LOONGSON

Dragon Architecture 32-bit Compact Reference Manual

V1.04

2025 3 years

Loongson Technology Co., Ltd.

北京市海淀区温泉镇中关村环保科技示范园龙芯产业园2号楼 100095
Loongson Industrial Park, building 2，Zhongguancun environmental protection park
Haidian District，Beijing

www.loongson.cn

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 for the introduction of the Dragon architecture 32-bit simplified version.

龙芯中科技术股份有限公司
Loongson Technology Corporation Limited

Version History

Document Update History: Document Name, Version Number		Dragon Architecture 32-bit Compact Reference Manual
		1.04
	Creator	Chip R&D Department
	Creation Date	2020/09/17

Update History

Version number	Update content
0.90	Internal review version.
0.91	Internal review version.
1.00	Released to the public.
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 have been implemented. 2) Fixed 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) Sections 2.2.3, 2.2.5.1, and 3.2.5 discuss the relationship between immediate values used in the instruction assembly representation and immediate values in the instruction code. This is an explanation.
1.02	Instruction content adjustment: 1) Remove the following six floating-point instructions from the floating-point instruction definition: FSCALEB.S/D, FLOGB.S/D, and FRINT.S/D. This is only required in the LA64 architecture. The instructions implemented below. Improvements and revisions to the manual: 1) 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. 2) Section 2.2.5.1 provides details on the specific operation method for calculating the offset value when the LL/SC instruction calculates the address. 3) All “NUL bit of CSR.TLBIDX” in Section 4.2.3 should be changed to “NE bit of CSR.TLBIDX”. 4) When the TLBRD instruction reads NE=0, the target CSR register is explicitly updated to 0. 5) In the description of page size (PS) in section 5.4.2, the PS value of the TLB entry corresponding to a page size of 4MB should be 21. 6) In the last sentence of section 5.4.2, [log2PS-1:12] should be [PS-1:12]. 7) In section 5.4.4, the exception for non-compliance of privilege level of the reporting page should be SignalException(PPI). 8) Adjust the definition of the original LIE[10] bit in section 7.4.4; adjust the definition of the IS[10] bit in section 7.4.5, and clarify IS[9:2] and IS[11]. The definition of IS[12] bits. 9) Section 7.4.5 explicitly sets all undefined Esubcode values to the default value of 0. 10) Section 7.5.3 describes the PPN fields of the CSR registers TLBELO0 and TLBELO1, taking into account the case where PALEN < 36. 11) 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.
1.03	Improvements and revisions to the manual: 1) The description of the address error exception in section 2.1.4 has been adjusted because the memory access restrictions imposed on application software at the instruction set level are no longer applied.

	When the address space is within the range, the only exception triggered by the instruction fetch address error is when the instruction fetch PC is misaligned. 2) Section 2.1.5 has been revised to no longer restrict the range of memory address space that application software can access at the instruction set level. The restrictions are implemented by the system software. 3) Section 2.2.1.10 MULH.WU is missing a U. 4) In section 3.1.4.4, 2Emin should appear in the exponent position. 5) 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. 6) Section 3.2.6.1 simplifies floating-point memory access instructions to address the requirement for memory address alignment in the 32-bit simplified version of the Dragon architecture. Describe the non-alignment exception condition. 7) The instruction function descriptions in sections 4.2.3.3 and 4.2.3.4 , 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 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. 9) In section 5.4.3.4, "INVTLB r0, r0" should be "INVTLB 0, r0, r0". 10) The range truncated in the second-to-last line of the pseudocode description in section 5.4.4 is incorrect. 11) The memory access instructions during the execution phase in section 6.2.2 will no longer trigger address error exceptions because the application is not restricted at the instruction set level. The range of memory address space that the software can access. 12) 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… 13) The RDTIME instruction described in section 7.6.1 should be changed to the RDCNTID instruction. 14) The statement involving GRLEN is determined to be 32.
1.04	Improvements and revisions to the manual: 1) Delete the content related to sign extension after concatenation of immediate values in Section 2.2.1.6 of the Natural Language and Pseudocode description to conform to this manual. The book only covers the context of 32-bit machines. 2) Add a list of "basic floating-point instructions for operating single-precision floating-point numbers and word integers" to Chapter 3. 3) Add FFINT.{S/D}.L and FTINT.L.{S/D} to section 3.2.3.2, and add them to section 3.2.3.3. Description of the FTINT{RM/RP/RZ/RNE}.L.{S/D} instruction.

Table of contents

Catalog

Table of Contents

1 Introduction

1.1 Overview of Dragon Architecture

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

The system architecture is characterized by fixed instruction lengths and standardized encoding formats. Most instructions have only two source operands and one destination operand.

The architecture uses a load/store approach, meaning only load/store memory access instructions can access memory; other instructions operate on registers within the processor core.

An immediate value in a device or instruction code.

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).

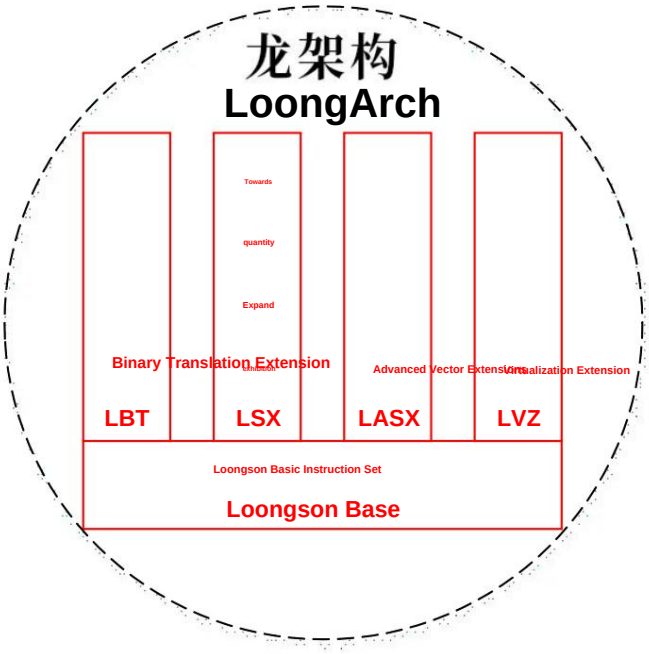


Figure 1-1 Components of the Dragon Architecture

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.

The LA32-bit version of the Dragon architecture includes both integer and floating-point instructions, fully supporting the generation of efficient target code by existing mainstream compilation systems.

The basic components have been further simplified to make them easier to implement and convenient for widespread use in teaching and research.

This manual will begin with a detailed description of the specifications for the 32-bit simplified version of the Dragon architecture in Chapter 2. Chapters 2 and 3 specifically cover the architecture...

The non-privileged instruction set portion of the architecture includes the functional definitions of basic integer instructions and basic floating-point instructions, as well as their application-level programming model. Chapter 4

Chapters 7 through 7 cover privileged resources in the infrastructure, primarily including privileged instructions and control and status registers.

This document introduces the Register (CSR) and its functional specifications regarding operating modes, exceptions and interrupts, and memory management. (The main text of this document...)

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

1.2 Instruction Encoding Format

All instructions in the Dragon architecture 32-bit simplified version are 32-bit fixed length, and instruction addresses require 4-byte boundary alignment.

An address error exception will be triggered if the address is misaligned.

The instruction encoding style is such that all register operand fields are arranged sequentially from bit 0 to bit 1. The opcode starts at bit 31.

They are arranged sequentially from high to low. 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.

Equations 2R, 3R, and 4R, and six encoding formats containing immediate values: 2RI8, 2RI12, 2RI14, 2RI16, 1RI21, and 126. Table 1-1 lists these.

The specific definitions of these nine typical encoding formats are as follows. It should be noted that there are a few instructions whose instruction encoding fields are not entirely equivalent to these nine typical formats.

It's not a standard instruction encoding format, but rather a slightly modified version. However, the number of such instructions is small, and the changes are minor, so they won't significantly impact...

This causes inconvenience for developers of the compilation system.

Table 1-1 Typical Instruction Encoding Format of the Dragon Architecture **32** -bit Simplified Version

	31–26	25–22	21–20	19–15	14–10	9–5	4–0
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 includes two parts: the instruction name and the operands. The Dragon Architecture 32-bit Simplified Edition specifies the prefix and suffix of the instruction name and operands.

A unified approach was taken to facilitate use by assembly programmers and compiler developers.

First, integer and floating-point instructions are distinguished by the prefix letter of their instruction names. All non-vector floating-point instructions begin with the letter "F".

beginning.

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, .BU, .HU, or .WU.

Do not specify whether the data type operated on by this instruction is a signed byte, a signed half-word, a signed word, an unsigned byte, an unsigned half-word, or an unsigned byte.

However, there is a special case: when whether the operands are signed or unsigned does not affect the result, the suffix in the instruction name...

The prefix 'U' is omitted, but this does not restrict the operands to only signed numbers. This applies to operands that are floating-point types, or more specifically...

These are instructions whose names begin with "F" and whose suffixes are .H, .S, .D, .W, and .WU, respectively indicating the data type the instruction operates on.

These are half-precision floating-point numbers, single-precision floating-point numbers, double-precision floating-point numbers, signed words, and unsigned words. It should be noted that not all instructions...

All instructions use a suffix in the form of ".XX" to indicate the operand of the instruction. This applies when the data width of the operand is implemented by a 32-bit processor.

The 64-bit architecture still determines this, as seen in instructions like SLT and SLTU, which do not include suffixes. Furthermore, privileges for operating CSR, TLB, and Cache also apply.

State instructions and instructions that move data between different register files also do not have this suffix indicating the type of 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".

Used to label floating-point registers. Here, N is a number indicating 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

In the instruction set defined in the 32-bit simplified version of the Dragon architecture, there are often some instructions that have the same or similar operation modes, differing only in the objects they operate on.

There are some differences. In the introduction of commands and functions in this manual, such commands are often grouped together for easy learning by the user.

For brevity, this manual uses a rule for abbreviating command names. In this rule, {A/B/C} indicates that A, B, C, and C are used respectively here.

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

This represents the instruction names ADD.W and ADD.D, while BLT[U] represents the instruction names BLT and BLTU. A more complex version...

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 32-bit simplified version defines a series of control and status registers (CSRs) for control.

The execution behavior of control instructions is described, and each Control Request (CSR) typically contains several fields. This manual will use the form CSR.%%%%.#### throughout the description.

This refers to the field named #### in the control status register whose name is abbreviated as %%%% . For example, CSR.CRMD.PLV represents the CRMD field.

The PLV field in the register.

2 Basic Integer Instructions

The non-privileged instruction set of the Dragon architecture 32-bit simplified version can be divided into basic integer instructions and basic floating-point instructions based on the differences in the software runtime context.

Point instructions are divided into two parts. This chapter will describe the integer instruction part. The basic integer instruction part is the most fundamental part of the non-privileged instruction subset.

part.

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), and word (W, 32 bits). The LA32 architecture does not support double-word

operations.

Integer instructions.

Byte, half-word, and 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.

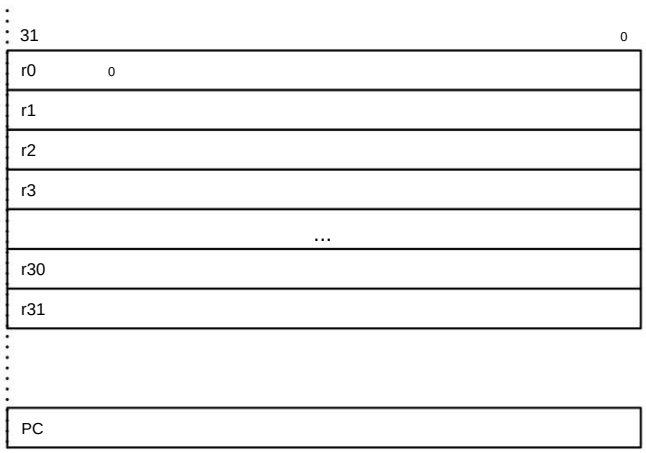


Figure 2-1 General-purpose registers and **PC**

[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 32 bits. (Basic)

Integer instructions and general-purpose registers are orthogonal. That is, from an architectural perspective, any register operand in these instructions can use a 32-bit register.

Any of the GR instructions. The only exception is that the destination register implicitly included in the BL instruction is always register r1, number 1. In the standard Dragon architecture...

In the Application Binary Interface (ABI), r1 is always used as a register to store the return address of a function call.

Utensils.

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 32-bit simplified version defines two privilege levels (PLVs): PLV0 and PLV3. Application software should run at the non-privileged PLV3 level to isolate it from system

software such as the operating system running at PLV0.

For more information on privilege levels, please see Section 4.1 .

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 Error Exception: Executing a privileged instruction in an application will immediately trigger an instruction privilege level error exception (IPE).

ÿ Address Error Exception: When a program malfunctions, causing an illegal instruction fetch, i.e., the fetch address is not aligned to a 4-byte boundary,

This will trigger an Address Fetch Error Exception (ADEF).

[1] In the 32-bit simplified version of the Loongson architecture, interrupts are always invisible to application software.

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 32-bit simplified version, the memory address space is a byte-addressable linear contiguous address space. The recommended memory address space for application software...

The access range is: 0~ 231-1 .

2.1.6 Tail end

The Dragon architecture 32-bit simplified version only uses little-endian storage.

2.1.7 Storage Access Types

The Dragon architecture 32-bit simplified version supports two storage access types: Coherent Cached (CC) and Strong Cached.

Strongly-ordered Uncached (SUC). 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

corresponds to the memory access type as follows: 0 — Strongly-ordered Uncached, 1 —

—Consistently cacheable, 2/3 reserved. The process of setting storage access types is transparent to 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.

Next memory access operation.

The Dragon architecture 32-bit simplified version only requires that strongly ordered, non-cached memory access instructions cannot have side effects; that is, such instructions cannot...

Speculative execution. Software can leverage this characteristic to access I/O devices in the system via strongly ordered, unbuffered memory access instructions. However, the gantry...

The 32-bit simplified version allows fetch operations on strongly ordered, uncached types to have side effects. This means that fetch operations on strongly ordered, uncached types...

Even if the action originates from a transfer prediction, it is permitted to execute. To prevent such speculative executions from causing unauthorized out-of-core memory accesses, execution is allowed.

Managing the address space requires filtering out risky accesses within the on-chip network.

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.

---

[1] This only applies to application software. For system software, the address falls within the address range configured in the direct address translation mode or the mapped address translation mode.

Within this range, the storage access type is configured by the specified control status register.

The cache coherency between the processor core's instruction cache and data cache is maintained by software. This means that for self-modifying code,

The software needs to use cache maintenance instructions to ensure cache consistency between the instruction cache and the data cache within the same core. Furthermore, because...

Due to the existence of pipelined architecture and speculative instruction fetching, software still needs to use the IBAR instruction to ensure that instruction fetching will always see the execution of store instructions.

Effect. When using software to maintain cache coherence between the same core instruction cache and data cache, CACOP with codes equal to 8 and 9...

The instructions (i.e., Hit Invalidate I-Cache and Hit Invalidate and Writeback D-Cache) are downgraded from privileged instructions to user-mode instructions.

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.

All memory access instructions must undergo address alignment checks. For memory access instructions that require address alignment checks, if the address they access is not...

If it is naturally aligned to 1, it will trigger an address unaligned exception (ALE).

2.1.9 Brief Description of Storage Consistency Model

The Dragon architecture 32-bit simplified version uses a weak consistency (WC) model for storage consistency. This section only...

A brief description of the weak consistency model adopted by the architecture.

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 32-bit simplified version, the instructions that can generate synchronous operations are the DBAR, IBAR, and LL-SC instruction pairs.

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.

2.2 Overview of Basic Integer Instructions

2.2.1 Arithmetic Operation Instructions

2.2.1.1ADD.W, SUB.W

Command format: add.w rd, rj, rk

sub.in rd, rj, rk

ADD.W adds the data in general-purpose register rj to the data in general-purpose register rk, and writes bits [31:0] of the result into general-purpose register rd. middle.

ADD.W:

```
tmp = GR[rj] + GR[rk]
GR[rd] = tmp[31:0]
```

SUB.W subtracts the data in general-purpose register rj from the data in general-purpose register rk, and writes bits [31:0] of the result into general-purpose register rd. middle.

SUB.W:

```
tmp = GR[rj] - GR[rk]
GR[rd] = tmp[31:0]
```

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

2.2.1.2ADDI.W

Command format: addi.w rd, rj, si12

ADDI.W adds the 32-bit sign-extended immediate value si12 to the data in the general-purpose register rj, and writes the result back to the register. Use register rd.

ADDI.W:

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

This instruction does not perform any special handling for overflow situations when it is executed.

2.2.1.3LU12I.W

Command format: lu12i.w rd, si20

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

LU12I.W:

GR[rd] = {si20, 12'b0}

This instruction, along with the ORI instruction, is used to load immediate values of more than 12 bits into a general-purpose register.

2.2.1.4SLT[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 will set the value of the general-purpose register rd to 1, otherwise set it to 0.

SLT:

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

SLTU treats the data in general-purpose register rj and the data in general-purpose register rk as unsigned integers and compares their magnitudes. If the former is smaller...

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

2.2.1.5SLT[U]I

Command 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 calculation.

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, 32)

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-extending the 12-bit immediate value si12 as an unsigned 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.

For SLT:

tmp = SignExtend(si12, 32)

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.

2.2.1.6PCADDU12I

Command format: pcaddu12i rd, si20

PCADDU12I concatenates the least significant bit of the 20-bit immediate value si20 with 12 bits of 0, adds the PC value of the instruction to the resulting data, and writes the sum to...

Enter it into the general-purpose register rd.

PCADDU12I:

GR[rd] = PC + {si20, 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.

2.2.1.7AND, OR, NOR, XOR

Command format: and rd, rj, rk

or rd, rj, rk

nor rd, rj, rk

free 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 back to the general-purpose register.

rd in.

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 to general-purpose register rd.

middle.

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 back to the general-purpose register.

rd in.

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 back to the general-purpose register.

rd in.

FREE:

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.

2.2.1.8ANDI, 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 register.

Use register rd.

ANDI:

GR[rd] = GR[rj] & ZeroExtend(ui12, 32)

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

In register rd.

WHEN:

GR[rd] = GR[rj] | ZeroExtend(ui12, 32)

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 to...

In the general-purpose register rd.

CHORUS:

GR[rd] = GR[rj] ^ ZeroExtend(ui12, 32)

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.

2.2.1.9NOP

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.

2.2.1.10 MUL.W, MULH.W[U]

Command format: mul.w rd, rj, rk

mulh.w rd, rj, rk

mulh.wu rd, rj, rk

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

MUL.W:

product = signed(GR[rj]) * signed(GR[rk])

GR[rd] = product[31:0]

MULH.W multiplies the data in general-purpose register rj and the data in general-purpose register rk as signed numbers, and the product is [63:32]. Bit data is written into the general-purpose register rd.

MULH.W:

product = signed(GR[rj]) * signed(GR[rk])

GR[rd] = product[63:32]

MULH.WU treats the data in general-purpose register rj and the data in general-purpose register rk as unsigned numbers and multiplies the product. [63:32] The bit data is written to the general-purpose register rd after sign extension.

MULH.WU:

product = unsigned(GR[rj]) * unsigned(GR[rk])

GR[rd] = product[63:32]

2.2.1.11 DIV.W[U], MOD.W[U]

Command format: div.w rd, rj, rk

mod.w rd, rj, rk

div.wu rd, rj, rk

mod.wu rd, rj, rk

DIV.W and DIV.WU divide the data in general-purpose register rj by the data in general-purpose register rk, and write the quotient into general-purpose register rd. middle.

DIV.W:

quotient = signed(GR[rj]) / signed(GR[rk])

GR[rd] = quotient[31:0]

DIV.WU:

quotient = unsigned(GR[rj]) / unsigned(GR[rk])

GR[rd] = quotient[31:0]

MOD.W and MOD.WU divide the data in general-purpose register rj by the data in general-purpose register rk, and write the remainder into the general-purpose register. In the device rd.

MOD.W:

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

MOD.WU:

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

When performing division operations with DIV.W and MOD.W, the operands are both treated as signed numbers. When performing division operations with DIV.WU and MOD.WU, All source operands are treated as unsigned numbers.

Each pair of quotient/remainder instructions for DIV.W/MOD.W and DIV.WU/MOD.WU results in a remainder whose sign is relative to the dividend. The remainders are consistent 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

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

SLL.W logically shifts the data in general-purpose register rj to the left, and writes the shift result into general-purpose register rd.

SLL.W:

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

SRL.W logically right-shifts the data in general-purpose register rj, and writes the shift result into general-purpose register rd.

SRL.W:

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

SRA.W performs an arithmetic right shift of the data in the general-purpose register rj, and writes the shift result into the general-purpose register rd.

SRA.W:

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

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

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

SLLI.W logically shifts the data in general-purpose register rj to the left, and writes the shift result into general-purpose register rd.

SLLI.W:

tmp = SLL(GR[rj], ui5)

GR[rd] = tmp[31:0]

SRLI.W logically right-shifts the data in general-purpose register rj and writes the shift result into general-purpose register rd.

SRLI.W:

tmp = SRL(GR[rj], ui5)

GR[rd] = tmp[31:0]

SRAI.W performs an arithmetic right shift of the data in general-purpose register rj, and writes the shift result into general-purpose register rd.

SRAI.W:

tmp = SRA(GR[rj], ui5)

GR[rd] = tmp[31:0]

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

2.2.3 Transfer Instructions

2.2.3.1BEQ, 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}, 32)
```

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}, 32)
```

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}, 32)
```

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}, 32)
```

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}, 32)
```

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}, 32)
```

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.

2.2.3.2B

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}, 32)
```

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.

2.2.3.3BL

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}, 32)
```

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.

2.2.3.4JIRL

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}, 32)
```

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.

2.2.4 Normal Memory Access Instructions

2.2.4.1LD.{B[U]/H[U]/W}, ST.{B/H/W}

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

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

Data is written to the general-purpose register rd.

LD.B:

```
vaddr = GR[rj] + SignExtend(si12, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
byte = MemoryLoad(paddr, BYTE)
GR[rd] = SignExtend(byte, 32)
```

LD.H:

```
vaddr = GR[rj] + SignExtend(si12, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
halfword = MemoryLoad(paddr, HALFWORD)
GR[rd] = SignExtend(halfword, 32)
```

LD.W:

```
vaddr = GR[rj] + SignExtend(si12, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
word = MemoryLoad(paddr, WORD)
GR[rd] = word
```

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

LD.BU:

```
vaddr = GR[rj] + SignExtend(si12, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
byte = MemoryLoad(paddr, BYTE)
GR[rd] = ZeroExtend(byte, 32)
```

LD.HU:

```
vaddr = GR[rj] + SignExtend(si12, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
halfword = MemoryLoad(paddr, HALFWORD)
GR[rd] = ZeroExtend(halfword, 32)
```

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

ST.B:

```
vaddr = GR[rj] + SignExtend(si12, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][7:0], paddr, BYTE)
```

ST.H:

```
vaddr = GR[rj] + SignExtend(si12, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][15:0], paddr, HALFWORD)
```

ST.W:

```
vaddr = GR[rj] + SignExtend(si12, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][31:0], paddr, WORD)
```

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} and ST.{B/H/W} instructions, as long as the memory access address is naturally aligned, the unaligned exception will not be triggered; otherwise...

This will trigger a non-alignment exception.

2.2.4.2PRELD

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 Atomic memory access instructions

2.2.5.1LL.W, SC.W

Command format: ll.w rd, rj, si14

sc.w rd, rj, si14

The instruction pair LL.W and SC.W is used to implement an atomic "read-modify-write" memory access sequence. The LL.W instruction fetches memory from a specified address. After sign-expanding a single word of data, it is written to the general-purpose register rd. The paired SC.W instruction operates on data of the same width and accesses the same data. Memory address. The mechanism for maintaining the atomicity of memory access operation sequences is that LL.W records the access address and sets a flag (LLbit is set to 1) during execution. The SC.W instruction checks the LLbit bit during execution. It only performs a write operation if the LLbit bit is 1; otherwise, it does not write. This is necessary when the software requires a certain level of success. When performing a "read-modify-write" memory access sequence for an atom, a loop needs to be constructed to repeatedly execute the LL-SC instruction pair until SC completes successfully. To construct this loop, the SC.{W/D} instruction will use a flag indicating whether its execution was successful (which can also be simply understood as what the SC instruction shows when it executes). 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 I/O master have completed a process in the cache line containing the address corresponding to that LLbit. Store operations.

If the storage access attribute of the LL-SC instruction at the access address is not cached, then the execution result is uncertain.

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}, 32)
```

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.

2.2.6 Barrier Commands

2.2.6.1DBAR

Command format: dbar hint

The DBAR instruction is used to establish a barrier between load/store memory access operations. Its immediate hint value indicates the synchronization of that barrier.

Objects and synchronization levels.

A hint value of 0 is required by default and indicates a fully functional synchronization barrier. This only applies after all previous load/store memory accesses have been performed.

Only after the "DBAR 0" instruction has been completely executed can it begin execution; and only after "DBAR 0" has finished executing can all subsequent load/store operations begin.

Only after a memory access operation can it begin to be executed.

If no specific function is implemented, all other hint values must be executed as hint=0.

2.2.6.2IBAR

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.

2.2.7 Other Miscellaneous Instructions

2.2.7.1SYSCALL

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.

2.2.7.2BREAK

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.

2.2.7.3RDCNTV{L/H}.W, RDCNTID

Command format: rdcntvl.w rd

rdcntvh.w rd

radcnts rj

The Dragon architecture 32-bit simplified version defines a constant frequency timer, the main body of which is a 64-bit counter called the Stable Counter. The Stable 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. Each timer has a software-configurable, globally unique number called the Counter ID.

The RDCNTV{L/H}.W instruction is used to read information from a constant frequency timer, where RDCNTVL.W reads bits [31:0] of the Counter and writes them to the Counter. In the general-purpose register rd, RDCNTVH.W reads bits [63:32] of the Counter. The RDCNTID Counter ID information is written to the general-purpose register rj. middle.

In the 32-bit simplified version of the Dragon architecture, the instructions RDCNTID rj, RDCNTH, and RDCNTL.W rd, RDCNTH.W rd, and RDCNTID rj actually correspond to the three RDTIMEL.W rd, zero, RDTIMEH.W rd, zero, and RDTIMEL.W zero, rj instructions in the 32-bit Dragon architecture, respectively. Special uses of the command.

3 Basic Floating-Point Instructions

This chapter introduces the floating-point instructions in the non-privileged subset of the Dragon Architecture 32-bit Compact Edition. The basic floating-point instructions in the Dragon Architecture 32-bit Compact Edition...

The function definition of the point instruction follows the 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 implement only the instructions in the basic floating-point instructions that operate on single-precision floating-point numbers and word integers (see Table 3-1).

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	BCEQZ, BCNEZ
instructions, floating-point normal memory access instructions	FLD.S, FST.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...

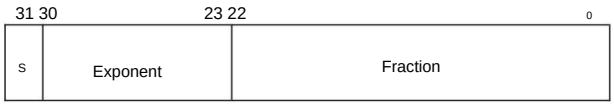
Programming with basic floating-point instructions builds upon the basic integer instruction programming model and further delves into the content discussed in this section.

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:



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

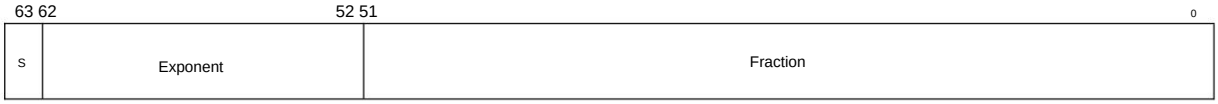
Exponent	Fraction	S	bit[22]	In
0	=0	0	0	+0
		1	0	-0
0	!=0	0 Any value		The denormalized number has a value of +2 -126 × (0.Fraction).

Exponent	Fraction	S	bit[22]	In
		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:



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...

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...

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

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 W, length 32 bits).

All 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.

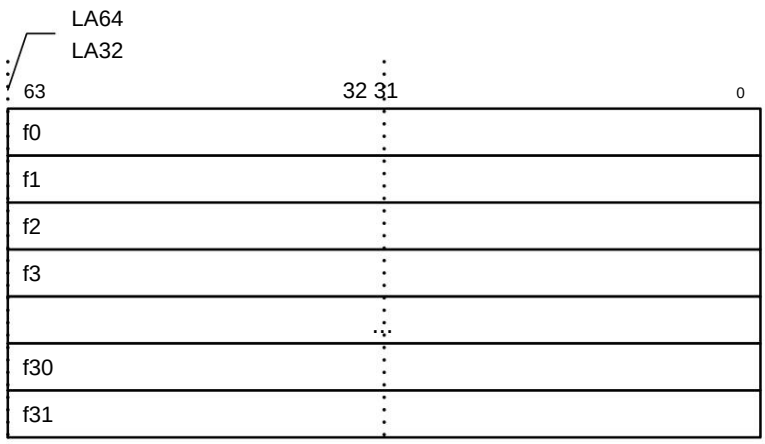


Figure 3-1 Floating-point register

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 is one CFR, denoted as fcc0, and each one can be read and written. The CFR has a bit width of 1 bit. The result of the floating-point comparison will be written to the condition.

In the 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 comes from 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 roundTiesToEven in IEEE 754-2008; 1: RZ, corresponding to roundTowardZero 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.
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 produce 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)
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.

[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.

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 a non-zero tiny value is detected, the final rounding result of the floating-point operation is not considered precise. 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][31:0])

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][31:0])

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]));

3.2.1.7FCOPYSIGN.{S/D}

Command format: fcopysign.s fd, fj, fk fcopysign.d fd, fj, fk

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			
		∞	normal	subnormal	0	∞	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])

3.2.2 Floating-point comparison instructions

3.2.2.1FCMP.cond.{S/D}

Instruction format: `fcmp.cond.s cc, fj, fk`. This is a floating-point fcmp.cond.d cc, fj, fk

comparison instruction that stores the comparison result in the specified status code (cc). There are 22 possible `cond` values for this instruction, which compare...

The conditions and criteria for judgment 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 Cannot be compared or are not equal to 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 Cannot be compared or are not equal to 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. Two operands cannot be compared if at least one of them is NaN.

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.wd	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 even number	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] = 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] = 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] = 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 to the floating-point register fd, using the method of "rounding to the nearest even number".

FTINTRNE.W.S:

FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], 0)

FTINTRNE.W.D:

FR[fd] = 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.

Command Name	Operations in the IEEE 754-2008 standard
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)

3.2.4 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, MOVGR2FRH.W

Command format: movgr2fr.w fd, rj

movgr2frh.w fd, rj

MOVGR2FR.W writes the value of the general-purpose register rj into 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.

MOVGR2FR.Wÿ

FR[fd][31:0] = GR[rj]

MOVGR2FRH.W writes the value of the general-purpose register rj into the high 32 bits of the floating-point register fd, while leaving the low 32 bits of the floating-point register fd unchanged.

MOVGR2FRH.Wÿ

FR[fd][63:32] = GR[rj]

FR[fd][31: 0] = FR[fd][31:0]

3.2.4.4MOVFR2GR.S, MOVFRH2GR.S

Command format: movfr2gr.s rd, fj

movfrh2gr.s rd, fj

MOVFR2GR/MOVFRH2GR.S writes the lower 32-bit/high 32-bit value of the floating-point register fj to the general-purpose register rd.

MOVFR2GR.Sÿ

GR[rd] = FR[fj][31:0]

MOVFRH2GR.Sÿ

GR[rd] = FR[fj][63:32]

3.2.4.5MOVGR2FCSR, MOVFCSR2GR

Command format: movgr2fcsr fcsr, rj

movfcsr2gr rd, fcsr

MOVGR2FCSR modifies the value of the software-writable field corresponding to the floating-point control status register indicated by fcsr based on the value of the general-purpose register rj. If

The MOVGR2FCSR instruction modifies FCSR0 so that both the Cause field bit and the corresponding Enables bit are 1, or modifies FCSR1.

The Enables field and the Cause field of FCSR2 are set so that the Cause bit and the corresponding Enables bit are both 1. The MOVGR2FCSR instruction itself will not...

Triggering a floating-point exception.

MOVGR2FCSRÿ

FCSR[fcsr] = GR[rj]

MOVFCSR2GR writes the 32-bit value of the floating-point control status register indicated by fcsr to the general-purpose register rd.

MOVFCSR2GRÿ

GR[rd] = FCSR[fcsr]

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][0] = 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 lowest bit of the general-purpose register rd, and pads the remaining bits of rd with 0.

MOVCF2GRÿ

GR[rd][0] = ZeroExtend(CFR[cj], 32)

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.

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.

BCEQZ:

if CFR[cj]==0 :

PC = PC + SignExtend({offs21, 2'b0}, 32)

BCNEZ:

if CFR[cj]!=0 :

PC = PC + SignExtend({offs21, 2'b0}, 32)

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, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
word = MemoryLoad(paddr, WORD)
FR[fd][31:0] = word
```

FLD.D:

```
vaddr = GR[rj] + SignExtend(si12, 32)
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, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(FR[fd][31:0], paddr, WORD)
```

FST.D:

```
vaddr = GR[rj] + SignExtend(si12, 32)
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, an unaligned exception will be triggered when the memory access address is not naturally aligned.

4. Overview of Privileged Resource Architecture

4.1 Privilege Levels

In the Dragon architecture 32-bit simplified version, the processor cores are divided into two privilege levels (Privilege Level, or PLV for short): PLV0 and PLV3.
The current privilege level of a processor core 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 level is PLV3. At this privilege level, privileged instructions cannot be executed to access privileged resources.

For Linux systems, only PLV0 corresponds to kernel mode in the architecture, while PLV3 corresponds to user mode.

4.2 Overview of Privileged Instructions

All privileged instructions are only accessible at the PLV0 privilege level. However, Hit-class CACOP instructions can be executed at the PLV3 privilege level.

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 specified register.
The bits in the CSR that correspond to the write mask being 1 are left unchanged, while the old value of the CSR is updated in the general register.
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.

In the Dragon architecture 32-bit simplified version, all CSR registers are 32 bits wide.

When a CSR access instruction accesses a CSR that is not defined in the architecture or not implemented in the hardware, the read operation returns all zeros, and the write operation does not modify the value.
Any software-visible state of the processor. It should be noted that the CSRWR and CSRXCHG instructions not only include write operations to update the CSR,
It also includes read operations that retrieve old CSR values.

4.2.2 Cache Maintenance Instructions

4.2.2.1CACOP

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.

`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...

Operate the secondary 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. Therefore, using direct address indexing means operating on the

VA[Way-1:0]th path of this cache and the VA[Index+Offset-1:Offset]th cache line.

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.

4.2.3 TLB Maintenance Commands

4.2.3.1TLBSRCH

Command format: tlbsrch

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 of each item in the TLB is calculated by sequentially incrementing the number from 0 to the last row.

4.2.3.2TLBRD

Command format: tlbrd

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 it to 1, and set CSR.ASID.ASID, CSR.TLBEHI, CSR.TLBELO0, CSR.TLBELO1 and CSR.TLBIDX.PS 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.

4.2.3.3TLBWR

Command format: tlbwr

The TLBWR instruction writes page table entry information stored in the TLB's associated CSR to a specified entry in the TLB. The page table entry information being filled comes from...

This applies to CSR.TLBEHI, CSR.TLBELO0, CSR.TLBELO1, and CSR.TLBIDX.PS. If CSR.ESTAT.Ecode=0x3F at this time, then...

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 executing TLBWR, the location where page table entries are written to the TLB is specified by the value of the Index field of CSR.TLBIDX. Specific corresponding rules...

Please refer to the TLBSRCH instruction for the calculation rules of various index values in the TLB.

4.2.3.4TLBFILL

Command format: tlbfill

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.ESTAT.Ecode=0x3F at this time, it is in the state of...

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 would be necessary 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 TLBFILL is executed, the page table entry is randomly selected by the hardware to be filled into which TLB entry.

4.2.3.5INVTLB

Command format: invtlb up, rj, rk

The INVTLB instruction is used to invalidate the contents of the TLB in order to maintain the consistency of page table data between the TLB and memory.

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.

4.2.4 Other Miscellaneous Instructions

4.2.4.1ERTN

Command format: ertn

The ERTN instruction is used to return from exception handling.

Update the PPLV, PIE and other information corresponding to the exception to CSR.CRMD, and at the same time jump to the ERA corresponding to the exception to start fetching the pointer.

The PPLV and PIE information corresponding to the exceptions comes from CSR.PRMD, and the ERA information corresponding to the exceptions 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.

4.2.4.2IDLE

Command format: idle level

After the IDLE instruction completes execution, the processor core will stop fetching instructions and enter a wait state until it is awakened by an interrupt or reset. From the stop state...

After being woken up by an interrupt, 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 32-bit simplified version of the Dragon architecture, PALEN is theoretically a positive integer not exceeding 36, with its specific value determined by the implementation.

The proposed value is 32.

5.2 Virtual Address Space and Address Translation Mode

In the Dragon architecture 32-bit simplified version, the virtual address space is linearly flat. For PLV0 level, the virtual address space size is 2^ 32 bytes.

The Dragon architecture 32-bit simplified 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 see Section 5.2.1. For a detailed explanation of page table mapping modes, please see Section 5.4.

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.

There are two direct mapping configuration windows, which can be used for both instruction fetching and load/store operations simultaneously.

The system software configures two direct-map configuration windows by configuring the CSR.DMW0~CSR.DMW1 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.

In the Dragon Architecture 32-bit Lite version, each Direct Mapping configuration window can be configured with a fixed-size virtual address space of 229 bytes.

When a virtual address hits a valid direct-mapped configuration window, its physical address is directly equal to the [28:0] bits of the virtual address appended to the address of that mapping window.

The high-order bits of the configured physical address. The hit detection method is: the highest 3 bits of the virtual address (bits [31:29]) match the bits [31:29] in the configuration window register.

They are equal, and the current 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.

5.3 Storage Access Types

As mentioned in Section 2.1.7 above, the 32-bit simplified version of the Dragon architecture supports two storage access types: Coherent Cached.

CC (Cardinal Cache) and Strongly-ordered Uncached (SUC).

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.

Save, 2/3 — 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

TLB uses a fully associative lookup table organization.

5.4.2 TLB Entries

The format of each TLB entry is shown in Figure 5-1, which contains two parts: a comparison part and a physical conversion part.

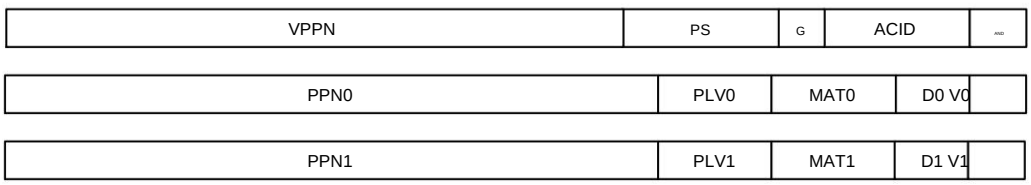


Figure 5-1 **TLB** Entry Format

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 matching the address information, the ASID information also needs to be compared.

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 exponentiation of the power. The Dragon architecture 32-bit simplified version only supports two page sizes: 4KB and 4MB, corresponding to the PS values in the TLB table entries.

12 and 21 1.

ÿ Virtual Double Page Number (VPPN), (VALEN-13) bits. In the Dragon architecture 32-bit simplified version, each page table entry stores an adjacent pair of odd-numbered bytes.

Since the TLB page table entries store virtual page numbers, the virtual page number stored in the TLB page table entries is the virtual page number in the system divided by 2, which is the least significant bit of the virtual page number.

It 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 page or an even-numbered page.

Physical conversion information of the page.

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 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. This page table entry can be accessed by any privilege level not lower than PLV.

Access to the program.

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

Managing the TLB in the 32-bit simplified version of the Dragon architecture involves software aspects, including TLB refilling and consistency between the TLB and memory page tables.

Maintenance is still entirely software-driven.

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 32-bit Dragon Architecture Lite version 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 refill operation is performed. This exception has its own independent exception entry. While the TLB refill exception is trapped, the hardware will automatically...

Setting DA to 1 and PG to 0 in CSR.CRMD automatically enters direct address translation mode, thus avoiding TLB refill exceptions.

If the exception handler itself triggers the TLB to refill the exception again, the exception context will not be saved or restored.

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.

[1] In the Linux kernel, a 4MB page size corresponds to a page table entry in a transparent big page, which is divided into two 2MB entries with the same page table attributes during the TLB filling process.

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. Privilege level non-compliance is manifested as the CSR.CRMD.PLV value of the page table entry being greater than the value in the page table entry.

PLV.

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.

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 section 4.2.3 of this manual.

5.4.3.3 TLB- related CSR

TLB-related CSRs are mainly divided into two categories according to their functions: the first category is used for TLB access and interaction interfaces, and the second category is used for software page table traversal. The third category is only used for TLB refill exceptions.

The first category includes:

ÿ BADV

ÿ OBSERVATION

ÿ EXPECTATION0

ÿ TLBELO1

ÿ TLBIDX

ÿ ACID

The second category includes:

ÿ PGDL

ÿ PGDH

ÿ PGD

The third category includes:

ÿ TLBRENTRY

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 32-bit simplified version allows hardware initialization without implementing TLB, allowing the software during startup to execute "INVTLB 0, r0, r0" to initialize the TLB. Complete this function.

5.4.4 TLB -based virtual-physical address translation process

This section describes the virtual-to-physical address translation process based on TLB.

```
# va: Virtual address to be
searched # 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 # TLB[]: TLB[N] represents the Nth entry in
the TLB # TLB_ENTRIES: Number of entries in the TLB

# Find TLB
tlb_found = 0
for i in range(TLB_ENTRIES) :
    if (TLB[i].E==1) and
      ((TLB[i].G==1) or (TLB[i].ASID==CSR.ASID.ASID)) and
      (TLB[i].VPPN[VALEN-1: TLB[i].PS+1]==va[VALEN-1: TLB[i].PS+1]) :
        if (tlb_found==0) :
            tlb_found = 1
            found_ps = TLB[i].PS
            if (va[found_ps]==0) :
                found_v = TLB[i].V0
                found_d = TLB[i].D0
                found_mat = TLB[i].MAT0
                found_plv = TLB[i].PLV0
                found_ppn = TLB[i].PPN0
            else :
                found_v = TLB[i].V1
                found_d = TLB[i].D1
                found_mat = TLB[i].MAT1
                found_plv = TLB[i].PLV1
                found_ppn = TLB[i].PPN1

        else: # Multiple hits occurred, and the processor's execution result is uncertain.

if (tlb_found==0) :
    SignalException(TLBR)                    #Report TLB re-entry exception

if (found_v==0) :
    case mem_type :
        FETCH : SignalException(PIF) # Reports an invalid fetch operation page exception
```

```
            LOAD : SignalException(PIL)          #Error message: Load operation page invalid exception
            STORE : SignalException(PIS)         #Store action page invalid exception
    elif (plv > found_plv) :
        SignalException(PPI)                     #Exceptions to non-compliant newspaper page privilege levels
    elif (mem_type==STORE) and (found_d==0)): # Write permission check is not enabled
        SignalException(PME)                     #Exception to page modification
    else :
        pa = {found_ppn[PALEN-13:found_ps-12], va[found_ps-1:0]}
        mat = found_mat
    ÿ
```

6 Exceptions and Interruptions

6.1 Interruption

6.1.1 Interrupt Types

The Dragon architecture 32-bit simplified version uses wired interrupts. Each processor core can internally record 12 wired interrupts, namely: 1...

Inter-core interrupt (IPI), 1 timer interrupt (TI), 8 hardware interrupts (HWI0~HWI7), and 2 software interrupts (SWI0~SWI1).

Some line interrupts are level interrupts, and all of them 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 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 Interrupt Priority

The response to multiple interrupts at the same time uses a fixed priority arbitration base address, with higher interrupt numbers having higher priority. Therefore, IPI has the highest priority.

TI is next, ..., SWI0 has the lowest priority.

6.1.3 Interrupt Entry Point

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. Please refer to Section 6.2.1 for the calculation rules for ordinary exception entry points .

6.1.4 Processor hardware interrupt handling procedure

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 multi-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 Exceptions

6.2.1 Exception Entrance

The entry point for TLB refill exceptions comes from CSR.TLBRENTRY.

All other common exceptions, except those mentioned above, share the same entry point: CSR.EENTRY. In this case, the software needs to access CSR.ESTA.

The specific exception type is determined by the information in the Ecode and IS fields.

6.2.2 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: exceptions with incorrect fetch address have the highest priority, followed by exceptions related to the instruction fetch TLB.

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, there may be only memory access instructions or multiple exceptions triggered simultaneously, with their priorities from highest to lowest as follows: memory access instructions requiring address alignment, and so on.

Address misalignment exception (ALE) > TLB related exception 1.

6.2.3 General Process for Exceptional Hardware Handling

Different exceptions may be handled in slightly different ways by the processor hardware. Here is a general handling procedure common to all exceptions.

Describe it.

When an exception is triggered, the processor hardware will perform 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 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 returns from an exception execution by executing the ERTN instruction, 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 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.

[1] The definition of TLB-related exceptions dictates that, under any circumstances, a memory access instruction will only generate a single type of TLB-related exception.

6.3 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=0ÿ

ÿ CSR.EUEN's FPUen is 0;

ÿ The LIE value in CSR.ECFG is 0;

ÿ In CSR.ESTAT, IS[1:0] are all 0;

ÿ CSR.TCFG's En=0;

ÿ CSR.LLBCTL's KLO=0;

ÿ In all implemented CSR.DMW, PLV0 and PLV3 are both 0;

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 a reset is determined by the implementation; if not, a software reset is required.

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
0x4	Exception Configuration	ECFG
0x5	Exceptional state	STATE
0x6	Exception return address	ERA
0x7	Error virtual address	BADV
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
0x20	Processor number	CPUID
0x30~0x33	Data storage	SAVE0~SAVE3
0x40	Timer number	TIME
0x41	Timer configuration	TCFG
0x42	Timer value	TVAL
0x44	Timed interrupt clear	TICLR
0x60	LLBit control	LLBCTL
0x88	TLB Re-entry Exception Address	TLBRENTRY
0x98	Cache tags	CTAG
0x180~0x181	Direct mapping configuration window	DMW0~DMW1

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 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 will return all zeros, and 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 these values to 0 will not change the processor state visible to other software, but software should not actively write these registers to ensure backward compatibility.

Storage.

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 current privilege level of the processor core, global interrupt enable, and address translation mode.

Table 7-2 Current Mode Information Register Definition

Bit	Name reading and writing		describe
1:0	POS	RW	Current privilege level. Its valid values are 0 and 3. 0 represents the highest privilege level, and 3 represents the lowest. Privilege 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, the hardware restores the value of the PPLV field of CSR.PRMD to its original value. here.
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, the hardware restores the value of the PIE field of CSR.PRMD to this value. inside.
3	AND	RW	Enable direct address translation mode, highly effective. When a TLB refill exception is triggered, the hardware sets this field to 1. When the execution of the ERTN instruction returns from the exception handler, if CSR.ESTAT.Ecode=0x3F, the hardware will... Set the 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 is triggered, the hardware sets this field to 0. When the execution of the ERTN instruction returns from the exception handler, if CSR.ESTAT.Ecode=0x3F, the hardware will... Set the 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 the software sets PG to 1, it is recommended to also set the DATF field to 0b01, which is a consistent cacheable type.
8:7	DATM	RW	Storage access type for load and store operations in direct address translation mode. When the software sets PG to 1, it is recommended to also set DATM to 0b01, which is a consistent cacheable type.
31:9	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, the hardware saves the processor core's privilege level and global interrupt enable bit to the pre-exception mode information register.

Used to restore the processor core's state upon exception return.

Table 7-3 Definition of Exception Mode Information Register

Bit	Name reading and writing		describe
1:0	PPLV	RW	When an exception is triggered, the hardware will record the old value of the PLV field in CSR.CRMD in this field. When the ERTN instruction is executed and the system returns from the exception handler, the hardware restores the value of this field to the PLV of CSR.CRMD. domain.
2	ABOUT	RW	When an exception is triggered, the hardware will record the old value of the IE field in CSR.CRMD in this field. When the ERTN instruction is executed and returns from the exception handler, the hardware restores the value of this field to the value in CSR.CRMD. domain.
31:3	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, the basic floating-point instruction set has software-configurable enable bits. When these enable bits are active...

When an instruction is invalid, executing the corresponding instruction will trigger the relevant instruction unavailability exception. The software uses this mechanism to determine the scope of context saving.

Hardware implementations can also utilize the control bits here to control circuit power consumption.

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).
31:1	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

7.4.4 Exception Control (ECFG)

This register is used to control the local enable bits for each interrupt.

Table 7-5 Exception Configuration Register Definitions

Bit	Name reading and writing		describe
9:0	LIE[9:0]	RW	Local interrupt enable bits, active high. These local interrupt enable bits are related to the 10 bits recorded in the IS[9:0] field of CSR.ESTAT. Each interrupt source corresponds to one interrupt source, with each bit controlling one interrupt source.
10	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	
12:11	LIE[12:11]	RW	Local interrupt enable bits, active high. These local interrupt enable bits are related to the 2 bits recorded in the IS[12:11] field of CSR.ESTAT. Each interrupt source corresponds to one interrupt source, with each bit controlling one interrupt source.
31:13	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

7.4.5 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 7-6 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.
9:2	IS[9:2]	R	Interrupt status bits for 8 hardware interrupts (HWI0~HWI7). Active high. 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.
10	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	
11	IS[11]	R	Interrupt status bit for Timer Interrupt (TI). Active high. In in-circuit interrupt mode, the hardware samples only one clock cycle. Each interrupt source records its status here. The requirement that all interrupts must be level interrupts is determined by the interrupt... The source is responsible for ensuring this, but it is not maintained here.
12	IS[12]	R	Inter-core interrupt (IPI) status bit. Active high. In in-circuit interrupt mode, the hardware only samples step-by-step. Each interrupt source records its status here. The requirement that all interrupts must be level interrupts is determined by the interrupt... The source is responsible for ensuring this, but it is not maintained here.
15:13	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 level-one encoding. When an exception is triggered, the hardware will assign the number defined in the Ecode column of Table 7-7 according to the exception type. The value is written to this field.
30:22	EsubCode	R	Exception type secondary encoding. When an exception is triggered, the hardware will assign the EsubCode column defined in Table 7-7 according to the exception type. The value is written into this field.
31	0	R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.	

Table 7-7 Exception Code Table

Ecode	EsubCode Exception Code		Exception types
0x0	0	INT	Interrupted.
0x1	0	PIL	Load operation page invalid exception
0x2	0	PIS	Store action page invalid exception
0x3	0	PIF	Invalid Fetch Operation Page Exception
0x4	0	SMEs	Page modification exceptions
0x7	0	PPI	Page privilege level non-compliance exception
0x8	0	ADEF	Fetch Address Error Exception
	1	ADEM	memory access instruction address error exception
0x9	0	BUT	Exceptions to address misalignment
0xB	0	SYS	System call exceptions

Ecode	EsubCode Exception Code		Exception types
0xC	0	BRK	Breakpoint Exception
0xD	0	I HAVE	The instruction has no exceptions
0xE	0	CALL	Command privilege level error exception
0xF	0	FPD	Floating-point instruction not enabled exception
0x12	0	FPE	Exceptions to basic floating-point instructions
0x1A-0x3E			Reserved code
0x3F	0	TLBR	TLB Refill Exception

7.4.6 Exception Return Address (ERA)

When an exception is triggered, the program counter (PC) of the instruction that triggered the exception will be recorded in this register.

Table 7-8 Exception Return Address Register Definitions

Bit	Name reading and writing	describe
31:0	PC	When an exception is triggered by RW, the hardware records the PC of the instruction that triggered the exception here.

7.4.7 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 TLB Refill Exception

ÿ Instruction Fetch Error (ADEF) exception: In this case, the PC of the instruction is recorded.

ÿ Address alignment misalignment exception (ALE)

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 Privilege Level Non-Compliance Exception (PPI)

Table 7-9 Error Virtual Address Register Definitions

Bit	Name reading and writing	describe
31:0	When a TLB refill exception or an address error-related exception is triggered, the hardware records the erroneous virtual address here, VAddr RW.	

7.4.8 Exception Entry Address (EENTRY)

This register is used to configure the entry addresses for exceptions and interrupts, excluding TLB refill exceptions.

Table **7-10** Exception Entry Address Register Definitions

Bit	Name reading and writing	describe
5:0	0	R is always 0 when read-only, and writes are ignored.
31:6	AND	RW represents bits [31:6] of the exception and interrupt entry address. This means that the lower 6 bits of the exception and interrupt entry address must be 0.

7.4.9 Processor ID (CPUID)

This register contains processor core number information.

Table **7-11** 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.

7.4.10 Data Saving (SAVE0~3)

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.

All data storage control status registers use the same format, as shown in Table 7-12.

Table **7-12** Data Storage Register Definitions

Bit	Name reading and writing	describe
31:0	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.

7.4.11 LLBit Control (LLBCTL)

This register is used for access control operations on LLBit.

Table **7-13 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.

Bit	Name reading and writing		describe
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.
31:3	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.

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 7-14 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 **7-14 TLB** Index Register Definition

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 4.2.3.1.
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. The values of the PS field in the TLB entries written are derived from this when the TLBWR and TLBFILL instructions are executed.
30	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.
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 TLBWR, the value of this bit is inverted and written to the E bit of the TLB entry being written. When executing the TLBWR or TLBFILL instruction, if CSR.ESTAT.Ecode != 0x3F, the value of that bit is inverted and then written. Move to bit E of the TLB entry being written; if CSR.ESTAT.Ecode=0x3F 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 **7-15** Definition of TLB Page Table High-Level Register

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. The TLBSRCH instruction queries the VPPN value used by the TLB, and the TLBWR and TLBFILL instructions write data to the TLB. The value of the VPPN field in the TLB entry comes from this. When triggering TLB refill exception, load operation page invalid exception, store operation page invalid exception, or fetch operation page invalid exception. When an exception occurs, such as a page write permission exception or a page privilege level non-compliance exception, bits [31:13] of the virtual address that triggered the exception are recorded. Recording ends 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 32-bit simplified version of the Dragon architecture uses a double-page structure, the low-order bits of the TLB entry correspond to two physical page entries, one odd and one even.

Odd-numbered page information is stored in TLBELO0, and odd-numbered page information is stored in TLBELO1. The format definitions of the TLBELO0 and TLBELO1 registers are exactly the same.

The definitions of its various domains are shown in Table 7-16.

Execute the TLBWR and TLBFILL instructions to write the following entries to the TLB table: G, PPN0, V0, PLV0, MAT0, D0, PPN1, V1, PLV1.

The values of the MAT1 and D1 fields come from TLBELO0 and TLBELO1, respectively.

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 **7-16 TLB** Entries Low-order Register Definitions

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.
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 **7-17** 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 instruction, it is used to query the ASID key value information of 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 **7-18** Definition of Global Directory Base Address Register in Lower Half Address Space

Bit	Name reading and writing		describe
11:0	0		R is always 0 when read-only, and writes are ignored.
31:12	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 **7-19** Definitions of Global Directory Base Register in High Half-Address Space

Bit	Name reading and writing		describe
11:0	0		R is always 0 when read-only, and writes are ignored.
31:12	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 to read the return value of CSR-type instructions.

Table **7-20** 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.
31:12	Base	R	If the highest bit of CSR.BADV is 0, the read return value is equal to the Base field of CSR.PGDL; otherwise, the read return value is not equal to the Base field of CSR.PGDL. It is equal to the Base field of CSR.PGDH.

7.5.8 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 **7-21 TLB** Refill Exception Entry Address Register Definition

Bit	Name reading and writing		describe
5:0	0	R	TLB refills the exception entry address [5:0]. Read-only is always 0, write is ignored.
31:6	PA	RW	TLB refills the exception entry address [31:6]. The address entered here should be a physical address.

7.5.9 Direct Mapping Configuration Window (DMW0~DMW1)

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 **7-22** Direct Mapping Configuration Window Register Definitions

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.
2:1	0		R0 is a reserved field. Reading it returns 0, and the software is not allowed to change its value.
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 timer, when the software uses the RDCNTID instruction to read the timer ID number, the returned value is the corresponding timer number.

Table **7-23** Timer Number Register Definitions

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 **7-24** 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 be automatically reloaded to the initial value configured in the TimeVal field, and then 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.
31: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 **7-25** Timer Remaining Register Definitions

Bit	Name reading and writing		describe
n-1:0	TimeVal	R is the current timer count value.	
31:n	0	R is always 0 when read-only, and writes are ignored.	

7.6.4 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 **7-26** 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.

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 condition2 : 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 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)
```

8.2.5 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)
```

8.2.6 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)
```

8.2.7 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)
```

8.2.8 Rounding Single-Precision Floating-Point Numbers

```
{bits(32) } FP32_roundToInteger(bits(N) x, bits(2) rm):
    return FP32_roundToIntegralExact(x)
```

8.2.9 Rounding Double-Precision Floating-Point Numbers

```
{bits(64) } FP64_roundToInteger(bits(N) x, bits(2) rm):
    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
RDCNTID.W rj		00000000000000000011000	rj	00000
RDCNTVL.W rd		00000000000000000001100000000		rd
RDCNTVH.W rd		00000000000000000001100100000		rd
ADD.W	rd, rj, rk	00000000000100000	rk	rj rd
SUB.W	rd, rj, rk	00000000000100010	rk	rj rd
SLT	rd, rj, rk	00000000000100100	rk	rj rd
SLTU	rd, rj, rk	00000000000100101	rk	rj rd
NOR	rd, rj, rk	00000000000101000	rk	rj rd
AND	rd, rj, rk	00000000000101001	rk	rj rd
OR	rd, rj, rk	00000000000101010	rk	rj rd
FREE	rd, rj, rk	00000000000101011	rk	rj rd
SLL.W	rd, rj, rk	00000000000101110	rk	rj rd
SRL.W	rd, rj, rk	00000000000101111	rk	rj rd
SRA.W	rd, rj, rk	00000000000110000	rk	rj rd
MUL.W	rd, rj, rk	00000000000111000	rk	rj rd
MULH.W	rd, rj, rk	00000000000111001	rk	rj rd
MULH.WU	rd, rj, rk	00000000000111010	rk	rj rd
DIV.W	rd, rj, rk	00000000001000000	rk	rj rd
MOD.W	rd, rj, rk	00000000001000001	rk	rj rd
DIV.WU	rd, rj, rk	00000000001000010	rk	rj rd
MOD.WU	rd, rj, rk	00000000001000011	rk	rj rd
BREAK	code	00000000001010100	code	
SYSCALL	code	00000000001010110	code	
SLLI.W	rd, rj, ui5	00000000010000001	ui5	rj rd
SRLI.W	rd, rj, ui5	00000000010001001	ui5	rj rd
SRAI.W	rd, rj, ui5	00000000010010001	ui5	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

	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
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
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
FCLASS.S fd, fj	0000000100010100001101 fj fd
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
FMOV.S fd, fj	0000000100010100100101 fj fd
FMOV.D fd, fj	0000000100010100100110 fj fd
MOVGR2FR.W fd, rj	0000000100010100101001 rj fd
MOVGR2FRH.W fd, rj	0000000100010100101011 rj fd
MOVFR2GR.S rd, fj	0000000100010100101101 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 0 0 cd
MOVCF2FR fd, cj	0000000100010100110101 00 cj fd
MOVGR2CF cd, rj	0000000100010100110110 rj 0 0 cd
MOVCF2GR rd, cj	0000000100010100110111 00 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
FTINTRP.W.S fd, fj	0000000100011010010001 fj fd
FTINTRP.W.D fd, fj	0000000100011010010010 fj fd
FTINTRZ.W.S fd, fj	0000000100011010100001 fj fd
FTINTRZ.W.D fd, fj	0000000100011010100010 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
FTINT.W.S	fd, fj	0000000100011011000001 fj fd
FTINT.WD	fd, fj	0000000100011011000010 fj fd
FFINT.S.W	fd, fj	0000000100011101000100 fj fd
FFINT.D.W	fd, fj	0000000100011101001000 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
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
TLBSRCH		00000110010010000010100000000000
TLBRD		00000110010010000010110000000000
TLBWR		00000110010010000011000000000000
TLBFILL		00000110010010000011010000000000
ERTN		00000110010010000011100000000000
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 00 cd
FCMP.cond.D cd, fj, fk		000011000010 cond fk fj 00 cd
FSEL	fd, fj, fk, ca	00001101000000 that fk fj fd
LU12I.W	rd, si20	0001010 si20 rd
PCADDU12I	rd, si20	0001110 si20 rd
LL.W	rd, rj, si14	00100000 yes14 rj rd
SC.W	rd, rj, si14	00100001 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
ST.B	rd, rj, si12	0010100100 si12 rj rd
ST.H	rd, rj, si12	0010100101 si12 rj rd

		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
ST.W	rd, rj, si12	0010100110 si12 rj rd
LD.BU	rd, rj, si12	0010101000 si12 rj rd
LD.HU	rd, rj, si12	0010101001 si12 rj rd
PRELD	hint, rj, si12	0010101011 si12 rj hint
FLD.S	fd, rj, si12	0010101100 si12 rj fd
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
DBAR	hint	00111000011100100 hint
VALLEY	hint	00111000011100101 hint
BCEQZ	cj, offs	010010 offs[15:0] 00 cj offs[20:16]
BCNEZ	cj, offs	010010 offs[15:0] 01 cj offs[20:16]
JIRL	rd, rj, offs	010011 offs[15:0] rj rd
B	offs	010100 offs[15:0] offs[25:16]
BL	offs	010101 offs[15:0] offs[25:16]
BEQ	rj, rd, offs	010110 offs[15:0] rj rd
BNE	rj, rd, offs	010111 offs[15:0] rj rd
BLT	rj, rd, offs	011000 offs[15:0] rj rd
BGE	rj, rd, offs	011001 offs[15:0] rj rd
BLTU	rj, rd, offs	011010 offs[15:0] rj rd
BGEU	rj, rd, offs	011011 offs[15:0] rj rd