure and traversal process.

In the LA32 architecture, only CSR.PWCL is implemented. Therefore, the PWCL register must contain all the information describing the page table structure, which leads to... The starting address of the page table and the lowest two-level directories cannot exceed 32 bits, a limitation that still exists under the LA64 architecture.

Table **7-29** defines the lower half registers for traversal control.

Bit	Name	reading and writing	describe
4:0	PTbase	RW	is the starting address of the last-level page table (level 0 page table).
9:5	PTwidth	RW	The number of index bits for the last-level page table (level 0 page table).
14:10	Dir1_base	RW	is the starting address of the lowest level directory (level 1 page table).
19:15	Dir1_width	RW	: The number of bits in the index of the lowest level directory (level 1 page tables). 0 indicates that this level does not exist.

Bit	Name reading and writing		describe
24:20	Dir2_base		The starting address of the second-lowest level directory (level 2 page table) in RW.
29:25	Dir2_width		RW index bit length for the next lower level directory (level 2 page table). 0 indicates that this level does not exist.
31:30	PTEWidth	RW	The bit width of each page table entry in memory. 0 represents 64 bits, 1 represents 128 bits, 2 represents 256 bits, and 3 represents 512 bits.

7.5.9 Page Table Traversal Control of the High Half (PWCH)

Together with the information in this register and the CSR.PWCL register, it defines the page table structure used by the operating system. This information will be used...

Instructs the software or hardware to perform page table traversal. See Section 5.4.5 for a diagram of the page table structure and traversal process.

This register is defined only in the LA64 architecture.

Table **7-30** defines the high half registers for traversal control.

Bit	Name reading and writing		describe
5:0	Dir3_base		The starting address of the RW next-highest level directory (3rd level page table).
11:6	Dir3_width		RW index bit length for the next higher level directory (level 3 page table). 0 indicates that this level does not exist.
17:12	Dir4_base		RW is the starting address of the highest-level directory (level 4 page table).
23:18	Dir4_width		RW specifies the number of bits in the index of the highest-level directory (level 4 page tables). 0 indicates that this level does not exist.
24	0	0	When the implementation does not support hardware page table traversal (CPUCFG.2.HPTW[bit24]=0), the read returns 0, and the software does not allow it. Allow it to change its value.
24	HPTW_En	RW	When hardware page table traversal is supported (CPUCFG.2.HPTW[bit24]=1), this bit enables hardware page table traversal functionality. The enable bit is set to 1 to enable and 0 to disable.
31:25	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.5.10 STLB Page Size (STLBPS)

This register is used to configure the page size in STLB.

Table **7-31 STLB** Page Size Register Definition

Bit	Name reading and writing		describe
5:0	PS		The page size of a RW STLB is a power of 2. For example, if the page size is 16KB, then PS = 0xE.
31:6	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.5.11 TLB Refill Exception Entry Address (TLBRENTRY)

This register is used to configure the entry address for a TLB refill exception. Because after a TLB refill exception is triggered, the processor core will enter the direct address...

Since this is a translation mode, the entry address entered here should be a physical address.

Table **7-32 TLB** Refill Exception Entry Address Register Definition (LA64 Architecture)

Bit	Name reading and writing		describe
11:0	0	R	TLB refills the exception entry address [11:0]. Read-only is always 0, write is ignored.
POLES-1:12	PPN	RW	TLB Refill the exception entry address [PALEN-1:12] bits. The address entered here should be a physical address.
63:POLES	0	R	is always 0 when read-only, and writes are ignored.

Table **7-33 TLB** Refill Exception Entry Address Register Definitions (LA32 Architecture)

Bit	Name reading and writing		describe
11:0	0	R	TLB refills the exception entry address [11:0]. Read-only is always 0, write is ignored.
31:12	PPN	RW	TLB Refill the exception entry address [31:12]. The address entered here should be a physical address.

7.5.12 TLB Refill Exception Error Virtual Address (TLBRBADV)

This register is used to record the virtual address of the error that triggers the TLB refill exception.

Table **7-34 TLB** Refill Exception Error Virtual Address Register Definitions

Bit	Name reading and writing		describe
GRLEN-1:0	VAddr	RW	When a TLB refill exception is triggered, the hardware records the erroneous virtual address here. For the LA64 architecture, in this situation... In this case, if the privilege level that triggers the exception is in 32-bit address mode, then the high 32 bits of the recorded virtual address are strong. The setting is 0.

7.5.13 TLB Refill Exception Return Address (TLBRERA)

This register stores the return address after the TLB refill exception handling is complete. In addition, this register also contains information to identify the current exception.

This is a flag for TLB refill exceptions.

Table **7-35 TLB** Refill Exception Return Address Register Definition

Bit	Name reading and writing		describe
0	IsTLBR	RW	A value of 1 indicates that the current context is TLB refill exception handling. When a TLB refill exception is triggered, the hardware sets this bit to 1. When this bit is 1, the ERTN instruction will clear it to 0 only if CSR.ERRCTL.IsMERR=0. Otherwise, it remains unchanged. Because a separate CSR is defined for TLB refill exceptions in the architecture, when this bit is 1, When ERTN returns, the information used to restore CSR.CRMD will come from CSR.TLBRPRMD; The ERTN return address information will come from CSR.TLBRERA; The table entry information to be written by the TLBWR and TLBFILL commands will come from CSR.TLBREHI. ÿ CSR.TLBELO0ÿCSR.TLBELO1ÿ The information retrieved by the TLBSRCH command comes from CSR.TLBREHI; The error virtual address information required for the execution of LDDIR and LDPTE instructions will come from CSR.TLBRBADV.

Bit	Name reading	and writing	describe
1	0		R is always 0 when read-only, and writes are ignored.
GRLEN-1:2	PC	RW	Record the [GRLEN-1:2] bits of the PC that triggered the TLB refill exception. This occurs when the ERTN instruction is executed to refill from the TLB. When the exception handler returns (at this time, IsTLBR in this register = 1 and CSR.ERRCTL.IsMERR = 0), the hardware automatically... The value stored here will be padded with two 0 bits at its least significant bit and used as the final return address.

7.5.14 TLB Refill Exception Data Saving (TLBRSAVE)

This register is used to temporarily store data for system software. Each data storage register can hold the data of one general-purpose register.

The reason for setting up an additional SAVE register for the TLB refill exception handler is to handle non-TLB refill exceptions.

The TLB refill exception is triggered during the process.

Table **7-36 TLB** Refill Exception Data Storage Register Definitions

Bit	Name reading	and writing	describe
GRLEN-1:0			Data RW is data that can only be read and written by software. The hardware will not modify the contents of this field except when executing CSR instructions.

7.5.15 TLB Refill Exception Entries Low Bits (TLBRELO0, TLBRELO1)

The TLBRELO0/1 registers are used when the TLB is in a TLB refill exception context (when CSR.TLBRERA.IsTLBR=1), to store TLB pointers.

During operation, the lower-order bits of the TLB entry contain information such as the physical page number. The format of the TLBRELO0/1 registers and the meaning of each field are respectively...

The TLBELO0/1 registers are identical.

However, the TLBRELO0/1 registers are not a complete replica of the TLBRELO0/1 registers when CSR.TLBRERA.IsTLBR=1.

This is reflected in two points:

ÿ Regardless of the value of CSR.TLBRERA.IsTLBR, executing the TLBRD instruction will only update the TLBELO0/1 registers.

ÿ Regardless of the value of CSR.TLBRERA.IsTLBR, executing the LDPTE instruction will only update the TLBRELO0/1 registers.

Table **7-37 TLB** Refill Exception Entries Low-order Register Definitions (LA64 Architecture)

Bit	Name reading	and writing	describe
0	In		The valid bit (V) of the RW page table entry.
1	D		Dirty position (D) of RW page table entries.
3:2	POS		Privilege Level (PLV) of RW page entries.
5:4	ALONG WITH		Storage Access Type (MAT) for RW page table entries.
6	G	RW	Global flags (G) for page table entries. When executing the TLBFILL and TLBWR instructions, the fill is only performed if the G bits in both TLBELO0 and TLBELO1 are 1. The G bit in the page table entry in the TLB is 1.
11:7	0		R is always 0 when read-only, and writes are ignored.
POLES-1:12	PPN		Physical page number (PPN) of the RW page table.

Bit	Name reading	and writing	describe
60:POLES	0		R is always 0 when read-only, and writes are ignored.
61	No.		Unreadable bit (NR) of RW page table entries.
62	NX		The non-executable bit (NX) of the RW page table entry.
63	RPLV	RW	Restricted Privilege Level (RPLV) enable for page tables. When RPLV=0, the page table entry can be accessed by any privilege level. Programs with a privilege level of at least PLV can access this page table entry; when RPLV=1, this page table entry can only be accessed by programs with a privilege level equal to PLV. access.

Table **7-38 TLB** Refill Exception Entries Low-order Register Definitions (LA32 Architecture)

Bit	Name reading	and writing	describe
0	In		The valid bit (V) of the RW page table entry.
1	D		Dirty position (D) of RW page table entries.
3:2	POS		Privilege Level (PLV) of RW page entries.
5:4	ALONG WITH		Storage Access Type (MAT) for RW page table entries.
6	G	RW	Global flags (G) for page table entries. When executing the TLBFILL and TLBWR instructions, the fill is only performed if the G bits in both TLBELO0 and TLBELO1 are 1. The G bit in the page table entry in the TLB is 1.
7	0		R is always 0 when read-only, and writes are ignored.
POLES-5:8	PPN		Physical page number (PPN) of the RW page table.
31:POLES-4	0		R is always 0 for read-only operations; write operations are ignored. This field does not exist when PALEN=36.

7.5.16 TLB Refill Exception Entries High Bit (TLBREHI)

The TLBREHI register stores TLB instruction operations when the system is in a TLB refill exception context (where CSR.TLBRERA.IsTLBR=1).

The lower-order bits of the TLB entry contain information such as the physical page number. The format of the TLBREHI register and the meaning of each field are related to the TLBREHI register.

It's the same as a memory.

However, the TLBREHI register is not a complete replica of the TLBREHI register when CSR.TLBRERA.IsTLBR=1. This is reflected in:

ÿ Regardless of the value of CSR.TLBRERA.IsTLBR, executing the TLBRD instruction will only update the TLBEHI register.

Table **7-39 TLB** Refill Exception Page Table High-order Register Definitions (LA64 Architecture)

Bit	Name reading	and writing	describe
5:0	PS	RW	TLB refills the page size value for exceptions. Specifically, when CSR.TLBRERA.IsTLBR=1, TLBWR and... The TLBFILL instruction writes the value of the PS field of the TLB entry from this instruction.
12:6	0		R is always 0 when read-only, and writes are ignored.

Bit	Name reading	and writing	describe
VALEN-1:13	VPPN RW		When CSR.TLBRERA.IsTLBR=1, the TLBSRCH instruction queries the VPPN value used by the TLB, and executes... The value of the VPPN field written to the TLB entry when executing the TLBWR and TLBFILL instructions comes from this. When a TLB refill exception is triggered, the [VALEN-1:13] bits of the virtual address that triggered the exception are recorded here.
63:VALEN	Sign_Ext	The R read return value is a sign extension of the highest bit of the VPPN field; these bits are ignored when writing.	

Table **7-40 TLB** Refill Exception Page Table High-order Register Definitions (LA32 Architecture)

Bit	Name reading	and writing	describe
5:0	PS	RW	TLB refills the page size value for exceptions. Specifically, when CSR.TLBRERA.IsTLBR=1, TLBWR and... The TLBFILL instruction writes the value of the PS field of the TLB entry from this instruction.
12:6	0	R is always 0 when read-only, and writes are ignored.	
31:13	VPPN RW		When CSR.TLBRERA.IsTLBR=1, the TLBSRCH instruction queries the VPPN value used by the TLB, and executes... The value of the VPPN field written to the TLB entry when executing the TLBWR and TLBFILL instructions comes from this. When a TLB refill exception is triggered, bits [31:13] of the virtual address that triggered the exception are recorded here.

7.5.17 **TLB** Refill Exception Pre-Exception Schema Information (TLBRPRMD)

When a TLB refill exception is triggered, the hardware will update the processor core's privilege level, guest mode, global interrupt enable, and watchpoint settings at that time.

The bits are saved to this register and used to restore the processor core's state upon exception return.

Table **7-41 Definition of TLB** Refill Exception Mode Information Register

Bit	Name reading	and writing	describe
1:0	PPLV	RW	When a TLB refill exception is triggered, the hardware will record the old value of the PLV field in CSR.CRMD in this field. When CSR.TLBRERA.IsTLBR=1, the hardware will return from the exception handler when executing the ERTN instruction. The values of each field are restored to the PLV field of CSR.CRMD.
2	ABOUT	RW	When a TLB refill exception is triggered, the hardware will record the old value of the IE field in CSR.CRMD in this field. When CSR.TLBRERA.IsTLBR=1, the hardware will return from the exception handler when executing the ERTN instruction. The value of each field is restored to the IE field in CSR.CRMD.
3	0	If virtualization extensions are not implemented, this bit is always 0 for read-only mode, and writes are ignored.	
4	WEIGHT	RW	When a TLB refill exception is triggered, the hardware will record the old value of the WE field in CSR.CRMD in this field. When CSR.TLBRERA.IsTLBR=1, the hardware will return from the exception handler when executing the ERTN instruction. The value of each field is restored to the WE field of CSR.CRMD.
31:5	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

7.5.18 Direct Mapping Configuration Window (DMW0~DMW3)

This set of registers is involved in completing the direct-mapped address translation mode. For details on this address translation mode, please refer to Section 5.2.1.

Table **7-42** Direct Mapping Configuration Window Register Definitions (LA64 Architecture)

Bit	Name reading and writing		describe
0	PLV0		A value of 1 for RW indicates that the configuration of this window can be used for direct address mapping translation under privilege level PLV0.
1	PLV1		A value of 1 for RW indicates that the configuration of this window can be used for direct mapping address translation under privilege level PLV1.
2	PLV2		A value of 1 for RW indicates that the configuration of this window can be used for direct mapping address translation under privilege level PLV2.
3	PLV3		A value of 1 for RW indicates that the configuration of this window can be used for direct mapping address translation under privilege level PLV3.
5:4	ALONG WITH		The RW virtual address is the memory access type of the memory access operation that falls under this mapping window.
59:6	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.
63:60	VSEG		RW directly maps bits [63:60] of the virtual address of the window.

Table **7-43** Direct Mapping Configuration Window Register Definitions (LA32 Architecture)

Bit	Name reading and writing		describe
0	PLV0		A value of 1 for RW indicates that the configuration of this window can be used for direct address mapping translation under privilege level PLV0.
1	PLV1		A value of 1 for RW indicates that the configuration of this window can be used for direct mapping address translation under privilege level PLV1.
2	PLV2		A value of 1 for RW indicates that the configuration of this window can be used for direct mapping address translation under privilege level PLV2.
3	PLV3		A value of 1 for RW indicates that the configuration of this window can be used for direct mapping address translation under privilege level PLV3.
5:4	ALONG WITH		The RW virtual address is the memory access type of the memory access operation that falls under this mapping window.
24:6	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.
27:25	PSEG		RW directly maps the physical address of the window in bits [31:29].
28	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.
31:29	VSEG		RW directly maps the virtual address of the window in bits [31:29].

7.6 Timer-related control status register

7.6.1 Timer Number (TID)

Each timer in the processor has a unique, identifiable number, configured in a register by software. Each timer is also unique.

For each corresponding timer, when the software uses the RDTIME instruction to read the timer value, the timer ID number returned is also the corresponding timer ID.

The timer number.

Table **7-44** Timer Number Register Definition

Bit	Name reading and writing		describe
31:0	TIME	RW	Timer number. Software configurable. During processor core reset, hardware can reset it to the value in CSR.CPUID. The same value for CoreID.

7.6.2 Timer Configuration (TCFG)

This register is the interface for configuring the timer in software. The effective number of bits for the timer is determined by the implementation, therefore the bit width of the TimeVal field in this register is...

It will also change accordingly.

Table **7-45** Timer Configuration Register Definitions

Bit	Name reading and writing		describe
0	In	RW	Timer enable bit. The timer will only count down when this bit is 1, and will be reset when it reaches 0. Timer interrupt signal.
1	Periodic RW		Timer cycle mode control bit. If this bit is 1, a timer interrupt will be set when the timer counts down to 0. Simultaneously with the signal, the timer will automatically reload to the initial value configured in the InitVal field, and then proceed to the next... The clock cycle continues to decrement. If this bit is 0, the timer will stop counting when it reaches 0, until the software... Configure the timer again.
n-1:2	InitVal RW		The initial value for the timer's countdown decrement. This initial value must be an integer multiple of 4. The hardware will automatically set this value. The least significant bit of the field value is padded with two 0 bits before it is used.
GRLEN-1:n	0	R is always 0 when read-only, and writes are ignored.	

7.6.3 Timer Value (TVAL)

The software can read this register to determine the current timer count. The effective number of bits for the timer is determined by the implementation, therefore this register...

The bit width of the TimeVal field will also change accordingly.

Table **7-46** Timer Remaining Register Definitions

Bit	Name reading and writing		describe
n-1:0	TimeVal	R is the current timer count value.	
GRLEN-1:n	0	R is always 0 when read-only, and writes are ignored.	

7.6.4 Timer Compensation (CNTC)

The software can configure this register to correct the timer's read value. The final read value is: the original timer count value + the countdown.

The timer compensation value. Note that configuring this register does not directly change the timer's count value.

In the LA32 architecture, this register is 32 bits, and its value is sign-extended to 64 bits before being added to the original counter value.

Table **7-47** Timer Compensation Register Definition

Bit	Name reading and writing		describe
GRLEN-1:0	Compensation RW software provides configurable timer compensation values.		

7.6.5 Timer Interrupt Clearing (TICLR)

The software clears the timer interrupt signal that the timer was set by writing 1 to bit 0 of the register.

Table **7-48** Timer Interrupt Clear Register Definitions

Bit	Name reading and writing		describe
0	CLR		When a value of 1 is written to this bit, the clock interrupt flag will be cleared. The register will always read a value of 0.
31:1	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.7 RAS Related Control Status Register

7.7.1 Machine Error Control (MERRCTL)

Because the timing of machine error exceptions cannot be predicted or controlled by the software, in order to prevent any other existing systems from being damaged when a machine error exception is triggered... In this context, a separate set of Conditional Records (CSRs) is defined specifically for machine error exceptions, allowing the system software to save and restore other contexts. This set of separate CSRs... Apart from MERRERA and ERRSAVE, the rest are concentrated in the MERRCTL register.

Table **7-49** Machine Error Control Register Definitions

Bit	Name reading and writing		describe
0	IsMERR	R	A value of 1 indicates that the current context is machine error exception handling. When a machine error exception is triggered, the hardware sets this bit to 1. When this bit is 1, executing the ERTN instruction will clear it to 0. Because the architecture defines a separate CSR for machine error exceptions, when this bit is 1, When ERTN returns, the information used to restore CSR.CRMD will come from PPLV, PIE, etc. in this register. domain; The ERTN return address information will come from CSR.ERRERA.
1	Repairable	R	A value of 1 indicates that the hardware can automatically repair the machine error, therefore the exception handler can proceed without any processing. Return.
3:2	PPLV	RW	When a machine error exception is triggered, the hardware will record the old value of the PLV field in CSR.CRMD in this field. When IsMERR in this register is 1, the hardware will return from the exception handler when the ERTN instruction is executed. The values of each field are restored to the PLV field of CSR.CRMD.
4	ABOUT	RW	When a machine error exception is triggered, the hardware records the old value of the IE field in CSR.CRMD in this field. When IsMERR in this register is 1, the hardware will return from the exception handler when the ERTN instruction is executed. The value of each field is restored to the IE field in CSR.CRMD.
5	0		If virtualization extensions are not implemented, this bit is always 0 for read-only mode, and writes are ignored.

Bit	Name reading and writing		describe
6	WEIGHT	RW	When a machine error exception is triggered, the hardware will record the old value of the WE field in CSR.CRMD in this field. When IsMERR in this register is 1, the hardware will return from the exception handler when the ERTN instruction is executed. The value of each field is restored to the WE field of CSR.CRMD.
7	PDA	RW	When a machine error exception is triggered, the hardware will record the old value of the DA field in CSR.CRMD in this field. When IsMERR in this register is 1, the hardware will return from the exception handler when the ERTN instruction is executed. The values of each field are restored to the DA field of CSR.CRMD.
8	PPG	RW	When a machine error exception is triggered, the hardware will record the old value of the PG field in CSR.CRMD in this field. When IsMERR in this register is 1, the hardware will return from the exception handler when the ERTN instruction is executed. The value of each field is restored to the PG field of CSR.CRMD.
10:9	PDATF	RW	When a machine error exception is triggered, the hardware will record the old value of the DATF field in CSR.CRMD in this field. When IsMERR in this register is 1, the hardware will return from the exception handler when the ERTN instruction is executed. The values of each field are restored to the DATF field of CSR.CRMD.
12:11	PDATM	RW	When a machine error exception is triggered, the hardware will record the old value of the DATM field in CSR.CRMD in this field. When IsMERR in this register is 1, the hardware will return from the exception handler when the ERTN instruction is executed. The values of each field are restored to the DATM field of CSR.CRMD.
15:13	0	R0	Reserved returns 0 when read, and must be written to 0, or can be masked using CSR mask write.
23:16	Cause	R	Machine error type encoding. Currently, only the value 0x1 is defined to indicate a cache check error. The remaining encoded values are reserved.
31:24	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.7.2 Machine Error Message **1/2 (MERRINFO1/2)**

When a machine error exception is triggered, the hardware stores more information related to the error into these two registers for system software diagnostics. Its format and the specific meaning of each field are defined by the specific implementation.

7.7.3 Machine Error Exception Entry Address (MERRENTRY)

This register is used to configure the entry address for machine error exceptions. Because after a machine error exception is triggered, the processor core will enter the direct address... Since this is a translation mode, the entry address entered here should be a physical address.

Table **7-50** Machine Error Exception Entry Base Address Register Definitions (LA64 Architecture)

Bit	Name reading and writing		describe
11:0	0	R	machine error exception entry address [11:0] bits. Always 0 for read-only, write is ignored.
POLES-1:12	PPN	RW	Machine Error Exception Entry Address [PALEN-1:12] bits. The address entered here should be a physical address.
63:POLES	0	R	is always 0 when read-only, and writes are ignored.

Table **7-51** Machine Error Exception Entry Base Address Register Definitions (LA32 Architecture)

Bit	Name	reading and writing	describe
11:0	0	R	machine error exception entry address [11:0] bits. Always 0 for read-only, write is ignored.
31:12	PPN	RW	Machine Error Exception Entry Address [31:12]. The address entered here should be a physical address.

7.7.4 Machine Error Exception Return Address (MERRERA)

This register is used to record the return address after machine error exception handling is completed.

Table **7-52** Machine Error Exception Program Counter Register Definitions

Bit	Name	reading and writing	describe
GRLEN-1:0	PC	RW	The PC records the instruction that triggered the machine error exception. When the ERTN instruction is executed, it returns from the machine error exception handler. When returning (at this time CSR.ERRCTL.IsMERR=1), the value stored here will be used as the return address.

7.7.5 Machine Error Exception Data Saving (MERRSAVE)

This register is used to temporarily store data for system software. Each data storage register can hold the data of one general-purpose register.

The reason for setting up an additional SAVE register for use by the machine error exception handler is due to the timing of machine error exceptions. The software cannot predict or control this; it could happen during any other exceptional processing.

Table **7-53** Machine Error Exception Data Saving Register Definitions

Bit	Name	reading and writing	describe
GRLEN-1:0	Data	RW	is data that can only be read and written by software. The hardware will not modify the contents of this field except when executing CSR instructions.

7.8 Performance monitoring related control status registers

The Dragon architecture defines a hardware performance monitoring mechanism that supports software performance analysis. The core of this mechanism is a series of performance monitors. A minimum of one monitor and a maximum of 32 monitors can be implemented, the exact number depending on the implementation. The software can access this information by reading CPUCFG.6.PMNUM[bit7:4]. To determine how many performance monitors are available.

Each performance monitor contains two CSRs: a performance monitoring configuration register (PMCFG) and a performance monitoring counter (PMCNT). All performance monitoring-related configuration registers (CSRs) are alternately addressed starting from address 0x200. The address of the nth performance monitoring configuration register is 0x200+n, and the address of the nth CSR is... The address of each performance monitoring counter is 0x201+n. All performance monitoring configuration registers have the same format, as described in Section 7.8.1; all performance... The format of the monitoring counter is the same, as described in Section 7.8.2.

7.8.1 Performance Monitoring Configuration (PMCFG)

Table **7-54** Performance Monitoring Configuration Register Definitions

Bit	Name reading and writing		describe
9:0	EvCode	RW	The event number of the monitored performance event. The definition of the event number consists of two parts: one part is explicitly stated in the architecture specification. The meaning of the event numbers is that all processors compatible with this architecture must implement them; the meaning of the remaining event numbers is related to the specific implementation. The definition is left to the processor implementer.
15:10	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.
16	PLV0	RW	PLV0 privilege level enables counting for this performance monitor. 1 - Start counting, 0 - Stop counting.
17	PLV1	RW	PLV1 privilege level enables counting for this performance monitor. 1 - Start counting, 0 - Stop counting.
18	PLV2	RW	PLV2 privilege level enables counting for this performance monitor. 1 - Start counting, 0 - Stop counting.
19	PLV3	RW	PLV3 privilege level enables counting for this performance monitor. 1 - Start counting, 0 - Stop counting.
20	SMEs	RW	is the performance monitoring counter overflow interrupt enable bit for this performance monitor. 1 – Enable, 0 – Disable.
22:21	0		If virtualization extensions are not implemented in R, the read-only value of this field will always be 0, and writes will be ignored.
31:23	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.8.2 Performance Monitoring Counter (PMCNT)

Table **7-55** Performance Monitoring Counter Register Definitions

Bit	Name reading and writing		describe
GRLEN-1:0	Count RW		The counter increments by 1 each time a performance event monitored by the performance monitor occurs. If the performance monitor has enabled the performance monitor count overflow interrupt, then when the most significant bit of Count is 1, it will... This interrupt is invoked. This also means that the software can cancel the interrupt by clearing the most significant bit of Count to 0.

7.9 Monitoring Point Related Control Status Register

The Dragon architecture defines hardware watchpoint functionality for instruction fetch and load/store operations. After configuring the watchpoints for instruction fetch and load/store in the software, The processor hardware will monitor memory access addresses for instruction fetch and load/store operations, and trigger a watchpoint exception when the watchpoint settings are met.

The watchpoint-related control status register serves as the interface for software configuration of instruction fetching and load/store operations on the watchpoint. Load/store of the watchpoint and instruction fetching... The control status registers of each monitoring point have a similar layout, consisting of a register containing the overall configuration of all monitoring points and a register recording all monitoring data. The system includes a register for the status of each monitoring point, as well as four registers required for the individual configuration of each monitoring point. Among these, the overall configuration register for the load/store monitoring point is... The address is 0x300, the address of the overall status register of the load/store monitoring point is 0x301, and the four components numbered 1 to 4 of the nth load/store monitoring point... The addresses of the set registers are 0x310+8n, 0x311+8n, 0x312+8n, and 0x313+8n respectively; the address of the fetch watchpoint overall configuration register is... 0x380, the address of the overall status register of the instruction fetch watchpoint is 0x381, and the addresses of the four configuration registers (numbers 1-4) of the nth instruction fetch watchpoint are... The next numbers are 0x390+8n, 0x391+8n, 0x392+8n, and 0x393+8n.

The load/store watchpoint and fetch watchpoint each have a maximum of 14 implementations, with the actual number determined by the specific implementation. The software can read...

The values of CSR.MWPC.Num and CSR.FWPC.Num determine how many hardware monitoring points can be used.

7.9.1 Overall Configuration of **Load/Store** Monitoring Points (MWPC)

The configuration information contained in this register is used to tell the software the exact number of monitoring points to load/store.

It is worth noting that the global enable control signal for all monitoring points is the WE bit in CSR.CRMD.

Table 7-56 Overall Configuration Register Definitions for **Load/Store** Monitoring Points

Bit	Name	reading and writing	describe
5:0	In	R	load/store: The number of monitoring points.
19:16	0		If virtualization extensions are not implemented in R, the read-only value of this field will always be 0, and writes will be ignored.
31:20	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.9.2 Overall Status of **Load/Store** Monitoring Points (MWPS)

Table 7-57 Definition of Overall Status Register for **Load/Store** Monitoring Points

Bit	Name	reading and writing	describe
n-1:0	Status	RW1	The load/store status indicates the hit rate of the watchpoints. Each watchpoint corresponds one-to-one with a specific bit, with bit i corresponding to the i-th watchpoint. When an address involved in a load/store operation hits a watchpoint, its corresponding bit is set to 1. This applies except during reset. The hardware will not clear the bits of this field to 0. The software can only clear it to 0 by comparing it to close-up 1; close-up 0 will be ignored.
15:n	0	R	is always 0 when read-only, and writes are ignored.
16	Skip	RW	The software sets this bit to 1 to instruct the hardware to ignore the next load/store watchpoint hit result. This "ignore" refers to... "Abbreviated" means neither setting the corresponding bit in the Stauts field of this register to 1 nor triggering a watchpoint exception. This function can... This simplifies watchpoint exceptions by preventing the endless re-triggering of the same watchpoint without canceling it. The processing. When the Skip bit is 1, if the hardware encounters a load/store watchpoint hit, it will ignore that hit. The Skip bit is cleared to 0 at the same time as the result is obtained. This means that each time the software sets the Skip bit to 1, the hardware ignores at most one... The monitoring point was hit. This characteristic also means that the value read after writing a 1 to this bit may not necessarily be 1. This Skip bit corresponds to all load/store monitoring points. If the software modifies the breakpoint configuration or changes the breakpoint, Do not set this bit; in fact, for safety, you should write 0 to clear the bit.
31:17	0	R	is always 0 when read-only, and writes are ignored.

7.9.3 Configure 1~4 **load/store** monitoring points **n (MWPnCFG1~4)**

The information contained in registers 1-3 of the configuration for each load/store monitoring point is directly used for comparison and judgment during monitoring point checks. Assuming the points to be compared...

The address being operated on is maddr, and the byte range is mbyten. The hit determination process for each monitoring point is as follows:

1. If CSR.CRMD.WE=0, terminate the process; otherwise, go to step 2.

2. If the current state is not debug mode but the DSOnly bit of MWPCFG3 is equal to 1, terminate the process; otherwise, proceed to step 3.

3. If the bit corresponding to the current privilege level in PLV0~PLV3 of MWPCFG3 is equal to 0, the judgment ends; otherwise, go to 4.

4. If the operation is a load operation but the LoadEn bit in MWPCFG3 is equal to 0, or if the operation is a store operation but...

   If the StoreEn bit in MWPCFG3 is equal to 0, the judgment terminates; otherwise, go to step 5.

5. If the LCL bit in MWPCFG3 is equal to 1, but CSR.ASID.ASID is not equal to the ASID in MWPCFG4, the judgment terminates.

   Otherwise, go to 6;

6. If (maddr & (~MWPCFG2.Mask)) != (MWPCFG1.VAaddr & (~MWPCFG2.Mask)), meaning the addresses are not comparable.

   Wait, determine if to terminate; otherwise, go to step 7.

7. If (~bytemask[7:0] & mbyten[7:0]) equals all 0, the judgment terminates; otherwise, the watch point is considered to have been hit.

The following section provides further explanation of the concepts of mbyten and bytemask that appeared in the above judgment process description.

`mbyten` represents the bytes involved in the operation. It is an 8-bit bit vector whose value corresponds to the type of the load/store operation and the low-order address value.

The specific definitions are shown in Table 7-58:

Table **7-58 Load/Store** Monitoring Point Judgment Process **(mbyten** Definition)

Command Name	maddr[2:0]							
	0	1	2	3	4	5	6	7
LD[X].B[U], ST[X].B, LD{GT/LE}.B, ST{GT/LE}.B	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
LD[X].H[U], ST[X].H LD{GT/LE}.H, ST{GT/LE}.H	0x03		0x0C		0x30		0xC0	
LD[X].W[U], ST[X].W, LD{GT/LE}.W, ST{GT/LE}.W, LDPTR.W, STPTR.W, LL.W, SC.W, AM{SWAP/ADD/AND/OR/XOR/MAX/MIN}[_DB].W, AM{MAX/MIN}[_DB].WU, FLD[X].S, FST[X].S, FLD{GT/LE}.S, FST{GT/LE}.S	0x0F				0xF0			
LD[X].D, ST[X].D, LD{GT/LE}.D, ST{GT/LE}.D, LDPTR.D, STPTR.D, LL.D, SC.D, AM{SWAP/ADD/AND/OR/XOR/MAX/MIN}[_DB].D, AM{MAX/MIN}[_DB].DU, FLD[X].D, FST[X].D, FLD{GT/LE}.D, FST{GT/LE}.D	0xFF							

bytemask represents a mask that excludes bytes from the comparison during the watchpoint comparison. It is an 8-bit bit vector whose value is the same as MWPCFG1.

The lower bits of VAddr are related to Size in MWPCFG3, as shown in the specific definition.

Table **7-59 Load/Store** Monitoring Point **bytemask** Definition

MWPCFG3.Size	MWPCFG1.VAddr[2:0]							
	0	1	2	3	4	5	6	7
0b00	0x00							
0b01	0xF0				0x0F			
0b10	0xFC		0xF3		0xCF		0x3F	
0b11	0xFE	0xFD	0xFB	0xF7	0xEF	0xDF	0xBF	0x7F

Table **7-60 Load/Store** Watchpoint Configuration **1** Register Definitions

Bit	Name reading	and writing	describe
GRLEN-1:0	VAddr	RW	is the virtual address to be compared at the load/store monitoring point.

Table **7-61 Load/Store** Monitoring Point Configuration **2** Register Definitions

Bit	Name reading	and writing	describe
GRLEN-1:0	Mask	RW	The mask bit for comparing the load/store monitoring point address. If the i-th bit (0ÿi<GRLEN) is 1, it indicates that the address is... The i-th bit is not included in the comparison.

Table **7-62 Load/Store** Watchpoint Configuration **3** Register Definitions

Bit	Name reading	and writing	describe
0	DSOnly	RW	A value of 1 indicates that this load/store watchpoint is only available in debug mode. Here, "available" has two meanings: First, the configuration register of the monitoring point can be modified by software in this mode. Second, after the monitoring point checks for a hit... Only in this mode will the watchpoint exception be triggered and the watchpoint status be marked. This bit can only be modified in debug mode (CSR.DBG.DS=1). This means that when running in debug mode (on... The (positional machine) software has priority access to the monitoring points.
1	PLV0	RW	This monitor point enables monitor point exceptions at the PLV0 privilege level. 1 - Enable, 0 - Disable.
2	PLV1	RW	This monitor point enables monitor point exceptions under PLV1 privilege level. 1 - Enable, 0 - Disable.
3	PLV2	RW	This monitor point enables monitor point exceptions under the PLV2 privilege level. 1 - Enable, 0 - Disable.
4	PLV3	RW	This monitor point enables monitor point exceptions under PLV3 privilege level. 1 - Enable, 0 - Disable.
6:5	0		If virtualization extensions are not implemented in R, the read-only value of this field will always be 0, and writes will be ignored.
7	LCL		A value of 1 for RW indicates that this is a local monitoring point.
8	LoadEn	RW	= 1 indicates that the load operation is checked by a watchpoint; otherwise, it is not checked.
9	StoreEn		A value of 1 for RW indicates that a watchpoint check is performed on the store operation; otherwise, no check is performed.
11:10	Size	RW	When performs a watchpoint check, it determines which bytes fall within the comparison range.
31:12	0	R0	is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

Table **7-63 Load/Store** Watchpoint Configuration **4** Register Definitions

Bit	Name reading and writing		describe
9:0	ACID		RW is compared to ASID
15:10	0		R is always 0 when read-only, and writes are ignored.
23:16	0		If virtualization extensions are not implemented in R, the read-only value of this field will always be 0, and writes will be ignored.
31:24	0		R is always 0 when read-only, and writes are ignored.

7.9.4 Overall Configuration of Command-Based Monitoring Point (FWPC)

The configuration information contained in this register is used to inform the software of the exact number of instruction fetch watchpoints.

It is worth noting that the global enable control signal for all monitoring points is the WE bit in CSR.CRMD.

Table **7-64** Overall Configuration Register Definitions for Instruction Fetch Watchpoint

Bit	Name reading and writing		describe
5:0	In		R represents the number of monitoring points.
19:16	0		If virtualization extensions are not implemented in R, the read-only value of this field will always be 0, and writes will be ignored.
31:20	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

7.9.5 Overall Status of the Command Watchpoint (FWPS)

Table **7-65** Definition of the Overall Status Register for the Instruction Fetch Watchpoint

Bit	Name reading and writing		describe
n-1:0	Status	RW1	This indicates the hit status of the watchpoints. Each watchpoint corresponds one-to-one with the watchpoint, with bit i corresponding to the i-th watchpoint. When a pointer fetching PC hits a watchpoint, its corresponding bit is set to 1. Except during reset, the hardware will not... Clear the bits in this field to 0. The software can only clear it to 0 by comparing it to close-up 1; close-up 0 will be ignored.
15:n	0		R is always 0 when read-only, and writes are ignored.
16	Skip	RW	The software sets this bit to 1 to instruct the hardware to ignore the next fetch watchpoint hit result. Ignoring means... This means neither setting the corresponding bit in the Stauts field of this register to 1 nor triggering a watchpoint exception. This function can be used without... By canceling a watchpoint, the endless re-triggering of the same watchpoint is avoided, thus simplifying the handling of watchpoint exceptions. reason. When the Skip bit is 1, if the hardware encounters a fetch watchpoint hit, it will ignore the hit result. At the same time, the Skip bit is cleared to 0. This means that each time the software sets the Skip bit to 1, the hardware will ignore the monitoring at most once. Viewpoint hit. This characteristic also means that if the software writes a 1 to this bit, the value read back may not be 1. This Skip bit corresponds to all instruction fetch watchpoints. If the software modifies the breakpoint configuration or changes the breakpoint, then... To set this bit, or even to clear it by writing 0 for safety, you need to do so.
31:17	0		R is always 0 when read-only, and writes are ignored.

7.9.6 Configure **1~3 (FWPnCFG1~3)** for the command monitoring point **n .**

The information contained in registers 1-3 of each instruction fetch watchpoint is directly used for comparison and judgment during watchpoint checks. The hit rate of each watchpoint...

The judgment process is as follows:

1. If CSR.CRMD.WE=0, terminate the process; otherwise, go to step 2.

2. If the current state is not debug mode but the DSOnly bit of FWPCFG3 is equal to 1, terminate the process; otherwise, proceed to step 3.

3. If the bit corresponding to the current privilege level in PLV0~PLV3 of FWPCFG3 is equal to 0, the judgment ends; otherwise, go to 4.

4. If the LCL bit in FWPCFG3 is equal to 1, but CSR.ASID.ASID is not equal to the ASID in FWPCFG4, the process terminates.

   Otherwise, go to 6;

5. If (pc & (~FWPCFG2.Mask)) != (FWPCFG1.VAddr & (~FWPCFG2.Mask)), meaning the addresses are not equal, then determine...

   If the connection is broken, the operation terminates; otherwise, the monitoring point is considered to have been hit.

Table **7-66** Instruction Fetch Watchpoint Configuration **1** Register Definitions

Bit	Name reading	and writing	describe
GRLEN-1:0	VAddr	RW	indicates the virtual address to be compared at the monitoring point.

Table **7-67** Instruction Fetch Watchpoint Configuration **2** Register Definitions

Bit	Name reading	and writing	describe
GRLEN-1:0	Mask	RW	This refers to the mask bit used for comparing the address of the reference monitoring point. If the i-th bit (0ÿi<GRLEN) is 1, it indicates that the i-th bit of the address is... Not included in the comparison.

Table **7-68** Instruction Fetch Watchpoint Configuration **3** Register Definitions

Bit	Name reading and writing		describe
0	DSOnly	RW	A value of 1 indicates that this fetch watchpoint is only available in debug mode. Here, "available" has two meanings: one... First, the configuration register of the monitoring point can be modified by software in this mode; second, after the monitoring point checks for a hit, only... Only in this mode will the watchpoint exception be triggered and the watchpoint status be marked. This bit can only be modified in debug mode (CSR.DBG.DS=1). This means that when running in debug mode (on... The (positional machine) software has priority access to the monitoring points.
1	PLV0	RW	This monitor point enables monitor point exceptions at the PLV0 privilege level. 1 - Enable, 0 - Disable.
2	PLV1	RW	This monitor point enables monitor point exceptions under PLV1 privilege level. 1 - Enable, 0 - Disable.
3	PLV2	RW	This monitor point enables monitor point exceptions under the PLV2 privilege level. 1 - Enable, 0 - Disable.
4	PLV3	RW	This monitor point enables monitor point exceptions under PLV3 privilege level. 1 - Enable, 0 - Disable.
6:5	0		If virtualization extensions are not implemented in R, the read-only value of this field will always be 0, and writes will be ignored.
7	LCL		A value of 1 for RW indicates that this is a local monitoring point.
31:8	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

Table **7-69** Instruction Fetch Watchpoint Configuration **4** Register Definitions

Bit	Name reading and writing		describe
9:0	ACID		RW is compared to ASID
15:10	0		R is always 0 when read-only, and writes are ignored.
23:16	0		If virtualization extensions are not implemented in R, the read-only value of this field will always be 0, and writes will be ignored.
31:24	0		R is always 0 when read-only, and writes are ignored.

7.10 Debugging related control status registers

7.10.1 Debug Register (DBG)

Table **7-70** Debug Register Definitions

Bit	Name reading and writing		describe
0	DS	R	A value of 1 indicates that the system is currently in debug mode. When a debug exception is triggered in non-debug mode, the hardware sets this bit to 1. When the bit is 1, the ERTN instruction is executed to clear the bit to 0.
7:1	DRev		The version number of R's debugging mechanism. 1 is the initial version.
8	OF THE		R = 1 indicates that the debug exception type for entering debug mode is debug external interrupt exception (DEI).
9	DCL		R = 1 indicates that the debug exception type that enters debug mode is the debug call exception (DCL).
10	DFW		R = 1 indicates that the debug exception type for entering debug mode is the Debug Fetch Watchpoint Exception (DFW).
11	DMW		R = 1 indicates that the debug exception type that caused the user to enter debug mode is the debug load/store watchpoint exception (DMW).
15:12	0		R0 is read-only (0)
21:16	Ecode	R	When a non-debug exception occurs in debug mode, the exception type code is recorded here. The meaning of the codes here is the same as in the table. The definitions in 7-8 are basically the same, with only three differences: The TLB refill exception reused exception code 0x7; The debug call exception reused exception code 0xC; The machine error exception reused exception code 0xE.
31:22	0		R0 is read-only (0)

7.10.2 Debug Exception Return Address (DERA)

Table **7-71** Definition of Debug Exception Return Address Register

Bit	Name reading and	writing	describe
63:0	PC	RW	When a debug exception is triggered in non-debug mode, the hardware will record the PC that triggered the exception here. When CSR.DBG.DS=1, the return address is retrieved from here when the ERTN instruction is executed.

7.10.3 Debug Data Saving (DSAVE)

This register is used to temporarily store data for system software. Each data storage register can hold the data of one general-purpose register.

The reason for setting up an additional SAVE register for the debug exception handler is that debug exceptions can occur in any scenario.

Furthermore, the handling of debugging exceptions should be transparent to the software on the host being debugged.

Table **7-72** Definitions of Debug Data Storage Registers

Bit	Name reading and	writing	describe
63:0	Data	RW is data	that is only available for software to read and write. The hardware will not modify the contents of this field except when executing CSR instructions.

7.11 Message Interrupt Related Control Status Register

7.11.1 Message interruption status **0ÿ3 (MSGIS0~3)**

Table **7-73** Definition of Message Interrupt Status Register **0**

Bit	Name reading and	writing	describe
63:0	IS	R records	the interrupt status of messages 0 to 63 sequentially from bit 0 to bit 63. A value of 1 indicates that the interrupt for that message has been enabled.

Table **7-74** Definition of Message Interrupt Status Register **1**

Bit	Name reading and	writing	describe
63:0	IS	R records	the interrupt status of messages 64 to 127 sequentially from bits 0 to 63. A value of 1 indicates that the interrupt for that message has been enabled.

Table **7-75** Definitions of Message Interrupt Status Register **2**

Bit	Name reading and	writing	describe
63:0	IS	R records	the interrupt status of messages 128 to 191 sequentially from bits 0 to 63. A value of 1 indicates that the interrupt for that message has been enabled.

Table **7-76** Definition of Message Interrupt Status Register **3**

Bit	Name reading and	writing	describe
63:0	IS	R records	the interrupt status of messages 192 to 255 sequentially from bits 0 to 63. A value of 1 indicates that the interrupt for that message has been enabled.

7.11.2 Message Interruption Request (MSGIR)

This register is a 32-bit CSR. When accessed using the CSR instruction in the LA64 architecture, it is sign-extended to 64 bits before being returned, therefore...

Regardless of whether it is an LA32 or LA64 architecture, a negative return value indicates that there is no valid message interrupt request in the register.

Table **7-77** Message Interrupt Request Register Definitions

Bit	Name reading and writing		describe
7:0	IntNum	R	is the message interrupt number that initiated the message interrupt request. This field is only meaningful when the Null bit of this register is 0.
30:8	0	R0	is read-only (0)
31	Null	R	= 1 indicates that there is no valid message interruption request at present, otherwise it is 0.

7.11.3 Message Interrupt Enable (MSGIE)

Table **7-78** Message Interrupt Enable Register Definitions

Bit	Name reading and writing		describe
7:0	PT	R/W	Priority Threshold. Enables the lower priority threshold for message interruption.
31:8	0	R0	is read-only (0)

8 Appendix A Functional Definition Pseudocode Description

8.1 Operator Interpretation in Pseudocode

This section lists the meanings of statement keywords and various operators involved in pseudocode, as well as the operator precedence relationships.

In addition, the common conventions for representing numerical values in pseudocode are as follows:

ÿ Decimal numbers are represented without a prefix or with a prefix of "'d" or "##'d", where the prefix "##'d" indicates that the bit width of this decimal number is ##.

Bit;

ÿ Use the prefix "'b" or "##'b" to represent binary numbers, where the prefix "##'b" indicates that the bit width of this binary number is ## bits;

ÿ Use the prefix "'h" or "##'h" to represent hexadecimal numbers, where the prefix "##'h" indicates that the bit width of this hexadecimal number is ## bits.

In hexadecimal numbers, A through F are written in uppercase.

Table 8-1 Explanation of Key Words in Statements

Operators	meaning
Return type function name (variable , …): function body return Return value	Function definition
if Condition 1 Execute Statement 1 Elif Judgment conditio2 Execute statement 2 else: Execute statement 3	conditional statements
case Determine the variable of: Value1 Execute statement 1 Value2 Execute statement 2 default: Default execution statement	case conditional statement
Judgment conditions ? TRUE executes the statement. FALSE statement	Conditional statements
for loop variable in sequence Execution statement	for loop statement
range() N	A sequence of integers from 0 to N-1 with a step size of 1.
range(Start value End value Step value)	A sequence of specified step values from the start value (inclusive) to the end value (exclusive).
break	Abort the current loop
signed(…)	Signed integers
unsigned(…)	unsigned integers
fp16(…)	half-precision floating-point number
fp32(…)	Single-precision floating-point number
fp64(…)	Double-precision floating-point numbers

Operators	meaning
boolean	Boolean type
bit	Bit type
integer	Integer type
bits() N	N-bit type
ZeroExtend(variable, N)	Variable zero-extended to N bits
SignExtend(variable, N)	Variable sign extended to N bits
isSNaN(variable)	The value is TRUE if the variable is a signaling NaN number, and FALSE otherwise.
isQNaN(variable)	The value is TRUE if the variable is a quiet NaN, otherwise it is FALSE.
SignalException(exception)	Triggering exceptions
#	Single-line comment
=	Assignment

Table 8-2 Explanation of String Operators

Operators	meaning
[M :N]	N to M bits of the bit string
{N {M }}	Bit string M is copied N times and concatenated
{N , M , ... }	The bit strings N, M, ... are concatenated in sequence

Table 8-3 Definitions of Arithmetic Operators

Operators	meaning
+	add
-	reduce
*	take
/	remove
%	Mold taking
**	power

Table 8-4 Explanation of Comparison Operators

Operators	meaning
==	equal
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Table 8-5 : Definitions of Bitwise Operators

Operators	meaning
&	Bitwise AND
|	Bitwise or
^	bitwise XOR
~	Invert bitwise
<<	Logical left shift
>>	Logical right shift
>>>	Arithmetic right shift

Table 8-6 Explanation of Logical Operators

Operators	meaning
and	Logic AND
or	Logical OR
not	Logical NOT

The operator precedence in pseudocode, from highest to lowest, is listed in Table 8-7:

Table 8-7 Operator Precedence

Operators	meaning
**	power
~	Invert bitwise
*, /, %	Multiplication, division, modulo
+, -	Add, subtract
<<, >>, >>>	Logical left shift, logical right shift, arithmetic right shift
&	Bitwise AND
^, |	Bitwise XOR, Bitwise OR
>, <, >=, <=	Greater than, less than, greater than or equal to, less than or equal to
==, !=	Equal to, not equal to
not	Logical NOT
and, or	Logical AND, Logical OR

8.2 Pseudocode Description of Functions

The pseudocode definitions used in the instruction descriptions in this manual are as follows.

8.2.1 Logical Left Shift

```
bits(N) SLL(bits(N) x, integer sa):
    if sa==0 :
        result = x
    else :
        result = {x[N-sa-1:0], {sa{1'b0}}}
    return result
```

8.2.2 Logical Right Shift

```
bits(N) SRL(bits(N) x, integer sa):
    if sa==0 :
        result = x
    else :
        result = {{sa{1'b0}}, x[N-1:sa]}
    return result
```

8.2.3 Arithmetic right shift

```
bits(N) SRA(bits(N) x, integer sa):
    if sa==0 :
        result = x
    else :
        result = {{in{x[N-1]}}, x[N-1:in]}
    return result
```

8.2.4 Cyclic Right Shift

```
bits(N) ROTR(bits(N) x, integer sa):
    if sa==0 :
        result = x
    else :
        result = {x[sa-1:0], x[N-1:sa]}
    return result
```

8.2.5 Count the number of consecutive **1s** starting from the highest digit.

```
{bits(N) } CLO(bits(N) x):
    cnt = 0
    for i in range(N) :
        if x[N-1-i]==1'b0 :
            return cnt
        else :
            cnt = cnt + 1
```

8.2.6 Count the number of consecutive **zeros** starting from the highest digit.

```
{bits(N) } CLZ(bits(N) x):
    cnt = 0
    for i in range(N) :
        if x[N-1-i]==1'b1 :
            return cnt
        else :
            cnt = cnt + 1
```

8.2.7 Count the number of consecutive **1s** starting from the least significant digit.

```
{bits(N) } CTO(bits(N) x):
    cnt = 0
    for i in range(N) :
        if x[i]==1'b0 :
            return cnt
        else :
            cnt = cnt + 1
```

8.2.8 Count the number of consecutive zeros starting from the least **significant bit .**

```
{bits(N) } CTZ(bits(N) x):
    cnt = 0
    for i in range(N) :
        if x[i]==1'b1 :
            return cnt
        else :
            cnt = cnt + 1
```

8.2.9 Bit string reversal

```
{bits(N) } BITREV(bits(N) x):
    for i in range(N) :
        res[i] = x[N-1-i]
    return res
```

8.2.10 CRC-32 checksum calculation

```
bits(32) CRC32(old_chksum, msg, width, poly):
    new_chksum = (old_chksum & 0xFFFFFFFF) ^ {{(64-width){1'b0}}, msg}
    for i in range(width):
        if (new_chksum & 1'b1):
            new_chksum = (new_chksum >> 1)                    ^   poly
        else:
            new_chksum = ((new_chksum >> 1)
    return new_chksum
```

8.2.11 Converting Single-Precision Floating-Point Numbers to Signed Word Integers

```
{bits(32) } FP32convertToSint32(bits(32) x, bits(2) rm):
    case {rm} of:
        {2'd0}: return Sint32_convertToIntegerExactTiesToEven(x)
        {2'd1}: return Sint32_convertToIntegerExactTowardZero(x)
        {2'd2}: return Sint32_convertToIntegerExactTowardPositive(x)
        {2'd3}: return Sint32_convertToIntegerExactTowardNegative(x)
```

8.2.12 Converting Single-Precision Floating-Point Numbers to Signed Double-Word Integers

```
{bits(64) } FP32convertToSint64(bits(32) x, bits(2) rm):
    case {rm} of:
        {2'd0}: return Sint64_convertToIntegerExactTiesToEven(x)
        {2'd1}: return Sint64_convertToIntegerExactTowardZero(x)
        {2'd2}: return Sint64_convertToIntegerExactTowardPositive(x)
        {2'd3}: return Sint64_convertToIntegerExactTowardNegative(x)
```

8.2.13 Converting Double-Precision Floating-Point Numbers to Signed Word Integers

```
{bits(32) } FP64convertToSint32(bits(64) x, bits(2) rm):
    case {rm} of:
        {2'd0}: return Sint32_convertToIntegerExactTiesToEven(x)
        {2'd1}: return Sint32_convertToIntegerExactTowardZero(x)
        {2'd2}: return Sint32_convertToIntegerExactTowardPositive(x)
        {2'd3}: return Sint32_convertToIntegerExactTowardNegative(x)
```

8.2.14 Converting Double-Precision Floating-Point Numbers to Signed Double-Word Integers

```
{bits(64) } FP64convertToSint64(bits(64) x, bits(2) rm):
    case {rm} of:
        {2'd0}: return Sint64_convertToIntegerExactTiesToEven(x)
        {2'd1}: return Sint64_convertToIntegerExactTowardZero(x)
        {2'd2}: return Sint64_convertToIntegerExactTowardPositive(x)
        {2'd3}: return Sint64_convertToIntegerExactTowardNegative(x)
```

8.2.15 Rounding Single-Precision Floating-Point Numbers

```
{bits(32) } FP32_roundToInteger(bits(N) x):
    return FP32_roundToIntegralExact(x)
```

8.2.16 Rounding Double-Precision Floating-Point Numbers

```
{bits(64) } FP64_roundToInteger(bits(N) x):
    return FP64_roundToIntegralExact(x)
```

9. Appendix B : List of Instruction Codes

		31 30 29 28 27 26 25 24 23 22 21 20 19 18	17	16 15	14 13 12 11 10	09 08 07 06 05	04 03 02 01 00
CLO.W	rd, rj	0000000000000000000100				rj	rd
CLZ.W	rd, rj	0000000000000000000101				rj	rd
CTO.W	rd, rj	0000000000000000000110				rj	rd
CTZ.W	rd, rj	0000000000000000000111				rj	rd
CLO.D	rd, rj	0000000000000000001000				rj	rd
CLZ.D	rd, rj	0000000000000000001001				rj	rd
CTO.D	rd, rj	0000000000000000001010				rj	rd
CTZ.D	rd, rj	0000000000000000001011				rj	rd
REVB.2H	rd, rj	0000000000000000001100				rj	rd
REVB.4H	rd, rj	0000000000000000001101				rj	rd
REVB.2W	rd, rj	0000000000000000001110				rj	rd
REVB.D	rd, rj	0000000000000000001111				rj	rd
REVH.2W	rd, rj	0000000000000000010000				rj	rd
REVH.D	rd, rj	0000000000000000010001				rj	rd
BITREV.4B	rd, rj	0000000000000000010010				rj	rd
BITREV.8B	rd, rj	0000000000000000010011				rj	rd
BITREV.W	rd, rj	0000000000000000010100				rj	rd
BITREV.D	rd, rj	0000000000000000010101				rj	rd
EXT.W.H	rd, rj	0000000000000000010110				rj	rd
EXT.W.B	rd, rj	0000000000000000010111				rj	rd
RDTIMEL.W rd, rj		0000000000000000011000				rj	rd
RDTIMEH.W rd, rj		0000000000000000011001				rj	rd
RDTIME.D	rd, rj	0000000000000000011010				rj	rd
CPUCFG	rd, rj	0000000000000000011011				rj	rd
ASRTLE.D	rj, rk	00000000000000010			rk	rj	00000
ASRTGT.D	rj, rk	00000000000000011			rk	rj	00000
ALSL.W	rd, rj, rk, sa2	00000000000010 sa2			rk	rj	rd
ALSL.WU	rd, rj, rk, sa2	00000000000011 sa2			rk	rj	rd
BYTEPICK.W rd, rj, rk, sa2		0000000000100 sa2			rk	rj	rd
BYTEPICK.D rd, rj, rk, sa3		0000000000011	sa3		rk	rj	rd
ADD.W	rd, rj, rk	00000000000100000			rk	rj	rd
ADD.D	rd, rj, rk	00000000000100001			rk	rj	rd
SUB.W	rd, rj, rk	00000000000100010			rk	rj	rd
SUB.D	rd, rj, rk	00000000000100011			rk	rj	rd
SLT	rd, rj, rk	00000000000100100			rk	rj	rd
SLTU	rd, rj, rk	00000000000100101			rk	rj	rd
MASKEQZ	rd, rj, rk	00000000000100110			rk	rj	rd

<table>
<tr><th rowspan="2"></th><th>3</th><th>3</th><th>2</th><th>2</th><th>2</th><th>2</th><th>2</th><th>2</th><th>2</th><th>2</th><th>2</th><th>2</th><th>1</th><th>1</th><th>1</th><th>1</th><th>1</th><th>1</th><th>1</th><th>1</th><th>1</th><th>1</th><th>0</th><th>0</th><th>0</th><th>0</th><th>0</th><th>0</th><th>0</th><th>0</th><th>0</th><th>0</th></tr>
<tr><th>1</th><th>0</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>MASKNEZ rd, rj, rk</td><td colspan="17">00000000000100111</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>NOR rd, rj, rk</td><td colspan="17">00000000000101000</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>AND rd, rj, rk</td><td colspan="17">00000000000101001</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>OR rd, rj, rk</td><td colspan="17">00000000000101010</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>FREE rd, rj, rk</td><td colspan="17">00000000000101011</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>ORN rd, rj, rk</td><td colspan="17">00000000000101100</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>ANDN rd, rj, rk</td><td colspan="17">00000000000101101</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>SLL.W rd, rj, rk</td><td colspan="17">00000000000101110</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>SRL.W rd, rj, rk</td><td colspan="17">00000000000101111</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>SRA.W rd, rj, rk</td><td colspan="17">00000000000110000</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>SLL.D rd, rj, rk</td><td colspan="17">00000000000110001</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>SRL.D rd, rj, rk</td><td colspan="17">00000000000110010</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>SRA.D rd, rj, rk</td><td colspan="17">00000000000110011</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>ROTR.W rd, rj, rk</td><td colspan="17">00000000000110110</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>ROTR.D rd, rj, rk</td><td colspan="17">00000000000110111</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MUL.W rd, rj, rk</td><td colspan="17">00000000000111000</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MULH.W rd, rj, rk</td><td colspan="17">00000000000111001</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MULH.WU rd, rj, rk</td><td colspan="17">00000000000111010</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MUL.D rd, rj, rk</td><td colspan="17">00000000000111011</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MULH.D rd, rj, rk</td><td colspan="17">00000000000111100</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MULH.DU rd, rj, rk</td><td colspan="17">00000000000111101</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MULW.DW rd, rj, rk</td><td colspan="17">00000000000111110</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MULW.D.WU rd, rj, rk</td><td colspan="17">00000000000111111</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>DIV.W rd, rj, rk</td><td colspan="17">00000000001000000</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MOD.W rd, rj, rk</td><td colspan="17">00000000001000001</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>DIV.WU rd, rj, rk</td><td colspan="17">00000000001000010</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MOD.WU rd, rj, rk</td><td colspan="17">00000000001000011</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>DIV.D rd, rj, rk</td><td colspan="17">00000000001000100</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MOD.D rd, rj, rk</td><td colspan="17">00000000001000101</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>DIV.YOU rd, rj, rk</td><td colspan="17">00000000001000110</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>MOD.YOU rd, rj, rk</td><td colspan="17">00000000001000111</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>CRC.WBW rd, rj, rk</td><td colspan="17">00000000001001000</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>CRC.WHW rd, rj, rk</td><td colspan="17">00000000001001001</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>CRC.W.W.W rd, rj, rk</td><td colspan="17">00000000001001010</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>CRC.WDW rd, rj, rk</td><td colspan="17">00000000001001011</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>CRCC.W.B.W rd, rj, rk</td><td colspan="17">00000000001001100</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>CRCC.WHW rd, rj, rk</td><td colspan="17">00000000001001101</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>CRCC.W.W.W rd, rj, rk</td><td colspan="17">00000000001001110</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>CRCC.W.D.W rd, rj, rk</td><td colspan="17">00000000001001111</td><td colspan="5">rk</td><td colspan="5">rj</td><td colspan="5">rd</td></tr>
<tr><td>BREAK code</td><td colspan="17">00000000001010100</td><td colspan="15">code</td></tr>
<tr><td>DBCL code</td><td colspan="17">00000000001010101</td><td colspan="15">code</td></tr>
</table>

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SYSCALL	**code**	00000000001010110																	code														
ALSL.D	**rd, rj, rk, sa2**	000000000010110 sa2																	rk					rj					rd				
SLLI.W	**rd, rj, ui5**	00000000010000001																	ui5					rj					rd				
SLLI.D	**rd, rj, ui6**	0000000001000001																ui6						rj					rd				
SRLI.W	**rd, rj, ui5**	00000000010001001																	ui5					rj					rd				
SRLI.D	**rd, rj, ui6**	0000000001000101																ui6						rj					rd				
SRAI.W	**rd, rj, ui5**	00000000010010001																	ui5					rj					rd				
SRAI.D	**rd, rj, ui6**	0000000001001001																ui6						rj					rd				
ROTRI.W	**rd, rj, ui5**	00000000010011001																	ui5					rj					rd				
ROTRI.D	**rd, rj, ui6**	0000000001001101																ui6						rj					rd				
BSTRINS.W	**rd, rj, msbw, lsbw**	00000000011											msbw					0	lsbw					rj					rd				
BSTRPICK.W rd, rj, msbw, lsbw		00000000011											msbw					1	lsbw					rj					rd				
BSTRINS.D	**rd, rj, msbd, lsbd**	0000000010										msbd						lsbd						rj					rd				
BSTRPICK.D rd, rj, msbd, lsbd		0000000011										msbd						lsbd						rj					rd				
FADD.S	**fd, fj, fk**	00000001000000001																	fk					fj					fd				
FADD.D	**fd, fj, fk**	00000001000000010																	fk					fj					fd				
FSUB.S	**fd, fj, fk**	00000001000000101																	fk					fj					fd				
FSUB.D	**fd, fj, fk**	00000001000000110																	fk					fj					fd				
FMUL.S	**fd, fj, fk**	00000001000001001																	fk					fj					fd				
FMUL.D	**fd, fj, fk**	00000001000001010																	fk					fj					fd				
FDIV.S	**fd, fj, fk**	00000001000001101																	fk					fj					fd				
FDIV.D	**fd, fj, fk**	00000001000001110																	fk					fj					fd				
FMAX.S	**fd, fj, fk**	00000001000010001																	fk					fj					fd				
FMAX.D	**fd, fj, fk**	00000001000010010																	fk					fj					fd				
FMIN.S	**fd, fj, fk**	00000001000010101																	fk					fj					fd				
FMIN.D	**fd, fj, fk**	00000001000010110																	fk					fj					fd				
FMAXA.S	**fd, fj, fk**	00000001000011001																	fk					fj					fd				
FMAXA.D	**fd, fj, fk**	00000001000011010																	fk					fj					fd				
FMINA.S	**fd, fj, fk**	00000001000011101																	fk					fj					fd				
FMINA.D	**fd, fj, fk**	00000001000011110																	fk					fj					fd				
FSCALEB.S fd, fj, fk		00000001000100001																	fk					fj					fd				
FSCALEB.D fd, fj, fk		00000001000100010																	fk					fj					fd				
FCOPYSIGN.S fd, fj, fk		00000001000100101																	fk					fj					fd				
FCOPYSIGN.D fd, fj, fk		00000001000100110																	fk					fj					fd				
FABS.S	**fd, fj**	0000000100010100000001																						fj					fd				
FABS.D	**fd, fj**	0000000100010100000010																						fj					fd				
FNEG.S	**fd, fj**	0000000100010100000101																						fj					fd				
FNEG.D	**fd, fj**	0000000100010100000110																						fj					fd				
FLOGB.S	**fd, fj**	0000000100010100001001																						fj					fd				
FLOGB.D	**fd, fj**	0000000100010100001010																						fj					fd				
FCLASS.S	**fd, fj**	0000000100010100001101																						fj					fd				

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCLASS.D fd, fj	0000000100010100001110																						fj					fd				
FSQRT.S fd, fj	0000000100010100010001																						fj					fd				
FSQRT.D fd, fj	0000000100010100010010																						fj					fd				
FRECIP.S fd, fj	0000000100010100010101																						fj					fd				
FRECIP.D fd, fj	0000000100010100010110																						fj					fd				
FRSQRT.S fd, fj	0000000100010100011001																						fj					fd				
FRSQRT.D fd, fj	0000000100010100011010																						fj					fd				
FRECIPE.S fd, fj	0000000100010100011101																						fj					fd				
FRECIPE.D fd, fj	0000000100010100011110																						fj					fd				
FRSQRTE.S fd, fj	0000000100010100100001																						fj					fd				
FRSQRTE.D fd, fj	0000000100010100100010																						fj					fd				
FMOV.S fd, fj	0000000100010100100101																						fj					fd				
FMOV.D fd, fj	0000000100010100100110																						fj					fd				
MOVGR2FR.W fd, rj	0000000100010100101001																						rj					fd				
MOVGR2FR.D fd, rj	0000000100010100101010																						rj					fd				
MOVGR2FRH.W fd, rj	0000000100010100101011																						rj					fd				
MOVFR2GR.S rd, fj	0000000100010100101101																						fj					rd				
MOVFR2GR.D rd, fj	0000000100010100101110																						fj					rd				
MOVFRH2GR.S rd, fj	0000000100010100101111																						fj					rd				
MOVGR2FCSR fcsr, rj	0000000100010100110000																						rj					fcsr				
MOVFCSR2GR rd, fcsr	0000000100010100110010																						fcsr					rd				
MOVFR2CF cd, fj	0000000100010100110100																						fj					00		cd		
MOVCF2FR fd, cj	000000010001010011010100																								cj			fd				
MOVGR2CF cd, rj	0000000100010100110110																						rj					00		cd		
MOVCF2GR rd, cj	000000010001010011011100																								cj			rd				
FCVT.S.D fd, fj	0000000100011001000110																						fj					fd				
FCVT.D.S fd, fj	0000000100011001001001																						fj					fd				
FTINTRM.W.S fd, fj	0000000100011010000001																						fj					fd				
FTINTRM.W.D fd, fj	0000000100011010000010																						fj					fd				
FTINTRM.L.S fd, fj	0000000100011010001001																						fj					fd				
FTINTRM.L.D fd, fj	0000000100011010001010																						fj					fd				
FTINTRP.W.S fd, fj	0000000100011010010001																						fj					fd				
FTINTRP.W.D fd, fj	0000000100011010010010																						fj					fd				
FTINTRP.L.S fd, fj	0000000100011010011001																						fj					fd				
FTINTRP.L.D fd, fj	0000000100011010011010																						fj					fd				
FTINTRZ.W.S fd, fj	0000000100011010100001																						fj					fd				
FTINTRZ.W.D fd, fj	0000000100011010100010																						fj					fd				
FTINTRZ.L.S fd, fj	0000000100011010101001																						fj					fd				
FTINTRZ.LD fd, fj	0000000100011010101010																						fj					fd				
FTINTRNE.W.S fd, fj	0000000100011010110001																						fj					fd				
FTINTRNE.W.D fd, fj	0000000100011010110010																						fj					fd				

		31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10		09 08 07 06 05	04 03 02 01 00
FTINTRNE.L.S	fd, fj	0000000100011010111001		fj	fd
FTINTRNE.L.D	fd, fj	0000000100011010111010		fj	fd
FTINT.W.S	fd, fj	0000000100011011000001		fj	fd
FTINT.WD	fd, fj	0000000100011011000010		fj	fd
FTINT.LS	fd, fj	0000000100011011001001		fj	fd
FTINT.LD	fd, fj	0000000100011011001010		fj	fd
FFINT.S.W	fd, fj	0000000100011101000100		fj	fd
FFINT.S.L	fd, fj	0000000100011101000110		fj	fd
FFINT.D.W	fd, fj	0000000100011101001000		fj	fd
FFINT.D.L	fd, fj	0000000100011101001010		fj	fd
FRINT.S	fd, fj	0000000100011110010001		fj	fd
FRINT.D	fd, fj	0000000100011110010010		fj	fd
SLTI	rd, rj, si12	0000001000	si12	rj	rd
SLTUI	rd, rj, si12	0000001001	si12	rj	rd
ADDI.W	rd, rj, si12	0000001010	si12	rj	rd
ADDI.D	rd, rj, si12	0000001011	si12	rj	rd
LU52I.D	rd, rj, si12	0000001100	si12	rj	rd
ANDI	rd, rj, ui12	0000001101	ui12	rj	rd
OR	rd, rj, ui12	0000001110	ui12	rj	rd
CHORUS	rd, rj, ui12	0000001111	ui12	rj	rd
CSRRD	rd, csr	00000100	csr	00000	rd
CSRWR	rd, csr	00000100	csr	00001	rd
CSRXCHG	rd, rj, csr	00000100	csr	rj!=0,1	rd
CAP	code, rj, si12	0000011000	si12	rj	code
LDDIR	rd, rj, level	00000110010000	level	rj	rd
LDPTE	rj, seq	00000110010001	seq	rj	00000
IOCSRRD.B	rd, rj	0000011001001000000000		rj	rd
IOCSRRD.H	rd, rj	0000011001001000000001		rj	rd
IOCSRRD.W	rd, rj	0000011001001000000010		rj	rd
IOCSRRD.D	rd, rj	0000011001001000000011		rj	rd
IOCSRWR.B	rd, rj	0000011001001000000100		rj	rd
IOCSRWR.H	rd, rj	0000011001001000000101		rj	rd
IOCSRWR.W	rd, rj	0000011001001000000110		rj	rd
IOCSRWR.D	rd, rj	0000011001001000000111		rj	rd
TLBCLR		00000110010010000010000000000000			
TLBFLUSH		00000110010010000010010000000000			
TLBSRCH		00000110010010000010100000000000			
TLBRD		00000110010010000010110000000000			
TLBWR		00000110010010000011000000000000			
TLBFILL		00000110010010000011010000000000			
ERTN		00000110010010000011100000000000			

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
IDLE	level	00000110010010001																	level														
INVTLB	up, rj, rk	00000110010010011																	rk					rj					on				
FMADD.S	fd, fj, fk, fa	000010000001												but					fk					fj					fd				
FMADD.D	fd, fj, fk, fa	000010000010												but					fk					fj					fd				
FMSUB.S	fd, fj, fk, fa	000010000101												but					fk					fj					fd				
FMSUB.D	fd, fj, fk, fa	000010000110												but					fk					fj					fd				
FNMADD.S	fd, fj, fk, fa	000010001001												but					fk					fj					fd				
FNMADD.D	fd, fj, fk, fa	000010001010												but					fk					fj					fd				
FNMSUB.S	fd, fj, fk, fa	000010001101												but					fk					fj					fd				
FNMSUB.D	fd, fj, fk, fa	000010001110												but					fk					fj					fd				
FCMP.cond.S	cd, fj, fk	000011000001												cond					fk					fj					0 0		cd		
FCMP.cond.D	cd, fj, fk	000011000010												cond					fk					fj					0 0		cd		
FSEL	fd, fj, fk, ca	00001101000000														that			fk					fj					fd				
ADDU16I.D	rd, rj, si16	000100						si16																rj					rd				
LU12I.W	rd, si20	0001010							si20																				rd				
LU32I.D	rd, si20	0001011							si20																				rd				
PCADDI	rd, si20	0001100							si20																				rd				
PCALAU12I	rd, si20	0001101							si20																				rd				
PCADDU12I	rd, si20	0001110							si20																				rd				
PCADDU18I	rd, si20	0001111							si20																				rd				
LL.W	rd, rj, si14	00100000								yes14														rj					rd				
SC.W	rd, rj, si14	00100001								yes14														rj					rd				
LL.D	rd, rj, si14	00100010								yes14														rj					rd				
SC.D	rd, rj, si14	00100011								yes14														rj					rd				
LDPTR.W	rd, rj, si14	00100100								yes14														rj					rd				
STPTR.W	rd, rj, si14	00100101								yes14														rj					rd				
LDPTR.D	rd, rj, si14	00100110								yes14														rj					rd				
STPTR.D	rd, rj, si14	00100111								yes14														rj					rd				
LD.B	rd, rj, si12	0010100000										si12												rj					rd				
LD.H	rd, rj, si12	0010100001										si12												rj					rd				
LD.W	rd, rj, si12	0010100010										si12												rj					rd				
LD.D	rd, rj, si12	0010100011										si12												rj					rd				
ST.B	rd, rj, si12	0010100100										si12												rj					rd				
ST.H	rd, rj, si12	0010100101										si12												rj					rd				
ST.W	rd, rj, si12	0010100110										si12												rj					rd				
ST.D	rd, rj, si12	0010100111										si12												rj					rd				
LD.BU	rd, rj, si12	0010101000										si12												rj					rd				
LD.HU	rd, rj, si12	0010101001										si12												rj					rd				
LD.WU	rd, rj, si12	0010101010										si12												rj					rd				
PRELD	hint, rj, si12	0010101011										si12												rj					hint				
FLD.S	fd, rj, si12	0010101100										si12												rj					fd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FST.S	fd, rj, si12	0010101101										si12												rj					fd				
FLD.D	fd, rj, si12	0010101110										si12												rj					fd				
FST.D	fd, rj, si12	0010101111										si12												rj					fd				
LDX.B	rd, rj, rk	00111000000000000																	rk					rj					rd				
LDX.H	rd, rj, rk	00111000000001000																	rk					rj					rd				
LDX.W	rd, rj, rk	00111000000010000																	rk					rj					rd				
LDX.D	rd, rj, rk	00111000000011000																	rk					rj					rd				
STX.B	rd, rj, rk	00111000000100000																	rk					rj					rd				
STX.H	rd, rj, rk	00111000000101000																	rk					rj					rd				
STX.W	rd, rj, rk	00111000000110000																	rk					rj					rd				
STX.D	rd, rj, rk	00111000000111000																	rk					rj					rd				
LDX.BU	rd, rj, rk	00111000001000000																	rk					rj					rd				
LDX.HU	rd, rj, rk	00111000001001000																	rk					rj					rd				
LDX.WU	rd, rj, rk	00111000001010000																	rk					rj					rd				
PRELDX	hint, rj, rk	00111000001011000																	rk					rj					hint				
FLDX.S	fd, rj, rk	00111000001100000																	rk					rj					fd				
FLDX.D	fd, rj, rk	00111000001101000																	rk					rj					fd				
FSTX.S	fd, rj, rk	00111000001110000																	rk					rj					fd				
FSTX.D	fd, rj, rk	00111000001111000																	rk					rj					fd				
SC.Q	rd,rk,rj	00111000010101110																	rk					rj					rd				
LLACQ.W	rd, rj	0011100001010111100000																						rj					rd				
SCREL.W	rd, rj	0011100001010111100001																						rj					rd				
LLACQ.D	rd, rj	0011100001010111100010																						rj					rd				
SCREL.D	rd, rj	0011100001010111100011																						rj					rd				
AMCAS.B	rd, rk, rj	00111000010110000																	rk					rj					rd				
AMCAS.H	rd, rk, rj	00111000010110001																	rk					rj					rd				
AMCAS.W	rd, rk, rj	00111000010110010																	rk					rj					rd				
AMCAS.D	rd, rk, rj	00111000010110011																	rk					rj					rd				
AMCAS_DB.B	rd, rk, rj	00111000010110100																	rk					rj					rd				
AMCAS_DB.H	rd, rk, rj	00111000010110101																	rk					rj					rd				
AMCAS_DB.W	rd, rk, rj	00111000010110110																	rk					rj					rd				
AMCAS_DB.D	rd, rk, rj	00111000010110111																	rk					rj					rd				
AMSWAP.B	rd, rk, rj	00111000010111000																	rk					rj					rd				
AMSWAP.H	rd, rk, rj	00111000010111001																	rk					rj					rd				
AMADD.B	rd, rk, rj	00111000010111010																	rk					rj					rd				
AMADD.H	rd, rk, rj	00111000010111011																	rk					rj					rd				
AMSWAP_DB.B	rd, rk, rj	00111000010111100																	rk					rj					rd				
AMSWAP_DB.H	rd, rk, rj	00111000010111101																	rk					rj					rd				
AMADD_DB.B	rd, rk, rj	00111000010111110																	rk					rj					rd				
AMADD_DB.H	rd, rk, rj	00111000010111111																	rk					rj					rd				
AMSWAP.W	rd, rk, rj	00111000011000000																	rk					rj					rd				

	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15	14 13 12 11 10	09 08 07 06 05	04 03 02 01 00
AMSWAP.D rd, rk, rj	00111000011000001	rk	rj	rd
AMADD.W rd, rk, rj	00111000011000010	rk	rj	rd
APPENDIX.D rd, rk, rj	00111000011000011	rk	rj	rd
AMAND.W rd, rk, rj	00111000011000100	rk	rj	rd
AMAND.D rd, rk, rj	00111000011000101	rk	rj	rd
AMOR.W rd, rk, rj	00111000011000110	rk	rj	rd
LOVE.D rd, rk, rj	00111000011000111	rk	rj	rd
AMXOR.W rd, rk, rj	00111000011001000	rk	rj	rd
AMXOR.D rd, rk, rj	00111000011001001	rk	rj	rd
AMMAX.W rd, rk, rj	00111000011001010	rk	rj	rd
AMMAX.D rd, rk, rj	00111000011001011	rk	rj	rd
ADMIN.W rd, rk, rj	00111000011001100	rk	rj	rd
AMMIN.D rd, rk, rj	00111000011001101	rk	rj	rd
AMMAX.WU rd, rk, rj	00111000011001110	rk	rj	rd
AMMAX.DU rd, rk, rj	00111000011001111	rk	rj	rd
AMEN.ME rd, rk, rj	00111000011010000	rk	rj	rd
ADMIN.YOU rd, rk, rj	00111000011010001	rk	rj	rd
AMSWAP_DB.W rd, rk, rj	00111000011010010	rk	rj	rd
AMSWAP_DB.D rd, rk, rj	00111000011010011	rk	rj	rd
AMADD_DB.W rd, rk, rj	00111000011010100	rk	rj	rd
AMADD_DB.D rd, rk, rj	00111000011010101	rk	rj	rd
AMAND_DB.W rd, rk, rj	00111000011010110	rk	rj	rd
AMAND_DB.D rd, rk, rj	00111000011010111	rk	rj	rd
AMOR_DB.W rd, rk, rj	00111000011011000	rk	rj	rd
AMOR_DB.D rd, rk, rj	00111000011011001	rk	rj	rd
AMXOR_DB.W rd, rk, rj	00111000011011010	rk	rj	rd
AMXOR_DB.D rd, rk, rj	00111000011011011	rk	rj	rd
AMMAX_DB.W rd, rk, rj	00111000011011100	rk	rj	rd
AMMAX_DB.D rd, rk, rj	00111000011011101	rk	rj	rd
AMMIN_DB.W rd, rk, rj	00111000011011110	rk	rj	rd
ADMIN_DB.D rd, rk, rj	00111000011011111	rk	rj	rd
AMMAX_DB.WU rd, rk, rj	00111000011100000	rk	rj	rd
AMMAX_DB.DU rd, rk, rj	00111000011100001	rk	rj	rd
ADMIN_DB.WU rd, rk, rj	00111000011100010	rk	rj	rd
AMMIN_DB.DU rd, rk, rj	00111000011100011	rk	rj	rd
DBAR hint	00111000011100100	hint		
VALLEY hint	00111000011100101	hint		
FLDGT.S fd, rj, rk	00111000011101000	rk	rj	fd
FLDGT.D fd, rj, rk	00111000011101001	rk	rj	fd
FLDLE.S fd, rj, rk	00111000011101010	rk	rj	fd
FLDLE.D fd, rj, rk	00111000011101011	rk	rj	fd

		31 30 29 28 27 26	25 24 23 22 21 20 19 18 17 16 15	14 13 12 11 10	09 08	07 06 05	04 03 02 01 00
FSTGT.S	**fd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 0 1 1 0 0		rk	rj		fd
FSTGT.D	**fd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 0 1 1 0 1		rk	rj		fd
FSTLE.S	**fd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 0 1 1 1 0		rk	rj		fd
FSTLE.D	**fd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 0 1 1 1 1		rk	rj		fd
LDGT.B	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0		rk	rj		rd
LDGT.H	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 0 0 0 1		rk	rj		rd
LDGT.W	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 0 0 1 0		rk	rj		rd
LDGT.D	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 0 0 1 1		rk	rj		rd
LDLE.B	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 0 1 0 0		rk	rj		rd
LDLE.H	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 0 1 0 1		rk	rj		rd
LDLE.W	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 0 1 1 0		rk	rj		rd
LDLE.D	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 0 1 1 1		rk	rj		rd
STGT.B	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0		rk	rj		rd
STGT.H	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 1 0 0 1		rk	rj		rd
STGT.W	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0		rk	rj		rd
STGT.D	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 1 0 1 1		rk	rj		rd
STLE.B	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 1 1 0 0		rk	rj		rd
STLE.H	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1		rk	rj		rd
STLE.W	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0		rk	rj		rd
STLE.D	**rd, rj, rk**	0 0 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1		rk	rj		rd
BEQZ	**rj, offs**	0 1 0 0 0 0	offs[15:0]		rj		offs[20:16]
BNEZ	**rj, offs**	0 1 0 0 0 1	offs[15:0]		rj		offs[20:16]
BCEQZ	**cj, offs**	0 1 0 0 1 0	offs[15:0]		0 0	cj	offs[20:16]
BCNEZ	**cj, offs**	0 1 0 0 1 0	offs[15:0]		0 1	cj	offs[20:16]
JIRL	**rd, rj, offs**	0 1 0 0 1 1	offs[15:0]		rj		rd
B	**offs**	0 1 0 1 0 0	offs[15:0]		offs[25:16]		
BL	**offs**	0 1 0 1 0 1	offs[15:0]		offs[25:16]		
BEQ	**rj, rd, offs**	0 1 0 1 1 0	offs[15:0]		rj		rd
BNE	**rj, rd, offs**	0 1 0 1 1 1	offs[15:0]		rj		rd
BLT	**rj, rd, offs**	0 1 1 0 0 0	offs[15:0]		rj		rd
BGE	**rj, rd, offs**	0 1 1 0 0 1	offs[15:0]		rj		rd
BLTU	**rj, rd, offs**	0 1 1 0 1 0	offs[15:0]		rj		rd
BGEU	**rj, rd, offs**	0 1 1 0 1 1	offs[15:0]		rj		rd