LoongArch Reference Manual

Volume 1: Basic Architecture

Loongson Technology Corporation Limited

Version 1.10

Table of Contents

List of Figures

List of Tables

About this manual

Copyright Statement

Disclaimer

Loongson Technology Corporation Limited

Building No.2, Loongson Industrial Park,
Zhongguancun Environmental Protection Park, Haidian District, Beijing

Tel: 010-62546668

Fax: 010-62600826

Reading Guide

This is the first volume of *LoongArch Reference Manual*, which describes the basic part of the LoongArch architecture.

Translator's Note

These documents were translated by Yanteng Si and Feiyang Chen.

This is the translation of https://github.com/loongson/LoongArch-Documentation/releases/latest/download/LoongArch-Vol1-v1.00-CN.pdf.

Due to the limited knowledge of the translators, there are some inevitable errors and omissions existing in this document, please feel free to correct.

License

Contributors

Since the release of the project, we have gotten several errata and content changes donated. Here are all the people who have contributed to LoongArch Documentation as an open source project. Thank you everyone for helping make this a better book for everyone.

The contributors are listed in alphabetical order.

```
Chao LI <lichao@loongson.cn>
```

```
Chenghua Xu <xuchenghua@loongson.cn>
Dandan Zhang <zhangdandan@loongson.cn>
Feiyang Chen <chenfeiyang@loongson.cn>
FreeFlyingSheep <fyang.168.hi@163.com>
Konstantin Romanov <konstantinsromanov@gmail.com>
LI Chao <lichao@loongson.cn>
limeidan <limeidan@loongson.cn>
liuzhensong <liuzhensong@loongson.cn>
mengqinggang <mengqinggang@loongson.cn>
Qi Hu <huqi@loongson.cn>
qmuntal <quimmuntal@gmail.com>
tangxiaolin <tangxiaolin@loongson.cn>
WANG Xuerui <git@xen0n.name>
wangguofeng <wangguofeng@loongson.cn>
Wu Xiaotian <wuxiaotian@loongson.cn>
Wu Xiaotian <yetist@gmail.com>
Xi Ruoyao <xry111@mengyan1223.wang>
Yang Yujie <yangyujie@alumni.sjtu.edu.cn>
Yang Yujie <yangyujie@loongson.cn>
Yanteng <siyanteng@loongson.cn>
Yanteng Si <siyanteng@loongson.cn>
```

Chapter 1. Introduction

The **LoongArch** architecture (LoongArch) is an **I**nstruction **S**et **A**rchitecture (ISA) that has **R**educed **I**nstruction **S**et **C**omputer (RISC) style. The LoongArch Reference Manual is used to explain the LoongArch specification. This is the first of three volumes, which describes the basic part of LoongArch.

1.1. Overview of LoongArch ISA

LoongArch has the typical characteristics of RISC. LoongArch instructions are of fixed size and have regular instruction formats. Most of the instructions have two source operands and one destination operand. LoongArch is a load-store architecture; this means only the load/store instructions can access memory the operands of the other instructions are within the processor core or the immediate number in the instruction opcode.

LoongArch is divided into two versions, the 32-bit version (LA32) and the 64-bit version (LA64). LA64 applications are “application-level backward binary compatibility” with LA32 applications. That means LA32 applications can run directly on the machine compatible with LA64, but the behavior of system softwares (such as the kernel) on the machine compatible with LA32 is not guaranteed to be the same as on the machine compatible with LA64.

LoongArch is composed of a basic part (Loongson Base) and an expanded part, as shown in the figure. The expansion part includes **L**oongson **B**inary **T**ranslation (LBT), **L**oongson **V**irtuali**Z**ation (LVZ), **L**oongson **S**IMD E**X**tension (LSX), and **L**oongson **A**dvanced **S**IMD E**X**tension(LASX).

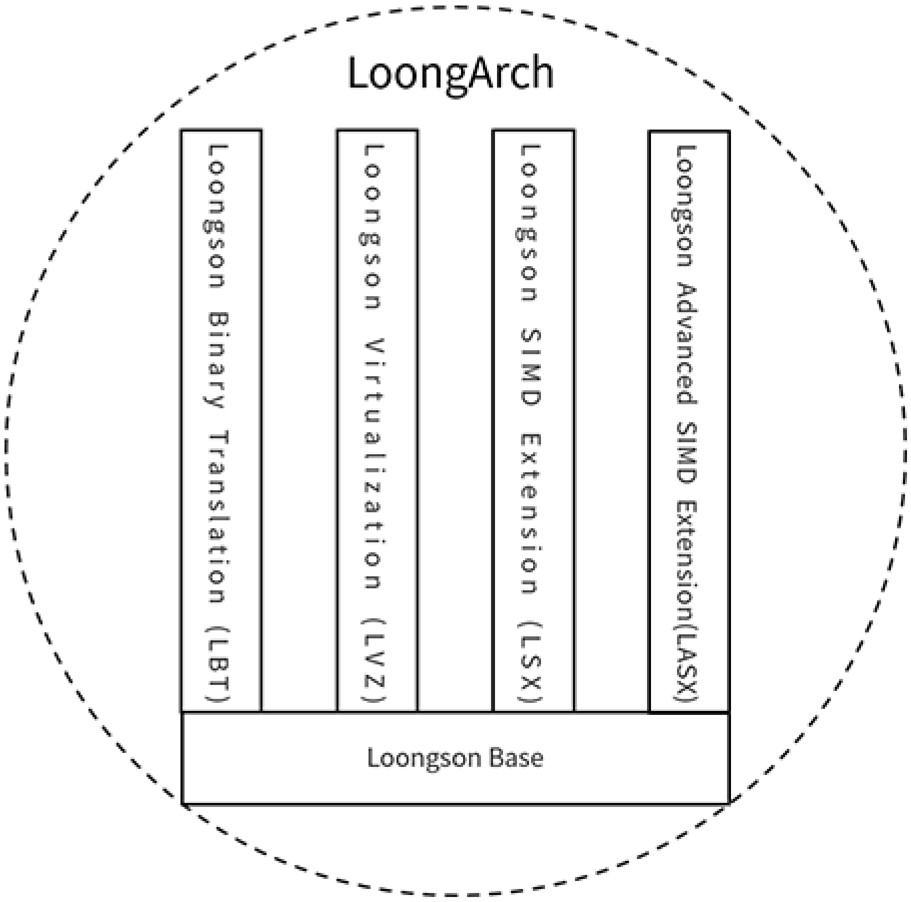
LoongArch
Loongson Binary Translation (LBT)
Loongson Virtualization (LVZ)
Loongson SIMD Extension (LSX)
Loongson Advanced SIMD Extension(LASX)
Loongson Base

Figure 1. LoongArch components

The basic part of LoongArch includes an non-privileged instruction set and a privileged instruction set. The non-privileged instruction set defines commonly used integer and floating-point instructions, which can adequately support the current mainstream compiler to generate efficient target codes.

The virtualization extension part of LoongArch is used for operating system virtualization to provide hardware acceleration to improve performance. This part involves basically all privileged resources, including some privileged instructions and control and status registers, functions added in exceptions and interrupts, memory management, and so on.

The binary translation extension part of LoongArch is used to improve the execution efficiency of the cross-instruction system binary translation on the LoongArch platform. It expands on the basic part and also includes two parts, the non-privileged instruction set and the privileged instruction set.

LoongArch vector instruction extension and advanced vector instruction extension both use SIMD instructions to accelerate CPU-bound applications. They are basically the same in terms of instruction functions. The difference is that the vector length of the vector instruction extension operation is 128 bits and the vector length of the advanced vector instruction extension operation is 256 bits.

For the architecture compatible with LoongArch, the basic part of the LoongArch must be implemented, and the extended part can be implemented optionally. Each extension part can be selected flexibly, but when choosing to implement LASX, LSX must be implemented. Some optional subsets of functions are included in the basic part and each extension part. The software can detect whether these optional functions are implemented via the `CPUCFG` instruction.

The follow-up evolution of the LoongArch adopts a “fine-grained incremental evolution” method. The so-called “fine-grained” means that each functional subset in the basic part or the extended part can evolve independently. The so-called “incremental” means that for any part that can be evolved independently, the higher version is always forward binary compatible[1] with the lower version.

Starting from Chapter 2 of this manual, the specification of the LoongArch will be described in detail. The contents of Chapter 2 and 3 involve the non-privileged instruction set part of the architecture, including the function definitions of basic integer instructions and basic floating-point instructions and their application-level programming models. Chapters 4 to 7 are used to describe the privileged resources in the architecture, mainly including the introduction of privileged instructions, control and status registers, function specifications in operating modes, exceptions and interrupts, memory management, and etc. The pseudo-code descriptions designed to describe the function definitions of instructions are concentrated in Appendix A. The specific coding definitions of the instructions involved are listed in Appendix B.

1.2. Instruction formats

All LoongArch instructions are fixed 32 bits and required to be aligned on 4-byte boundaries. If the address of an instruction is not aligned, address error exception will be triggered.

The style of instruction encoding is that all register operand fields are placed in order from low to high starting from the 0th bit, while the opcode field is placed in order from the 31st bit from high to low. The immediate field, which has different lengths according to different instruction types, is located between the register field and the opcode field if the instruction contains an immediate operand. Specifically, it contains 9 typical instruction formats, including 3 formats without immediate data (2R, 3R, and 4R), and 6 formats with immediate data (2RI8, 2RI12, 2RI14, 2RI16, 1RI21, and I26). The table below lists the specific definitions of these 9 typical formats. There are a few instructions whose encoding style is not completely equivalent to these 9 typical instruction formats. However, the number of such instructions is small and the instructions change little, which will not be inconvenient for compiler developers.

Table 1. Typical Instruction Formats in LoongArch

	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
2R-type	`opcode`																						`rj`					`rd`				
3R-type	`opcode`																	`rk`					`rj`					`rd`				
4R-type	`opcode`												`ra`					`rk`					`rj`					`rd`				
2RI8-type	`opcode`														`I8`								`rj`					`rd`				
2RI12-type	`opcode`										`I12`												`rj`					`rd`				

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
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. Assembly Instruction Mnemonic Formats

The assembly instruction mnemonic mainly includes the instruction name and the operand. LoongArch considers the prefix and suffix of instruction names and operands to make it easier for assembly programmers and compiler developers to use.

First, non-vector instructions and vector instructions, as well as integer and floating-point instructions, can be distinguished by the prefix of instruction name. The instruction name of a 128-bit vector instruction begins with the letter `V`; the instruction name of a 256-bit vector instruction begins with the letter `XV`. The instruction name of a non-vector floating-point instruction begins with the letter `F`; the instruction name of a 128-bit vector floating-point instruction begins with `VF`; the instruction name of a 256-bit vector floating-point instruction begins with `XVF`.

Secondly, most instructions use a suffix in the form of `.XX` in the instruction name to indicate the operand type of the instruction. This form of suffix is only used to characterize the type of the instruction operand. If the operand is an integer, the suffixes of the instruction name include `.B` (signed byte), `.H` (signed half word), `.W` (signed word), `.D` (signed double word), `.BU` (unsigned byte), `.HU` (unsigned half word), `.WU` (unsigned word), and `.DU` (unsigned double word). An exception is that if whether the operand is signed or unsigned does not affect the result of the operation, the suffix of the instruction name will not carry `U`. In this case, the suffix does not limit the operand to the signed number. If operand is a floating-point number, the suffixes of the instruction name are `.H` (half precision), `.S` (single-precision), `.D` (double-precision), `.W` (signed word), `.L` (signed double word), `.WU` (unsigned word), `.LU` (unsigned double word). In addition, for instructions involving vector operations, the suffix `.V` of the instruction name indicates that the entire vector data is operated as a whole. An exception is that if the length of the operand of an instruction is determined by whether the processor is 32-bit or 64-bit, the instruction name has no suffix, such as `SLT` instruction and `SLTU` instruction. Privileged instructions for operating CSRs, TLB, and Cache, and instructions for moving data between different register files have no suffix.

If the length and sign of the source operand and the destination operand are the same, the instruction name will have only one suffix. If the length and sign of all source operands are the same, but not the same as the destination operand, the instruction name will have two suffixes. From left to right, the first suffix decorates the destination operand, and the second suffix decorates the source operand. If the source operation and destination operand are more complicated, the instruction name will list the destination operand and each source operand in order from left to right. The order is consistent with the order of the subsequent operands in the instruction mnemonic. For example, in the instruction `MULW.D.WU rd, rj, rk`, `.D` decorates the destination operand `rd`, and `.WU` decorates the source operands `rj` and `rk`; this means that the multiplication is to multiply two unsigned words to obtain a double word result which will be written into `rd`. For another example, in the instruction `CRC.WBW rd, rj, rk`, the first `.W` decorates `rd`, `.B` decorates `rj`, and the second `.W` decorates `rk`; this means that the CRC check operation is to use the byte message in `rj` and the 32-bit original check value in `rk` to generate a new 32-bit check value which will be written into `rd`.

Register operands distinguish register files by the first letter. `rN` refers to general registers; `fN` refers to floating-point registers; `vN` refers to 128-bit vector registers; `xN` refers to 256-bit vector registers. Among them, `N` is a number that represents the `N`th register.

1.4. Conventions Used in this Manual

1.4.1. Instruction Abbreviation

There are many instructions defined by LoongArch that appear frequently and have similar behaviors. Generally, they only have some differences in operands. For the convenience of readers, such instructions are often introduced together. For the sake of brevity, this manual uses abbreviation rules for the instruction name. `{A/B/C}` means to use `A`, `B`, and `C` to combine the instruction name. `A[B]` means to use `A` and `AB` to combine the instruction name. For example, `ADD.{W/D}` represents two instruction names `ADD.W` and `ADD.D`, while `BLT[U]` represents two instruction names `BLT` and BLTU. A more complicated example is `ADD[I].{W/D}`, which represents four instruction names `ADD.W`, `ADD.D`, `ADDI.W` and `ADDI.D`. Even though instruction names can be abbreviated, it does not mean that their opcode fields have similar contents.

1.4.2. References to Control and Status Registers

LoongArch defines a series of **C**ontrol and **S**tatus **R**egisters (CSRs), which are used to control the execution behavior of instructions. Each CSR usually contains several fields. This manual use `CSR.%%%.####` to refer to the `####` field of the control and status register whose name is abbreviated as `%%%`. For example, `CSR.CRMD.PLV` represents the `PLV` field in the `CRMD` register. When the virtualization extension is implemented, there are two sets of CSRs in the processor, one belongs to the Host and the other belongs to the Guest. If the two sets of CSRs cannot be distinguished by the context, `CSR.XXXX` is used to represent the CSRs of the Host and `GCSR.XXXX` is used to represent the CSRs of the Guest.

1.5. Version Evolution

The initial version of LoongArch is V1, denoted as LoongArch V1. The content of the standard is not specified in the LoongArch Reference Manual and belongs to LoongArch V1 by default. Since LoongArch V1, the subsequent evolution of LoongArch adopts the method of fine-grained incremental evolution. By "fine-grained" evolution, I mean that each subset of functionality in the base or extensions can evolve independently; By "incremental" I mean that a higher version is always compatible with the previous binary for any part that can be evolved independently. In order to more concisely reflect the stages of the above architecture evolution process, a number of new feature subset extensions added in a certain stage are collectively referred to as a new version extension. For example, the new hardware page table traversal support, byte/half-word atomic memory access instructions, and other additions to LoongArch V1 are collectively referred to as LoongArch V1.1. It should be pointed out that the subset of features added in each new version has its own identifier in the CPUCFG instruction return value. It is recommended that the software use this information rather than the version number of the Godson architecture to determine the supported features of the running processor. Architecture specifications do not require processor hardware to implement functions that directly reflect the supported architecture version number.

1.5.1. New In LoongArch V1.1

LoongArch V1.1 adds the following:

1. New instructions for approximately solving floating-point root and inverse floating-point root, including FRECIPE.S, FRECIPE.D, FRSQRTE.S, FRSQRTE.D for scalar operations. VFRECIPE.S, VFRECIPE.D, VFRSQRTE.S, VFRSQRTE.D commands for 128-bit SIMD operations and XVFRECIPE.S, XVFRECIPE.D, XVFRSQRTE.S, XVFRSQRTE.D commands for 256-bit SIMD operations instructions.
2. Add SC.Q instruction.
3. Add LLACQ.W, SCREL.W, LLACQ.D, SCREL instructions.
4. Add AMCAS.B, AMCAS.H, AMCAS.W, AMCAS.D, AMCAS_DB.B, AMCAS_DB.H, AMCAS_DB.W,

AMCAS_DB.D, AMSWAP.B, AMSWAP.H instructions.

5. Add AMADD.B, AMADD.H, AMSWAP_DB.B, AMSWAP_DB.H, AMADD_DB.B, AMADD_DB.H instructions.
6. Add the function definition of non-zero hint value in the dbar instruction part.
7. A new method for determining whether 32-bit integer division instructions on 64-bit machines are affected by the higher 32-bit value of the source operand register.
8. Standardize the way to determine the sequential execution behavior of load memory access operations at the same address.
9. Add the definition of a message interrupt.
10. Hardware page table traversals are allowed.

1. Translator's note: Forward compatibility here may be ambiguous.

Chapter 2. Basic Integer Instructions

According to the context of the software runtime, the non-privileged instruction set of the basic part of LoongArch includes basic integer instructions and basic floating-point instructions. This chapter will describe the integer instruction part. The basic integer instruction part is the most basic part of the non-privileged instruction subset.

2.1. Programming Model of Basic Integer Instruction

The basic integer instruction programming model described in this section only involves the operating environment of the application software, which is always related to some privileged resources. Therefore, the concept of privileged resources will be introduced where necessary to ensure the completeness of the description. Although the content of privileged resources is covered here, it will not be expanded in detail. Readers who need a comprehensive and in-depth understanding can refer to the relevant chapters in the manual according to the prompts in the text.

2.1.1. Data Types

There are 5 data types operated by basic integer instructions, namely: **b**it (b), **B**yte (B, length 8b), **H**alfword (H, length 16b), **W**ord (W, length 32b), **D**oubleword (D, length 64b). In LA32, there are no integer instructions for operating doubleword. Byte, half-word, word and double-word data types all use two’s complement encoding.

2.1.2. Registers

The registers involved in basic integer instructions include General Registers (GR) and Program Counters (PC), as shown in the figure.

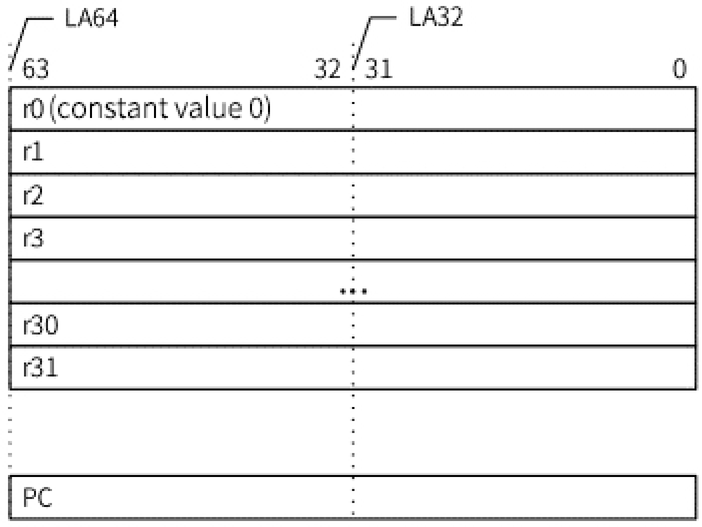

Figure 2. GR and PC

2.1.2.1. General-purpose Registers

There are 32 **G**eneral purpose **R**egisters (GR), denoted as `r0`-`r31`, and the value of register `r0` is always `0`. The length of GR is recorded as GRLEN. The length of GR in LA32 is 32 bits, and the length of GR in LA64 is 64 bits. There is an orthogonal relationship between basic integer instructions and general registers. That is, from an architectural point of view, any register operand in this instruction can use any of the 32 GRs. The only exception is that the destination register implicit in the `BL` instruction must be `r1`. In the standard LoongArch **A**pplication **B**inary **I**nterface (ABI), `r1` is as storing the return address of a function call.

2.1.2.2. PC

There is only one PC, which records the address of the current instruction. The `PC` register cannot be modified directly by instructions, it can only be modified indirectly by branch instructions, exception trap and exception return instructions. However, the `PC` register can be directly read as the source operand of some non-branch instructions. The length of `PC` is always the same as the length of GR.

2.1.3. Running Privilege Levels

LoongArch defines 4 running **P**rivilege **L**e**V**els (PLV), namely PLV0-PLV3. The specific privilege level of the application is determined by the system software at runtime, and the application cannot accurately aware this. In LoongArch, the application usually runs on PLV3. For more information about privilege levels, see Privilege Levels.

2.1.3.1. Privileged Resources Accessible by Applications

Generally speaking, privileged resources cannot be directly accessed by application running at a non-privileged level, but when `RPCNTL1`/`RPCNTL2`/`RPCNTL3` in `CSR.MISC` is set, the `CSRRD` instruction can be executed at the privilege level of PLV1/PLV2/PLV3 to read performance monitor counters. For more information about performance monitor counters, see Control and Status Registers Related to Performance Monitoring.

2.1.3.2. Disabling of Some Non-privileged Functions

Some non-privileged functions that are enabled by default after power-on reset can be disabled by the system software during execution. By setting the `DRDTL1`/`DRDTL2`/`DRDTL3` bits in `CSR.MISC` to `1`, the execution of `RDTIME` instructions at the PLV1/PLV2/PLV3 level can be prohibited, or will trigger the **I**nstruction **P**rivilege error **E**xception (IPE).

2.1.4. Exceptions and interrupts

Exceptions and interrupts will interrupt the currently executing program and switch the control flow to the entry of the exception/interrupt handler to start execution. Exceptions are caused by abnormal conditions that occur during the execution of the instruction, and interrupts are caused by external events (such as interrupt signal input). In the manual, it will strictly distinguish the two concepts of “generating an exception/interrupt” and “triggering an exception/interrupt”. The difference between the two is that the former does not necessarily cause a change in the control flow, while the latter must change the current control flow to an entry point of the exception/interrupt handler.

The handling specifications for exceptions and interrupts belong to the privileged resource handling part of the architecture. Here is a brief introduction to the exceptions that the application can perceive.

- **SYS**tem call exception (SYS): the execution of the SYSCALL instruction will trigger the system call exception immediately.
- **B**r**E**a**K**point exception (BEK): executing the BREAK instruction will trigger a breakpoint exception immediately.
- **I**nstruction **N**on-defined **E**xception (INE): if the executed instruction code is not defined in the architecture, or the architecture specification defines the instruction as not existing in the current context, then the instruction non-defined exception will be triggered immediately.
- **I**nstruction **P**rivilege error **E**xception (IPE): in addition to the special circumstances listed in Running Privilege Levels, executing a privileged instruction in the application software will definitely trigger the instruction privilege level error exception immediately.
- **AD**dress error **E**xception (ADE): when the program has a functional error that causes the address of the instruction fetch or memory access instruction to appear illegal (such as the instruction fetch address is not aligned on 4-byte boundaries, and the privileged address space is accessed), **AD**dress error **E**xception for **F**etching instructions (ADEF) or **AD**dress error **E**xception for **M**emory access instructions

(ADEM) will be triggered.

- **F**loating-**P**oint error **E**xception (FPE): when the floating-point number instruction is executed, special processing is required for data exceptions, which can generate or trigger the basic floating-point error exception. See Floating-Point Move Instructions for more information.

2.1.5. Memory Address Space

Only the virtual address space visible to the application is involved here. The translation of virtual memory addresses to physical memory addresses is determined by the runtime environment. These contents relate to the relevant specifications of privileged resources in the architecture and will be introduced in the second half of this manual. The memory address space on LoongArch is a continuous linear address space, which is addressed in bytes.

In LA32, the specification of the memory address space that application can access is: `0`-2^{31}`-1`.

In LA64, the range of memory address space accessible by application is: `0`-$2^{\mathtt{VALEN-1}}$`-1`. Generally `VALEN` is in the range of `[40,48]`. Application can determine the specific value of `VALEN` by executing the `CPUCFG` instruction to read the `VALEN` field of the `0x1` configuration word.

When the virtual address of the instruction fetch or memory access instruction in the application exceeds the above range, ADEF or ADEM will be triggered.

2.1.6. Endian

LoongArch bit designations are always little-endian.

2.1.7. Memory Access Types

LoongArch supports three types of memory access: **C**oherent **C**ached (CC), **S**trongly-ordered **U**n**C**ached (SUC) and **W**eakly-ordered **U**n**C**ached (WUC). The memory access type used for a location is associated with the virtual address, which is determined by the Memory Access Type field. The relationship of the memory access type and `MAT` field is: `0` - SUC, `1` - CC, `2` - WUC, and `3` - reserved. The memory access type setting process is transparent to the application.

When using consistent cacheable access type, the accessed object can be either the final memory object or the caches. This type of access is usually used to access faster.

When using SUC or WUC access, the final memory object can only be directly accessed. The difference between the two is: SUC access meets sequential consistency, that is, all accesses are executed in strict accordance with the order in the program and the next memory access operation cannot be started before the current memory access operation is completely completed. While the WUC read access allows speculative execution, and WUC written data can be merged inside the processor core to a larger scale (such as a Cache line) and then written out in a burst mode. Subsequent writes in the merge process can overwrite the data written earlier.

In LoongArch, only SUC memory access instructions must not have side effects, that is, such instructions cannot be predictive executed. Software can use this feature to access I/O devices in the system through SUC type memory access instructions. However, LoongArch allows SUC fetch instruction operations to have side effects. This means that the access type is a SUC type of fetch instruction operation, even if it originates from the result of branch prediction, it is allowed to be executed. In order to prevent the out-of-core memory access operations generated by such speculative execution from erroneously entering the illegal physical address space, it is necessary to filter out the risky accesses, This will be done on the chip.

The WUC type of access is usually used to accelerate the access to UC memory data, such as video memory data.

2.1.7.1. Cache Coherency Maintenance of Instruction Cache

The Cache coherency between the instruction Cache of a certain processor core and the Cache in other processor cores or Cache Coherenr I/O Master must be maintained by hardware.

The Cache coherency maintenance between the instruction Cache and the data Cache within the processor core can be implemented as hardware maintenance. This means that for the self-modifying code, the software does not need to use the `CACOP` instruction to maintain the Cache coherency between the instruction Cache and the data Cache within the same core. However, due to the pipeline structure and speculative instruction fetching behavior, the software still needs to use the `IBAR` instruction to ensure that the instruction fetching must be able to see the execution effect of the store instruction.

2.1.8. Unaligned Memory Access

The fetch addresses of all instruction fetches must be aligned on 4-byte boundaries, otherwise the ADEF will be triggered.

Except for atomic memory access instructions, integer bound check memory access instructions and floating-point bound check memory access instructions, other load/store memory access instructions can be implemented to allow memory access addresses to be unaligned. However, in an implementation that allows memory access address misalignment, the system mode software can configure the `ALCL0`-`ALCL3` control bits in `CSR.MISC` to address these load/store memory access instructions at the privilege levels of PLV0-PLV3. Alignment check is needed, too. For memory accessed instructions that require address alignment checks, if the address accessed is not naturally aligned, an **A**ddress a**L**ignment fault **E**xception (ALE) will be triggered.

2.1.9. Overview of Memory Consistency

The memory consistency model of the LoongArch uses the **W**eak **C**onsistency (WC) model. This section only gives a brief description of the weak consistency model adopted by the architecture.

In the weak consistency model, synchronization operations need to be distinguished from ordinary memory accesses. The programmer must use the synchronization operations defined by the architecture to protect the access to the write shared unit to ensure that multiple processor cores have access to the write shared unit mutually exclusive. The following restrictions are imposed on the sequence of memory access events:

- The execution of the synchronization operation satisfies the sequence consistency condition. That is, synchronization operations are executed in all processor cores strictly in the order in which they appear in the program, and the next synchronization operation cannot be started until the current synchronization operation is completely completed.
- Before any ordinary memory access operation is allowed to be executed, all synchronization operations prior to this memory access operation in the same processor core have been completed.
- Before any synchronization operation is allowed to be executed, all ordinary memory access operations that precede this synchronization operation in the same processor have been completed.

The instructions that can generate synchronous operations in LoongArch include `DBAR`, `IBAR`, `AM` atomic memory access instructions with `DBAR` function, and `LL`-`SC` instruction pairs.

2.2. Overview of Basic Integer Instructions

This section will describe the functions of application-level basic integer instructions in LA64. For LA32, it only needs to implement a subset of them. The instruction list contained in this subset is shown in the table. Because the length of GR in LA32 is only 32 bits, the sign extension operation in “sign extend the 32-bit result into the general register `rd`” in the subsequent instruction description is not required.

Table 2. Application-level basic integer instructions in LA32

Arithmetic operation instructions	ADD.W, SUB.W, ADDIW, ALSL.W, LU12L.W, SLT, SLTU, SLTI, SLTUI, PCADDI, PCADDU12I, PCALAU12I, AND, OR, NOR, XOR, ANDN, ORN, ANDI, ORI, XORI, MUL.W, MULH.W, MULH.WU, DIV.W, MOD.W, DIV.WU, MOD.WU
Bit-shift instructions	SLL.W, SRL.W, SRA.W, ROTR.W, SLLI.W, SRLI.W, SRAI.W, ROTRI.W
Bit-manipulation instructions	EXT.W.B, EXT.W.H, CLO.W, CLZ.W, CTO.W, CTZ.W, BYTEPICK.W, REVB.2H, BITREV.4B, BITREV.W, BSTRINS.W, BSTRPICK.W, MASKEQZ, MASKNEZ
Branch instructions	BEQ, BNE, BLT, BGE, BLTU, BGEU, BEQZ, BNEZ, B, BL, JIRL
Memory access instructions	LD.B, LD.H, LD.W, LD.BU, LD.HU, ST.B, ST.H, STW, PRELD
Atomic memory access instructions	LL.W, SC.W
Barrier instructions	DBAR, IBAR
Other instructions	SYSCALL, BREAK, RDTIMEL.W, RDTIMEH.W, CPUCFG

In addition, for those instructions whose data length of the operation object is GR length, the operation length is 32 bits in LA32 and 64 bits in LA64. Unless there are special circumstances, no special instructions will be given in the instruction function description.

2.2.1. Arithmetic Operation Instructions

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

Instruction formats:

```
add.w   rd, rj, rk
add.d   rd, rj, rk
sub.w   rd, rj, rk
sub.d   rd, rj, rk
```

The ADD.W instruction performs the operation that the [31:0] bit data in the general register rj plus the [31:0] bit data in the general register rk; the resultant [31:0] bit is sign extension, then written into the general register rd.

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

The SUB.W instruction performs the operation that the [31:0] bit data in the general register rk minus the [31:0] bit data in the general register rj; the resultant [31:0] bit is sign extension, then written into the general register rd.

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

The ADD.D instruction performs the operation that the [63:0] bit data in the general register rj plus the [63:0] bit data in the general register rk; the result is written into the general register rd.

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

The SUB.D instruction performs the operation that the [63:0] bit data in the general register rj minus the [63:0] bit data in the general register rk; writes the result into the general register rd.

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

When the above instructions are executed, no special handling will be done on overflow.

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

Instruction formats:

```
addi.w      rd, rj, si12
addi.d      rd, rj, si12
addu16i.d   rd, rj, si16
```

The ADDI.W instruction performs the operation that the [31:0] bit data in the general register rj plus the 12-bit immediate si12 sign extension 32-bit data; the resultant [31:0] bit is sign extension, then written into the general register rd.

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

The ADDI.D instruction performs the operation that the [63:0] bit data in the general register plus to the 64-bit data after 12-bit immediate si12 sign-extension; the result is written into the general register rd.

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

ADDU16I.D shifts the 16-bit immediate si16 logic to the left by 16 bits and then sign extensions the resultant data, the result plus [63:0] bit data in the general register rj, and the result of the addition is written into the general register rd. The ADDU16I.D instruction is used in conjunction with the LDPTR.W/D and STPTR.W/D instructions to accelerate GOT table-based access in position-independent codes.

```
ADDU16I.D:
    tmp = GR[rj][63:0] + SignExtend({si16, 16'b0}, 64)
    GR[rd] = tmp[63:0]
```

When the above instructions are executed, no special handling will be done on overflow.

2.2.1.3. ALSL.{W[U]/D}

Instruction formats:

```
alsl.w  rd, rj, rk, sa2
alsl.d  rd, rj, rk, sa2
alsl.wu rd, rj, rk, sa2
```

The ALSL.W instruction performs the operation that logical shift the [31:0] bit data in the general register rj to the left (sa2 + 1) and it plus the [31:0] bit data in the general register rk; then write the result into the general register rd after the sign extension.

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

ALSL.WU logical shift the [31:0] bit data in the general register rj to the left (sa2 + 1) bit and it plus the [31:0] bit data in the general register rk; then the result is [31:0] bit zero after expansion, write to general register rd.

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

The ALSL.D instruction performs the operation that logical shift the [63:0] bit data in the general register rj (sa2 + 1) to the left and it plus the [63:0] bit data in the general register rk; then the result is written into the general register rd.

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

When the above instructions are executed, no special handling will be done on overflow.

TIP When writing assembly, you need to fill in the immediate field with the **real shift value**, i.e. `(sa2+1)`, not the value in the immediate field of the instruction code.

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

Instruction formats:

```
lu12i.w     rd, si20
lu32i.d     rd, si20
lu52i.d     rd, rj, si12
```

The LU12I.W instruction performs the operation that splice the 12-bit 0 behind the lowest bit of the 20-bit immediate si20, then writes it into the general register rd after sign extension.

```
LU12I.W:
    GR[rd] = SignExtend({si20, 12'b0}, GRLEN)
```

The `LU32I.D` instruction performs the operation that splice the bit data `[31:0]` in the general register `rd` behind the lowest bit of the 20-bit immediate `si20` sign extension data; then the result is written into the general register `rd`.

```
LU32I.D:
    GR[rd] = {SignExtend(si20, 32), GR[rd][31:0]}
```

The `LU52I.D` instruction performs the operation that splice the `[51:0]` bit data in the general register `rj` behind the lowest bit of the 12-bit immediate `si12` sign extension data; then the result is written into the general register `rd`.

```
LU52I.D:
    GR[rd] = {si12, GR[rj][51:0]}
```

When the above instructions are executed, no special handling will be done on overflow.

2.2.1.5. `SLT[U]`

Instruction formats:

```
    slt     rd, rj, rk
    sltu    rd, rj, rk
```

The SLT instruction performs the operation that compares the data in the general register rj with the data in the general register rk as signed integers. If the former is smaller than the latter, the value of the general register rd is set to 1, otherwise it is set to 0.

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

The SLTU instruction performs the operation that compares the data in the general register rj with the data in the general register rk as unsigned integers. If the former is less than the latter, the value of the general 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 length compared by SLT and SLTU is consistent with the length of the general register of the executing machine.

2.2.1.6. SLT[U]I

Instruction formats:

```
slti    rd, rj, si12
sltui   rd, rj, si12
```

The SLTI instruction performs the operation that compares the data in the general register rj and the 12-bit immediate si12 sign extension data as a signed integer for size comparison. If the former is smaller than the latter, the value of the general register rd is set to 1, otherwise it is set to 0.

```
SLTI:
    tmp = SignExtend(si12, GRLEN)
    GR[rd] = (signed(GR[rj]) < signed(tmp)) ? 1 : 0
```

The SLTUI instruction performs the operation that compares the data in the general register rj and the 12-bit immediate si12 sign extension data as an unsigned integer for size comparison. If the former is smaller than the latter, the value of the general register rd is set to 1, otherwise it is set to 0.

```
SLTUI:
    tmp = SignExtend(si12, GRLEN)
    GR[rd] = (unsigned(GR[rj]) < unsigned(tmp)) ? 1 : 0
```

The data length compared by SLTI and SLTUI is consistent with the length of the general register of the executing machine. Note that for SLTUI instructions, immediate data is still sign extended.

2.2.1.7. PCADDI, PCADDU12I, PCADDU18l, PCALAU12I

Instruction formats:

```
pcaddi      rd, si20
pcaddu12i   rd, si20
pcaddu18i   rd, si20
pcalau12i   rd, si20
```

The PCADDI instruction performs the operation that splice the 2 bit 0 behind the lowest bit of the 20-bit immediate data si20 and sign extension, the resultant data plus the PC of the instruction; then the result of the addition is written into the general register rd.

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

The PCADDU12I instruction performs the operation that splice the 12-bit 0 behind the lowest bit of the 20-bit immediate data si20 and signs extension, the resultant data plus the PC of the instruction; then the result of the addition is written into the general register rd.

```
PCADDU12I:
    GR[rd] = PC + SignExtend({si20, 12'b0}, GRLEN)
```

The PCADDU18I instruction performs the operation that splice the 18-bit 0 behind the lowest bit of the 20-bit immediate si20 and signs extension, the resultant data plus the PC of the instruction; then the result of the addition is written into the general register rd.

```
PCADDU18I:
    GR[rd] = PC + SignExtend({si20, 18'b0}, GRLEN)
```

The PCALAU12I instruction performs the operation that splice the 12-bit 0 behind the lowest bit of the 20-bit immediate data si20 and sign extension; the resultant data plus the PC of the instruction; then the lowest 12 bits of the addition result are erased and written into the general register rd.

```
PCALAU12I:
    tmp = PC + SignExtend({si20, 12'b0}, GRLEN)
    GR[rd] = {tmp[GRLEN-1:12], 12'b0}
```

The data length of the above instruction operation is consistent with the length of the general register of the executed machine.

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

Instruction formats:

```
and     rd, rj, rk
or      rd, rj, rk
nor     rd, rj, rk
xor     rd, rj, rk
andn    rd, rj, rk
orn     rd, rj, rk
```

The AND instruction performs the bitwise AND operation between the data in the general register rj and the data in the general register rk; then the result is written into the general register rd.

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

The OR instruction performs the bitwise OR operation between the data in the general register rj and the data in the general register rk; then the result is written into the general register rd.

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

The NOR instruction performs the bitwise OR operation between the data in the general register rj and the data in the general register rk; then the result is written into the general register rd.

```
NOR:
    GR[rd] = ~(GR[rj] | GR[rk])
```

The XOR instruction performs the bitwise XOR operation between the data in the general register rj and the data in the general register rk; then the result is written into the general register rd.

```
XOR:
    GR[rd] = GR[rj] ^ GR[rk]
```

The ANDN instruction performs the operation that reverses the data in the general register rk bit by bit, then performs the bitwise AND operation with the data in the general register rk and the data in the general register rj; then the result is written into the general register rd.

```
ANDN:
    GR[rd] = GR[rj] & (~GR[rk])
```

The ORN instruction performs the operation that reverses the data in the general register rk bit by bit, then performs a bitwise OR operation with the data in the general register rk and the data in the general register rj, and the result is written into the general register rd.

```
ORN:
    GR[rd] = GR[rj] | (~GR[rk])
```

The data length of the above instruction operation is consistent with the length of the general register of the executed machine.

2.2.1.9. ANDI, ORI, XORI

Instruction formats:

```
andi    rd, rj, ui12
ori     rd, rj, ui12
xori    rd, rj, ui12
```

The ANDI instruction performs the bitwise AND operation between the data in the general register rj and the 12-bit immediate zero extension data; then the result is written into the general register rd.

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

The ORI instruction performs the bitwise OR operation between the data in the general register rj and the 12-bit immediate zero extension data; then the result is written into the general register rd.

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

The XORI instruction performs the bitwise XOR operation between the data in the general register rj and the 12-bit immediate zero extension data; then the result is written into the general register rd.

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

The data length of the above instruction operation is consistent with the length of the general register of the executed machine.

2.2.1.10. NOP

The NOP instruction is an alias for the instruction `andi r0, r0, 0`. Its function is only to occupy the 4-byte instruction code position and increase the PC by 4, except that it will not change any other software-visible processor state.

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

Instruction formats:

```
mul.w       rd, rj, rk
mulh.w      rd, rj, rk
mulh.wu     rd, rj, rk
mul.d       rd, rj, rk
mulh.d      rd, rj, rk
mulh.du     rd, rj, rk
```

The MUL.W instruction performs the operation that multiplies the [31:0] bit data in the general register rj with the [31:0] bit data in the general register rk, the result of the multiplication [31:0] bit data is signed and written into the general register rd.

```
MUL.W:
    product = signed(GR[rj][31:0]) * signed(GR[rk][31:0])
    GR[rd] = SignExtend(product[31:0], GRLEN)
```

The MULH.W instruction performs the operation that multiplies the [31:0] bit data in the general register rj with the [31:0] bit data in the general register rk as a signed number, the result of the multiplication [63:32] bit data is sign extension and written into the general register rd.

```
MULH.W:
    product = signed(GR[rj][31:0]) * signed(GR[rk][31:0])
    GR[rd] = SignExtend(product[63:32], GRLEN)
```

The MULH.WU instruction performs the operation that multiplies the [31:0] bit data in the general register rj with the [31:0] bit data in the general register rk as unsigned numbers, the result of the multiplication [63:32] bit data is sign extension and written into the general register rd.

```
MULH.WU:
    product = unsigned(GR[rj][31:0]) * unsigned(GR[rk][31:0])
    GR[rd] = SignExtend(product[63:32], GRLEN)
```

The MUL.D instruction performs the operation that multiplies the [63:0] bit data in the general register rj with the [63:0] bit data in the general register rk, the result of the multiplication [63:0] bit data and written into the general register rd.

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

The MULH.D instruction performs the operation that multiplies the [63:0] bit data in the general register rj with the [63:0] bit data in the general register rk as a signed number, the result of the multiplication [127:64] bit data and written into the general register rd.

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

The MULH.DU instruction performs the operation that multiplies the [63:0] bit data in the general register rj and the [63:0] bit data in the general register rk as unsigned numbers, the result of the multiplication [127:64] bit data and written into the general register rd.

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

2.2.1.12. MULW.D.W[U]

Instruction formats:

```
    mulw.d.w    rd, rj, rk
    mulw.d.wu   rd, rj, rk
```

The MULW.D.W instruction performs the operation that multiplies the [31:0] bit data in the general register rj with the [31:0] bit data in the general register rk as a signed number, and the 64-bit product result is written into the general register rd.

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

The MULW.D.WU instruction performs the operation that multiplies the [31:0] bit data in the general register rj with the [31:0] bit data in the general register rk as unsigned numbers, and writes the 64-bit product result into the general register rd.

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

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

Instruction formats:

```
div.w       rd, rj, rk
mod.w       rd, rj, rk
div.wu      rd, rj, rk
mod.wu      rd, rj, rk
div.d       rd, rj, rk
mod.d       rd, rj, rk
div.du      rd, rj, rk
mod.du      rd, rj, rk
```

The `DIV.W` and `DIV.WU` instruction performs the operation that divide the `[31:0]` bit data in the general register `rj` by the `[31:0]` bit data in the general register `rk`, and the resulting quotient is sign extension and written into the general register `rd`.

```
DIV.W:
    quotient = signed(GR[rj][31:0]) / signed(GR[rk][31:0])
    GR[rd] = SignExtend(quotient[31:0], GRLEN)

DIV.WU:
    quotient = unsigned(GR[rj][31:0]) / unsigned(GR[rk][31:0])
    GR[rd] = SignExtend(quotient[31:0], GRLEN)
```

The `MOD.W` and `MOD.WU` instruction performs the operation that divide the `[31:0]` bit data in the general register `rj` by the `[31:0]` bit data in the general register `rk`, and the resulting remainder is sign extension and written into the general register `rd`.

```
MOD.W:
    remainder = signed(GR[rj][31:0]) % signed(GR[rk][31:0])
    GR[rd] = SignExtend(remainder[31:0], GRLEN)

MOD.WU:
    remainder = unsigned(GR[rj][31:0]) % unsigned(GR[rk][31:0])
    GR[rd] = SignExtend(remainder[31:0], GRLEN)
```

The `DIV.D` and `DIV.DU` instruction performs the operation that divide the `[63:0]` bit data in the general register `rj` by the `[63:0]` bit data in the general register `rk`, and the resulting quotient sign extension and written into the general register `rd`.

```
DIV.D:
    GR[rd] = signed(GR[rj][63:0]) / signed(GR[rk][63:0])

DIV.DU:
    GR[rd] = unsigned(GR[rj][63:0]) / unsigned(GR[rk][63:0])
```

The MOD.D and MOD.DU instruction performs the operation that divide the [63:0] bit data in the general register rj by the [63:0] bit data in the general register rk, and the resulting remainder is sign extension and written into the general register rd.

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

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

When DIV.W, MOD.W, DIV.D and MOD.D perform division operations, the operands are all regarded as signed numbers. When DIV.WU, M0D.WU, DIV.DU and MOD.DU perform division operations, the source operands are all regarded as unsigned numbers.

Each pair of instructions for finding the quotient/remainder satisfies the result of DIV.W/MOD.W, DIV.WU /MOD.WU, DIV.D/MOD.D, DIV.DU/MOD.DU, the remainder and the dividend The sign is 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 exception will be triggered.

2.2.2. Bit-shift Instructions

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

Instruction formats:

```
sll.w       rd, rj, rk
srl.w       rd, rj, rk
sra.w       rd, rj, rk
rotr.w      rd, ri, rk
```

The SLL.W instruction performs the operation that logical left shifts the bit data of [31:0] in the general register rj, and writes the sign extension of the shift result into the general register rd.

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

The SRL.W instruction performs the operation that logical right shifts the bit data of [31:0] in the general register rj, and writes the sign extension of the shift result into the general register rd.

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

The SRA.W instruction performs the operation that arithmetical right shifts [31:0] bit data in the general register rj, and writes the sign extension of the shift result into the general register rd.

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

The ROTR.W instruction performs the operation that cyclical right shifts the [31:0] bit data in the general register rj, and writes the sign extension of the shift result into the general register rd.

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

The shift amount of the above-mentioned shift instruction is all [4:0] bit data in the general register rk, and is regarded as an unsigned number.

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

Instruction formats:

```
sliw        rd, rj, ui5
srli.w      rd, rj, ui5
srai.w      rd, rj, ui5
rotri.w     rd, rj, ui5
```

The SLLI.W instruction performs the operation that logical left shifts the [31:0] bit data in the general register rj, and writes the sign extension of the shift result into the general register rd.

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

The SRLI.W instruction performs the operation that logical right shifts the [31:0] bit data in the general register rj to the right, and writes the sign extension of the shift result into the general register rd.

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

The SRAI.W instruction performs the operation that arithmetical right shifts the bit data of [31:0] in the general register rj, and writes the sign extension of the shift result into the general register rd.

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

The ROTRI.W instruction performs the operation that cyclical right shifts the [31:0] bit data in the general register rj, and the sign extension of the shift result is written into the general register rd.

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

The shift amounts of the above shift instructions are all 5-bit unsigned immediate ui5 in the instruction code.

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

Instruction formats:

```
sl.d        rd, rj, rk
srl.d       rd, rj, rk
sra.d       rd, rj, rk
rotr.d      rd, rj, rk
```

The SLL.D instruction performs the operation that logical left shifts the bit data of [63:0] in the general register rj, and writes the sign extension of the shift result into the general register rd.

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

The SRL.D instruction performs the operation that logical right shifts the bit data of [63:0] in the general register rj, and writes the sign extension of the shift result into the general register rd.

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

The SRA.D instruction performs the operation that arithmetic right shifts the bit data of [63:0] in the general register rj, and writes the sign extension of the shift result into the general register rd.

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

The ROTR.D instruction performs the operation that cyclical right shifts the bit data of [63:0] in the general register rj, and writes the sign extension of the shift result into the general register rd.

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

The shift amount of the above-mentioned shift instruction is all [5:0] bit data in the general register rk, and is regarded as an unsigned number.

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

Instruction formats:

```
slli.d      rd, rj, ui6
srli.d      rd, rj, ui6
srai.d      rd, rj, ui6
rotri.d     rd, rj, ui6
```

The SLII.D instruction performs the operation that logicalleft shifts the bit data of [63:0] in the general register rj, and the sign extension of the shift result is written into the general register rd.

```
SLLI.D:
    GR[rd] = SLL(GR[rj][63:0], ui6)
```

The SRLI.D instruction performs the operation that logical right shifts the bit data of [63:0] in the general register rj, and writes the sign extension of the shift result into the general register rd.

```
SRLI.D:
    GR[rd] = SRL(GR[rj][63:0], ui6)
```

The SRAI.D instruction performs the operation that arithmetically right shifts the bit data of [63:0] in the general register rj, and writes the sign extension of the shift result into the general register rd.

```
SRAI.D:
    GR[rd] = SRA(GR[rj][63:0], ui6)
```

The ROTRI.D instruction performs the operation that cyclical right shifts the [63:0] bit data in the general register rj, and the sign extension of the shift result is written into the general register rd.

```
ROTRI.D:
    GR[rd] = ROTR(GR[rj][63:0], ui6)
```

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

2.2.3. Bit-manipulation Instructions

2.2.3.1. EXT.W{B/H}

Instruction formats:

```
ext.w.b     rd, rj
ext.w.h     rd, rj
```

The EXT.W.B instruction performs the operation that will sign extension the bit data of [7:0] in the general register rj and write it into the general register rd.

```
EXT.W.B:
    GR[rd] = SignExtend(GR[rj][7:0], GRLEN)
```

The EXT.W.H instruction performs the operation that will sign extension the bit data of [15:0] in the general register rj and write it into the general register rd.

```
EXT.W.H:
    GR[rd] = SignExtend(GR[rj][15:0], GRLEN)
```

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

Instruction formats:

```
clo.w       rd, rj
clo.d       rd, rj
clz.w       rd, rj
clz.d       rd, rj
cto.w       rd, rj
cto.d       rd, rj
ctz.w       rd, rj
ctz.d       rd, rj
```

The CLO.W instruction performs the operation that for the data of bit [31:0] in the general register rj, the number of continuous bits 1 is measured from bit 31 to bit 0, and the result is written into the universal register rd.

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

The CLZ.W instruction performs the operation that for the data of bit [31:0] in the general register rj, the number of continuous bits 0 is measured from bit 31 to bit 0, and the result is written into the universal register rd.

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

The CTO.W instruction performs the operation that for the data of bit [31:0] in the general register rj, the number of continuous bits 1 is measured from bit 0 to bit 31, and the result is written into the universal register rd.

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

The CTZ.W instruction performs the operation that for the data of bit [31:0] in the general register rj, the number of continuous bits 0 is measured from bit 0 to bit 31, and the result is written into the universal register rd.

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

The CLO.D instruction performs the operation that for the data of bit [63:0] in the general register rj, the number of continuous bits 1 is measured from bit 63 to bit 0, and the result is written into the universal register rd.

```
CLO.D:
    GR[rd] = CL0(GR[rj][63:0])
```

The CLZ.D instruction performs the operation that for the data of bit [63:0] in the general register rj, the number of continuous bits 1 is measured from bit 0 to bit 63, and the result is written into the universal register rd.

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

The CTO.D instruction performs the operation that for the data of bit [63:0] in the general register rj, the number of continuous bits 0 is measured from bit 0 to bit 63, and the result is written into the universal register rd.

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

The `CTZ.D` instruction performs the operation that for the data of bit `[63:0]` in the general register `rj`, the number of continuous bits `0` is measured from bit `0` to bit `63`, and the result is written into the universal register `rd`.

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

2.2.3.3. `BYTEPICK.{W/D}`

Instruction formats:

```
bytepick.w  rd, rj, rk, sa2
bytepick.d  rd, rj, rk, sa3
```

The `BYTEPICK.W` instruction performs the operation that splice `[31:0]` bits in the general register `rj` behind `[31:0]` bits in the general register `rk`, and intercepts 4 consecutive bytes starting from the leftmost `sa2` byte, and writes the 32-bit bit string symbol into universal register `rd` after expansion.

```
BYTEPICK.W:
    tmp = {GR[rk][8*(4-sa2):0], GR[rj][31:8*(4-sa2)]}
    GR[rd] = SignExtend(tmp[31:0], GRLEN)
```

The `BYTEPICK.D` instruction performs the operation that splice `[63:0]` bits in the general register rj behind `[63:0]` bits in the general register `rk`, and intercepts 8 consecutive bytes starting from the leftmost `sa3` byte, and writes the 64-bit bit string symbol into universal register `rd` after expansion.

```
BYTEPICK.D:
    GR[rd] = {GR[rk][8*(8-sa3):0], GR[rj][63:8*(8-sa3)]}
```

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

Instruction formats:

```
revb.2h     rd, rj
revb.4h     rd, ri
revb.2w     rd, rj
revb.d      rd, rj
```

The `REVB.2H` instruction performs the operation that arranges the 2 bytes in the `[15:0]` bits in the general register `rj` in reverse order to form the `[15:0]` bits of the intermediate result, and reverses the 2 bytes in the `[31:16]` in the general register `rj` Arrange the `[31:16]` bits of the intermediate result, and write the 32-bit intermediate result sign extended to the general register `rd`.

```
REVB.2H:
    tmp0 = {GR[rj][ 7: 0], GR[rj][15: 8]}
    tmp1 = {GR[rj][23:16], GR[rj][31:24]}
    GR[rd] = SignExtend({tmp1, tmp0}, GRLEN)
```

The `REVB.4H` instruction performs the operation that arranges the 2 bytes in the `[15:0]` bits of the general register `rj` in reverse order and writes them into the `[15:0]` bits of the general register `rd`, and writes 2 words in the `[31:16]` bits of the general register `rj`. Write the sections in reverse order to bits `[31:16]` of the general register `rd`, and write the 2 bytes of bits `[47:32]` in the general register `rj` in reverse order to bits `[47:32]` of the general register rd. The 2 bytes in the `[63:48]` bits in the register `rj` are written in the `[63:48]` bits in the general register rd in reverse order.

```
REVB.4H:
    tmp0 = {GR[rj][ 7: 0], GR[rj][15: 8]}
    tmp1 = {GR[rj][23:16], GR[rj][31:24]}
    tmp2 = {GR[rj][39:32], GR[rj][47:40]}
    tmp3 = {GR[rj][55:48], GR[rj][63:56]}
    GR[rd] = {tmp3, tmp2, tmp1, tmp0}
```

The `REVB.2W` instruction performs the operation that writes the 4 bytes in the `[31:0]` bits of the general register `rj` into the `[31:0]` bits of the general register `rd` in reverse order, and writes 4 of the `[63:32]` bits in the general register `rj`. Write the byte in reverse order to bits `[63:32]` of the general register `rd`.

```
REVB.2W:
    tmp0 = {GR[rj][ 7: 0], GR[rj][15: 8], GR[rj][31:24], GR[rj][23:16]}
    tmp1 = {GR[rj][39:32], GR[rj][47:40], GR[rj][55:48], GR[rj][63:56]}
    GR[rd] = {tmp1, tmp0}
```

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

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

2.2.3.5. `REVH.{2W/D}`

Instruction formats:

```
revh.2w     rd, rj
revh.d      rd, rj
```

The `REVH.2W` instruction performs the operation that writes two half-words in bit `[31:0]` of general purpose register `rj` into bit `[31:0]` of general purpose register `rd`, and two half-words in bit `[63:32]` of general purpose register `rj` into bit `[63:32]` of general purpose register `rd`.

```
REVH.2W:
    tmp0 = {GR[rj][15: 0], GR[rj][31:16]}
    tmp1 = {GR[rj][47:32], GR[rj][63:48]}
    GR[rd] = {tmp1, tmp0}
```

The `REVH.D` instruction performs the operation that write four half-words in `[63:0]` bit of universal register `rj` in reverse order to universal register `rd`.

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

2.2.3.6. `BITREV.{4B/8B}`

Instruction formats:

```
bitrev.4b   rd, rj
bitrev.8b   rd, rj
```

The BITREV.4B instruction performs the operation that the `[7:0]` bit in general register `rj` is arranged in reverse order, the `[15:8]` bit in general register `rj` is arranged in reverse order, the `[23:16]` bit in general register `rj` is arranged in reverse order, and the `[31:24]` bit in general register `rj` is arranged in reverse order; the 32-bit intermediate result sign extension is written into general register `rd` in turn.

```
BITREV.4B:
    bstr32[31:24] = BITREV(GR[rj][31:24])
    bstr32[23:16] = BITREV(GR[rj][23:16])
    bstr32[15: 8] = BITREV(GR[rj][15: 8])
    bstr32[ 7: 0] = BITREV(GR[rj][ 7: 0])
    GR[rd] = SignExtend(bstr32, GRLEN)
```

The `BITREV.8B` instruction performs the operation that the `[7:0]` bit in general register `rj` is arranged in reverse order, the `[15:8]` bit in general register `rj` is arranged in reverse order, the `[23:16]` bit in general register `rj` is arranged in reverse order, the `[31:24]` bit in general register `rj` is arranged in reverse order; the `[39:32]` bit in general register `rj` is arranged in reverse order; the `[47:40]` bit in general register `rj` is arranged in reverse order; the `[55:48]` bit in general register `rj` is arranged in reverse order; the `[63:56]` bit in general register `rj` is arranged in reverse order; the 32-bit intermediate result sign extension is written into general register `rd` in turn.

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

2.2.3.7. `BITREV.{W/D}`

Instruction formats:

```
bitrev.w        rd, rj
bitrev.d        rd, rj
```

The `BITREV.W` instruction performs the operation that the `[31:0]` bit in general register `rj` is arranged in reverse order; the 32-bit intermediate result sign extension is written into general register `rd` in turn.

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

The `BITREV.D` instruction performs the operation that the `[63:0]` bit in general register `rj` is arranged in reverse order; the 32-bit intermediate result sign extension is written into general register `rd` in turn.

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

2.2.3.8. `BSTRINS.{W/D}`

Instruction formats:

```
bstrins.w       rd, rj, msbw, lsbw
bstrins.d       rd, rj, msbd, lsbd
```

The `BSTRINS.W` instruction performs the operation that replaces the `[msbw:lsbw]` bit in the lowest 32 bits of the general register `rd` with the `[msbw-lsbw:0]` bit in the general register `rj`, and the resulting 32-bit result is sign extension and written into the general register `rd`.

```
BSTRINS.W:
    bstr32[31:msbw+1] = GR[rd][31: msbw+1]
    bstr32[msbw:lsbw] = GR[rj][msbw-lsbw:0]
    bstr32[lsbw-1:0] = GR[rd][lsbw-1:0]
    GR[rd] = SignExtend(bstr32[31:0], GRLEN)
```

The `BSTRINS.D` instruction performs the operation that replaces the `[msbd:lsbd]` bit in the general register `rd` with the `[msbd-lsbd:0]` bit in the general register `rj`, and the rest of the general register `rd` remains unchanged.

```
BSTRINS.D:
    GR[rd][63:msbd+1] = GR[rd][63:msbd+1]
    GR[rd][msbd:lsbd] = GR[rj][msbd-lsbd:0]
    GR[rd][lsbd-1:0] = GR[rd][lsbd-1:0]
```

2.2.3.9. `BSTRPICK.{W/D}`

Instruction formats:

```
bstrpick.w  rd, rj, msbw, lsbw
bstrpick.d  rd, rj, msbd, lsbd
```

`BSTRPICK.W` extracts the `[msbw:Isbw]` bit in the general register `rj` and zero-extends it to 32 bits, and the formed 32-bit intermediate result is sign extension and written into the general register `rd`.

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

`BSTRPICK.D` extracts the `[msbd:Isbd]` bit in the general register `rj` and zero-extends it to 64 bits and writes it into the general register `rd`.

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

2.2.3.10. `MASKEQZ`, `MASKNEZ`

Instruction formats:

```
maskeqz     rd, rj, rk
masknez     rd, rj, rk
```

MASKEQZ and MASKNEZ instructions perform conditional assignment operations. When MASKEQZ is executed, if the value of the general register rk is equal to 0, the general register rd is set to 0, otherwise it is assigned to the value of the rj register.

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

When MASKNEZ is executed, if the value of the general register rk is not equal to 0, the general register rd is set to 0, otherwise it is assigned to the value of the rj register.

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

2.2.4. Branch Instructions

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

Instruction formats:

```
beq     rj, rd, offs16
bne     rj, rd, offs16
blt     rj, rd, offs16
bge     rj, rd, offs16
bltu    rj, rd, offs16
bgeu    rj, rd, offs16
```

The BEQ instruction performs the operation that compares the values of general register rj and general register rd, if the two are equal, jump to the target address, otherwise it does not jump.

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

The BNE instruction performs the operation that compares the values of general register rj and general register rd, if the two are not equal, jump to the target address, otherwise it does not jump.

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

The BLT instruction performs the operation that compares the values of general register rj and general register rd as signed numbers. If the former is smaller than the latter, it jumps to the target address, otherwise it does not jump.

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

The BGE instruction performs the operation that compares the values of general register rj and general register rd as signed numbers. If the former is greater than or equal to the latter, it jumps to the target address, otherwise it does not jump.

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

The BLTU instruction performs the operation that compares the values of general register rj and general register rd as unsigned numbers. If the former is less than the latter, it jumps to the target address, otherwise it does not jump.

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

The BGEU instruction performs the operation that compares the values of general register rj and general register rd as unsigned numbers. If the former is greater than or equal to the latter, it jumps to the target address, otherwise it does not jump.

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

The calculation method of the jump target address of the above-mentioned six branch instructions is to logically shift the 16-bit immediate offs16 in the instruction code by 2 bits and then sign expand, and the resulting offset value is added to the PC of the branch instruction.

TIP When writing assembly, you need to fill in the immediate field with the **real offset value** in bytes, i.e. (offs16<<2).

2.2.4.2. BEQZ, BNEZ

Instruction formats:

```
beqz        rj, offs21
bnez        rj, offs21
```

The BEQZ instruction performs the operation that judges the value of the general register rj, if it is equal to 0, jump to the target address, otherwise it does not jump.

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

The BNEZ instruction performs the operation that judges the value of the general register rj, if it is not equal to 0, it jumps to the target address, otherwise it does not jump.

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

The jump target address of the above two branch instructions is to logical left shift the 21-bit immediate offs21 in the instruction code by 2 bits and then sign extension, and the resulting offset value is added to the PC of the branch instruction.

TIP When writing assembly, you need to fill in the immediate field with the **real offset value** in bytes, i.e. (offs21<<2).

2.2.4.3. B

Instruction formats:

```
b       offs26
```

The B instruction performs the operation that jumps to the target address unconditionally. The jump target address is to logical left shift the 26-bit immediate offs26 in the instruction code by 2 bits and then sign extension, and the resulting offset value is added to the PC of the branch instruction.

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

TIP When writing assembly, you need to fill in the immediate field with the **real offset value** in bytes, i.e. (offs26<<2).

2.2.4.4. BL

Instruction formats:

```
bl      offs26
```

The BL instruction performs the operation that jumps to the target address unconditionally, and writes the result of adding 4 to the PC value of the instruction into the No.1 general register r1.

The jump target address of the instruction is to shift the 26-bit immediate offs26 in the instruction code to the left by 2 bits and then sign extend it. The shift value is added to the PC of the branch instruction.

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

In LA ABI, the No.1 general register r1 serves as the return address register ra.

TIP When writing assembly, you need to fill in the immediate field with the **real offset value** in bytes, i.e. (offs26<<2).

2.2.4.5. JIRL

Instruction formats:

```
jirl        rd, rj, offs16
```

JIRL jumps to the target address unconditionally, and the PC value of the instruction plus 4; then writes the result into the general register rd.

The jump target address of the instruction is to logically shift the 16-bit immediate offs16 in the instruction code by 2 bits to the left and then sign extension, and the resulting offset value is added to the value in the general register rj.

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

When rd is equal to 0, the function of JIRL is a common non-call indirect jump instruction.

JIRL with rd equal to 0, rj equal to 1 and offs16 equal to 0 is often used as an indirect jump from call return.

TIP When writing assembly, you need to fill in the immediate field with the **real offset value** in bytes, i.e. (offs16<<2).

2.2.5. Common Memory Access Instructions

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

Instruction formats:

```
ld.b        rd, rj, si12
ld.h        rd, rj, si12
ld.w        rd, rj, si12
ld.d        rd, rj, si12
ld.bu       rd, rj, si12
ld.hu       rd, rj, si12
ld.wu       rd, rj, si12
st.b        rd, rj, si12
st.h        rd, rj, si12
st.w        rd, rj, si12
st.d        rd, rj, si12
```

`LD.{B/H/W/D}` retrieves the data of one byte/halfword/word/double word from the internal sign extension and writes it into the general register `rd`.

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

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

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

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

`LD.{BU/HU/WU}` retrieves one byte/halfword/word data from the memory and writes it into the general register `rd` after zero extension.

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

LD.HU:
    vaddr = GR[rj] + SignExtend(si12, GRLEN)
    AddressCompli anceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    halfword = MemoryLoad(paddr, HALFWORD)
    GR[rd] = ZeroExtend(halfword, GRLEN)

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

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

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

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

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

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

The memory access address calculation method of the above instruction is sum the value in the general register `rj` and the sign extension 12-bit immediate value `sil2`.

For `LD.{H[U]/W[U]/D}` and `ST.{B/H/W/D}` instructions, no matter what kind of hardware implementation and environmental configuration, as long as their memory access addresses are naturally aligned When the memory access address is not naturally aligned, if the hardware implementation supports non-aligned memory access and the current computing environment is configured to allow non-aligned memory access, then the non-aligned exception will not be triggered, otherwise a non-aligned exception will be triggered.

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

Instruction formats:

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

`LDX.{B/H/W/D}` retrieves the data of one byte/halfword/word/double word from the internal sign extension and writes it into the general register `rd`.

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

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

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

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

`LDX.{BU/HU/WU}` retrieves one byte/halfword/word data from the internal zero extension and writes it into the general register `rd`.

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

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

LDX.WU:
    vaddr = GR[rj] + GR[rk]
    AddressCompli anceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    word = MemoryLoad(paddr, WORD)
    GR[rd] = ZeroExtend(word, GRLEN)
```

`STX.{B/H/W/D}` writes `[7:0]`, `[15:0]`, `[31:0]` and `[63:0]` bits of data in the general register `rd` into the memory.

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

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

STX.W:
    vaddr = GR[rj] + GR[rk]
    AddressCompli anceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    MemoryStore(GR[rd][31:0], paddr, WORD)

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

The memory access address calculation method of the above instruction is the value in the general register `rj` and the value in the general register `rk`. For `LDX.{H[U]/W[U]/D}` and `STX.{B/H/W/D}` instructions, no matter what kind of hardware implementation and environment configuration, as long as its memory access address is natural Aligned, will not trigger non-aligned exception; when the fetch address is not naturally aligned, if the hardware implementation supports non-aligned memory access and the current computing environment is configured to allow non-aligned memory access, then the non-aligned exception will not be triggered, otherwise a non-aligned exception will be triggered.

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

Instruction formats:

```
ldptr.w     rd, rj, si14
ldptr.d     rd, rj, si14
stptr.w     rd, rj, si14
stptr.d     rd, rj, si14
```

`LDPTR.{W/D}` retrieves the data of a word/double word from the internal sign extension and writes it into the general register `rd`.

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

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

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

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

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

The memory access address calculation method of the above instruction is to logical left shift the 14-bit immediate data `si14` by 2 bits, sign extension, and then sum the value in the general register `rj`.

TIP When writing assembly, you need to fill in the immediate field with the **real offset value** in bytes, i.e. `(si14<<2)`.

For `LDPTR.{W/D}` and `STPTR.{W/D}` instructions, no matter what kind of hardware implementation and environmental configuration, as long as the memory access address is naturally aligned, the non-aligned exception will not be triggered; when the memory address is not naturally aligned, if the hardware implementation supports unaligned memory access and the current computing environment is configured to allow unaligned memory access, then the unaligned exception will not be triggered, otherwise it will trigger the unaligned exception.

`LDPTR.{W/D}`, `STPTR.{W/D}` instructions are used in conjunction with `ADDU16I.D` instructions to accelerate GOT table-based access in position-independent codes.

2.2.5.4. PRELD

Instruction formats:

```
preld       hint, rj, si12
```

PRELD Reads a cache-line of data from memory in advance into the Cache. The access address is the 12bit immediate number of the value in the general register rj plus the symbol extension.

The processor learns from the hint in the PRELD instruction what type will be acquired and which level of Cache the data to be taken back fill in, hint has 32 optional values (0 to 31), 0 represents load to level 1 Cache, and 8 represents store to level 1 Cache. The remaining hint values are not defined and are processed for nop instructions when the processor executes.

If the Cache attribute of the access address of the PRELD instruction is not cached, then the instruction cannot generate a memory access action and is treated as a NOP instruction. The PRELD instruction will not trigger any exceptions related to MMU or address.

2.2.5.5. PRELDX

Instruction formats:

```
preldx      hint, rj, rk
```

The PRELDX instruction continuously prefetches data from memory into the Cache according to the configuration parameters, and the continuously prefetched data is a block (block) of length block_size starting from the specified base address (base) with a number of (block_num) spacing stride. The base address is the sum of the [63:0] bits in the general register rj and the sign extension [15:0] bits in the general register rk. The [I16] bits in general register rk are the address sequence ascending and descending flag bits, with 0 indicating address ascending and 1 indicating address descending. The value of bits [25:20] in general register rk is block_size, the basic unit of block_size is 16 bytes, so the maximum length of a single block is 1KB. The value of bits [39:32] in general register rk is block_num-1, so a single instruction can prefetch up to 256 blocks. The value of bits [59:44] in the block general register rk is treated as a signed number and defines the stride between adjacent blocks, the basic unit of stride is 1 byte. The value of bits [39:32] in rk is block.num-1, so a single instruction can prefetch up to 256 blocks. The value of bits [59:44] in general register rk is regarded as a signed number, which defines the corresponding The basic unit of stride and stride between adjacent blocks is 1 byte.

hint in the PRELDX instruction indicates the type of prefetch and the level of Cache into which the fetched data is to be filled. hint has 32 selectable values from 0 to 31. Currently, hint=0 is defined as load prefetch to level 1 data Cache, hint=2 is defined as load prefetch to level 3 Cache, hint-8 is defined as store prefetch to level 1 data Cache. The meaning of the rest of hint values is not defined yet, and the processor executes it as NOP instruction.

If the Cache attribute of the access address of the PRELDX instruction is not cached, then the instruction cannot generate a memory access action and is treated as a NOP instruction.

The PRELDX instruction does not trigger any exceptions related to MMU or address.

2.2.6. Bound Check Memory Access Instructions

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

Instruction formats:

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

`LDGT/LDLE.B/H/W/D` fetches a byte/half word word/double word data symbol extension from memory and writes it to the general register `rd`.

`STGT/STLE.B/H/W/D` writes the `[7:0]`/`[15:0]`/`[31:0]`/`[63:0]` bits of data from the general register `rd` to memory.

The access addresses of the above instructions come directly from the values in the general register `rj`. The access addresses of the above instructions are required to be naturally aligned, otherwise a non-alignment exception will be triggered.

`B/H/W/D` and `STGT.B/H/W/D` instructions check whether the value in general register `rj` is greater than the value in general register `rk`, and terminate the access operation and trigger the bound check exception if the condition is not satisfied; `B/H/W/D` and `STLE.B/H/W/D` instructions check whether the value in general register `rj` is less than or equal to the value in general register `rk`, and if the condition is not satisfied, the access operation is terminated and the bound check exception is triggered.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.7. Atomic Memory Access Instructions

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

Instruction formats:

```
amswap.w        rd, rk, rj
amswap_db.w     rd, rk, rj
amswap.d        rd, rk, rj
amswap_db.d     rd, rk, rj
amadd.w         rd, rk, rj
amadd_db.w      rd, rk, rj
amadd.d         rd, rk, rj
amadd_db.d      rd, rk, rj
amand.w         rd, rk, rj
amand_db.w      rd, rk, rj
amand.d         rd, rk, rj
amand_db.d      rd, rk, rj
amor.w          rd, rk, rj
amor_db.w       rd, rk, rj
amor.d          rd, rk, rj
amor_db.d       rd, rk, rj
amxor.w         rd, rk, rj
amxor_db.w      rd, rk, rj
amxor.d         rd, rk, rj
amxor_db.d      rd, rk, rj
ammax.w         rd, rk, rj
ammax_db.w      rd, rk, rj
ammax.d         rd, rk, rj
ammax_db.d      rd, rk, rj
ammin.w         rd, rk, rj
ammin_db.w      rd, rk, rj
ammin.d         rd, rk, rj
ammin_db.d      rd, rk, rj
ammax.wu        rd, rk, rj
```

```
ammax_db.wu     rd, rk, rj
ammax.du        rd, rk, rj
ammax_db.du     rd, rk, rj
ammin.wu        rd, rk, rj
ammin_db.wu     rd, rk, rj
ammin.du        rd, rk, rj
ammin_db.du     rd, rk, rj
```

The `AM*` atomic access instruction performs a sequence of “read-modify-write” operations on a memory cell atomically. Specifically, it retrieves the old value at the specified address in memory and writes it to the general register `rd`, performs some simple operations on the old value in memory and the value in the general register `rk`, and then writes the result of the operations back to the specified address in memory. The entire “read-modify-write” process is atomic, meaning that the processor executing the instruction does not perform any other access-write operations nor does it trigger any exceptions during the time between the return of the access read operation data and the global visibility of the access write operation, and no other processor cores or cache-consistent. The module has global visibility of the execution of the write operation on the Cache row where the instruction accesses the object.

The access address of an `AM*` atomic access instruction is the value of the general register `rj`. The access address of an `AM*` atomic access instruction always requires natural alignment, and failure to meet this condition will trigger a non-alignment exception.

Atomic access instructions ending in `.W` and `.WU` read and write memory and intermediate operations with a data length of 32 bits, while atomic access instructions ending in `.D` and `.DU` read and write memory and intermediate operations with a data length of 64 bits. Whether ending in `.W` or `.WU`, the data of a word retrieved from memory by an atomic access instruction is symbolically extended and written to the general register `rd`.

`AMSWAP[.DB].{W/D}` instruction writes the new value of memory from the general register `rk`. `AMADD[.DB].{W/D}` instruction writes the new value of memory from the result ofold value of memory plus the value in general register `rk`. `AMAND[DB].{W/D}` instruction writes the new value to memory as a result of the bitwise AND operation of the old value in memory and the value in general register `rk`. `AMOR[DB].{W/D}` instruction writes a new value to memory from `AMXOR[.DB]`. The new value written to memory by the `{W/D}` instruction is the result of the bitwise OR operation of the old value in memory and the value in general register `rk`. `AMMAX[_DB].{W/D}` instruction writes the new value to memory as the result of the bitwise AND operation of the old value in memory and the value in general register `rk`. The new value written to memory is the maximum value obtained by comparing the old value in memory with the value in general register `rk` as a signed number. `[_DB].{W/D}` instruction The new value written to memory is the minimum value obtained by comparing the old value of memory with the value in general register `rk` as if it were a signed number. The new value written to memory by the `AMMAX[DB].[WU/DU]` instruction is the maximum value obtained by comparing the old value in memory with the value in general register rk as an unsigned number. `AMMIN[_DB].{WU/DU}` instruction writes the new value to memory by comparing the old value in memory with the value in general register rk as an unsigned number. The new value written to memory is the minimum value obtained by comparing the old value in memory with the value in general register rk as an unsigned number.

`AM*_DB.W[U]/D[U]` instruction not only completes the above atomized operation sequence, but also implements the data barrier function at the same time. That is, all access operations preceding the atomic access instruction in the same processor core are completed before such atomic access instructions are allowed to be executed, and all access operations following the atomic access instruction in the same processor core are allowed to be executed only after such atomic access instructions are executed.

If the AM* atomic memory access instruction has the same register number as rd and rj, the execution will trigger an Instruction Non-defined Exception.

If the AM* atomic memory access instruction has the same register number as rd and rk, the execution result is uncertain. Please software to avoid this situation.

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

Instruction formats:

```
amswap.b        rd, rk, rj
amswap_db.b     rd, rk, rj
amswap.h        rd, rk, rj
amswap_db.h     rd, rk, rj
amadd.b         rd, rk, rj
amadd_db.b      rd, rk, rj
amadd.h         rd, rk, rj
amadd_db.h      rd, rk, rj
```

AM{SWAP/ADD}[_DB].{B/H} and AM{SWAP/ADD}[_DB].{W/D} are atomic access instructions, can atomically complete the "read - modify - write" sequence of operations on a memory cell, the main difference is that the data being accessed is byte/half-word or word/double-word.

AM{SWAP/ADD}[_DB].{B/H} retrieve the old byte/half word value at the specified address in memory and write it to the general register rd after symbol extension, At the same time, the old value in the memory is exchanged or added with the byte/half-word value of the general register rk [7:0]/[15:0] bit, and then the byte/half-word results will be written back to the specified address of the memory. The entire "read-modify-write" process is atomic, meaning that the execution of the instruction, from the access to read the data return to the access to write the implementation of the effect of global visibility at the time, the processor executing the instruction neither executes other memory access write operations nor triggers any exception, and no other processor core or Cache coherence module can globally see the execution effect of the write operation on the Cache line of the object accessed by the instruction.

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

AM{SWAP/ADD}[_DB].H access address of an atomic access instruction is always required to be naturally aligned, and a non-alignment exception is triggered if this condition is not met.

In addition to the above atomic sequence of operations, the AM{SWAP/ADD}_DB.{B/H} instruction also implements the data barrier function. That is, when this kind of atomic access instruction is allowed to execute before, all in the same processor core before the atomic access instruction access operations have been completed; at the same time, only until the completion of this kind of atomic access instruction execution, all in the same processor core after the atomic access instruction access operation is allowed to execute.

If rd and rj have the same register number in AM{SWAP/ADD}[_DB].{B/H} instruction, there is no exception for trigger instruction.

If the register numbers of rd and rk in an AM{SWAP/ADD}[_DB].{B/H} instruction are the same, the execution result is uncertain, so please ask the software to avoid this situation.

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

Instruction formats:

```
amcas.b         rd, rk, rj
amcas_db.b      rd, rk, rj
amcas.h         rd, rk, rj
amcas_db.h      rd, rk, rj
amcas.w         rd, rk, rj
amcas_db.w      rd, rk, rj
amcas.d         rd, rk, rj
amcas_db.d      rd, rk, rj
```

`AMCAS[_DB].{B/H/W/D}` instruction performs a byte/half-word/word/double-word sized Compare-and-Swap operation on a specified address in memory: The byte/half-word/word/double-word value retrieved from memory (old memory value) is compared with the value stored in the [7:0]/`[15:0]/[31:0]/[63:0]` location of the general-purpose register `rd` (expected value), and the value stored in the `[7:0]/[15:0]/[31:0]/[63:0]` location of the general-purpose register `rk` (new value) is written to the same location in the memory only when the comparison results are equal. Regardless of whether the comparison results are equal or not, the old memory value is written to the general-purpose register `rd` after sign expansion.

The above process, If a write occurs because the old memory value is equal to the expected value, then the entire "read - modify - write" process is atomic, that is, from the access to the read operation data return to the access to the write operation to perform the effect of the global visibility of this time, the processor executing the instruction is neither the implementation of the other access to the write operation nor trigger Any exception, and no other processor core or Cache Consistency Module to the instruction access object where the Cache line of the write operation of the execution of the effect of the global visible.

`AMCAS[_DB].{H/W/D}` The access address of the instruction is the value of general-purpose register `rj`, and the access address is always required to be naturally aligned, if this condition is not met, a non-aligned exception will be triggered.

In addition to the above atomic sequence of operations, the `AMCAS_DB.{B/H/W/D}` instruction also implements the data barrier function. That is, when this kind of atomic access instruction is allowed to execute before, all in the same processor core before the atomic access instruction access operations have been completed; at the same time, only when this kind of atomic access instruction execution is completed, all in the same processor core after the atomic access instruction access operations are allowed to execute.

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

Instruction formats:

```
ll.w        rd, rj, si14
ll.d        rd, rj, si14
sc.w        rd, rj, si14
sc.d        rd, rj, si14
```

The two pairs of instructions, `LL.W` and `SC.W`, `LL.D` and `SC.D`, are used to implement an atomic “read, modify, and write” sequence of memory access operations. The LL.{W/D} instruction retrieves a word/double-word data from the specified address of the memory and writes it to the general register rd after sign extension, and the paired SC. {W/D} instruction operates on the same length of data and has the same access Memory address. The atomic maintenance mechanism for the sequence of memory access operations is that when LL.{W/D} is executed, the access address is recorded and the previous flag is set (LLbit is set to 1), and the LLbit is checked when the SC.{W/D} instruction is executed. Only when the LLbit is 1, the write action will actually occur, otherwise it will not be written. When the software needs to successfully complete an atomic “read-modify-write” memory access operation sequence, it needs to construct a loop to repeatedly execute the LLSC instruction pair until the SC is successfully completed. In order to construct this loop, the SC.[W/D] instruction will write the flag of its execution success (or simply the LLbit value seen when the SC instruction is executed) into the general register rd and return.

During the execution of the paired LLSC, the following events will clear the LLbit to 0:

- The ERTN instruction is executed and the KL0 bit in CSR.LLBCTL is not equal to 1 when executed;
- Other processor cores or Cache Coherent I/O masters perform a store operation on the Cache line where the address corresponding to the LLbit is located.

If the memory access attribute of the LLSC instruction to the access address is not Cached, then the execution result is uncertain.

2.2.7.5. SC.Q

Instruction formats:

```
SC.Q            rd, rk, rj
```

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

SC.Q writes the 128-bit data {GR[rk][63:0], GR[rd][63:0]} obtained by splicing the general-purpose registers rk and rd into memory, and its access address is the value of the general-purpose register rj. SC.Q instruction will check LLbit when executing, and only when LLbit is 1, then it will write, otherwise it will not write, SC.Q instruction will write the flag of success or failure (also can be understood as the value of LLbit when SC.Q instruction executes) into general register rd and return to the memory.

The access address of SC.Q instruction is always required to be 16-byte aligned, if this condition is not met, a non-aligned exception will be triggered.

If the SC.Q instruction’s memory access attribute for the access address is not consistently cacheable (CC), the result of the execution is indeterminate.

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

Instruction formats:

```
ll.acq.w        rd, rj
ll.acq.d        rd, rj
sc.rel.w        rd, rj
sc.rel.d        rd, rj
```

`LL.ACQ.{W/D}` is an LL.{W/D} instruction with read-acquire semantics, that is, only when `LL.ACQ.{W/D}` is executed (globally visible), all subsequent access operations can start executing (globally visible effect); `SC.REL.{W/D}` is an `SC.{W/D}` instruction with write-release semantics, that is, only when `SC.REL.{W/D}` is executed (globally visible), all access operations can start executing (globally visible effect).

The `LL.ACQ.{W/D}` instruction fetches a word/double word of data symbol expansion from the specified address in memory and writes it to the general-purpose register rd, and at the same time records the access address and places a flag (LLbit set to 1). The `SC.REL.{W/D}` instruction conditionally writes the word/double-word value of `[31:0]/[63:0]` in the general-purpose register rd to the specified address in the memory, whether or not to write to the memory depends on the LLbit, and only when the LLbit is 1 does it really generate a write action, otherwise it does not write. `SC.REL` instruction will write the flag of success or failure of its execution (which can be simply understood as the LLbit value seen by the `SC.REL` instruction when it is executed) into the general-purpose register rd and return it, regardless of whether it writes to the memory or not.

During paired `LL-SC` execution, the following events clear the LLbit to zero:

- An `ERTN` instruction is executed and the KLO bit in `CSR.LLBCTL` is not equal to 1 at the time of execution.
- another processor core or Cache Coherent master completes a store operation on the Cache line corresponding to the address of the LLbit.

LL.ACQ and SC.REL instructions always require a natural alignment of the access address, if this condition is not met a non-alignment exception is triggered.

If the LL.ACQ and SC.REL instructions direct that the store access attribute of the access address is not cache-consistent (CC), then the result of the execution is indeterminate.

2.2.8. Barrier Instructions

2.2.8.1. DBAR

Instruction formats:

```
dbar        hint
```

The `DBAR` instruction is used to complete the barrier function between load/store memory access operations. The immediate hint it carries is used to indicate the synchronization object and synchronization degree of the barrier.

A hint value of `0` is mandatory by default, and it indicates a fully functional synchronization barrier. Only after all previous load/store access operations are completely executed, the `DBAR 0` instruction can be executed; and only after the execution of `DBAR 0` is completed, all subsequent load/store access operations can be executed.

If there is no special function implementation, all other hint values must be executed according to `hint=0`.

2.2.8.2. IBAR

Instruction formats:

```
ibar        hint
```

The IBAR instruction is used to complete the synchronization between the store operation and the instruction fetch operation within a single processor core. The immediate hint it carries is used to indicate the synchronization object and synchronization degree of the barrier.

A hint value of 0 is mandatory by default. It can ensure that the instruction fetch after the `IBAR 0` instruction must be able to observe the execution effect of all store operations before the `IBAR 0` instruction.

2.2.9. CRC Check Instructions

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

Instruction formats:

```
crc.w.b.w       rd, rj, rk
crc.w.h.w       rd, rj, rk
crc.w.w.w       rd, rj, rk
crc.w.d.w       rd, rj, rk
crcc.w.b.w      rd, rj, rk
crcc.w.h.w      rd, rj, rk
crcc.w.w.w      rd, rj, rk
crcc.w.d.w      rd, rj, rk
```

`CRC[C]W.{B/H/W/D}.W` is used to calculate the CRC-32 checksum, which stores the 32-bit cumulative CRC checksum stored in the general register `rk` in the general register `rj [7:0]`/`[15:0]`/`[31:0]`/`[63:0]` bit message, get a new 32-bit CRC checksum according to the CRC-32 checksum generation algorithm, and write it after sign extension into the general register `rd`. The difference is that `CRC.W.{B/H/W/D}.W` uses IEEE802.3 polynomial (polynomial value is `0xEDB88320`), `CRCC.W.{B/H/W/D}.W` uses Castagnoli polynomial (polynomial value is `0x82F63B78`). The CRC instructions defined in this manual only support the “LSB first” (little endian) standard, which means that the lowest bit of data (little endian) is transmitted first, and the lowest bit of the data is mapped to the coefficient of the most significant term of the message polynomial.

```
CRC.W.B.W:
    chksum = CRC32(GR[rk][31:0], GR[rj][7:0], 8, 0xEDB88320)
    GR[rd] = SignExtend(chksum, GRLEN)

CRC.W.H.W:
    chksum = CRC32(GR[rk][31:0], GR[rj][15:0], 16, 0xEDB88320)
    GR[rd] = SignExtend(chksum, GRLEN)
```

```
CRC.W.W.W:
    chksum = CRC32(GR[rk][31:0], GR[rj][31:0], 32, 0xEDB88320)
    GR[rd] = SignExtend(chksum, GRLEN)

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

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

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

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

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

2.2.10. Other Miscellaneous Instructions

2.2.10.1. `syscall`

Instruction formats:

```
syscall     code
```

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

The information carried in the code field in the instruction code can be used as a parameter passed by the exception handling routine.

2.2.10.2. `break`

Instruction formats:

```
break       code
```

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

The information carried in the code field in the instruction code can be used as a parameter passed by the exception handling routine.

2.2.10.3. `ASRT{LE/GT}.D`

Instruction formats:

```
asrtle.d        rj, rk
asrtgt.d        rj, rk
```

The value in general register `rj` and general register `rk` are compared as signed numbers. If the comparison conditions are not met, an exception for address bound checking is triggered. For the `ASRTLE.D` instruction, if the value in the general register `rj` is greater than the value in the general register `rk`, an exception is triggered; for the `ASRTGT.D` instruction, if the value in the general register `rj` is less than or equal to the value in the general register `rk`, an exception is triggered.

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

Instruction formats:

```
rdtimel.w       rd, rj
rdtimeh.w       rd, rj
rdtime.d        rd, rj
```

The LoongArch instruction system defines-a constant frequency timer, whose main body is-a 64-bit counter called StableCounter. StableCounter is set to 0 after reset, and then increments by `1` every counting clock cycle. When the count reaches all 1s, it automatically wraps around to `0` and continues to increment. At the same time, each timer has a software-configurable globally unique-number, called Counter ID. The characteristic of the constant frequency timer is that its timing frequency remains unchanged after reset, no matter how the clock frequency of the processor core changes.

The `RDTIME{L/W}.W` and `RDTIME.D` instructions are used to read constant frequency timer information, the StableCounter value is written into the general register `rd`, and the Counter ID number information is written into the general register `rj`. The difference between the three instructions is the difference in the Stable Counter information read. `RDTIMEL.W` reads the `[31:0]` bits of the Counter, `RDTIMEH.W` reads the `[63:32]` bits of the Counter, and `RDTIME.D` reads The entire 64-bit Counter value. On a 64-bit processor, the 32-bit value read by the `RDTIME{L/H}.W` instruction is sign extension and written to the general register `rd`. The `RDTIME(L/H).W` instruction is defined so that the 64-bit Counter can also be accessed on a 32-bit processor.

2.2.10.5. `cpucfg`

Instruction formats:

```
cpucfg      rd, rj
```

The `CPUCFG` instruction is used to dynamically identify which features of LoongArch are implemented in the running processor during the execution of the software. The realization of the functional characteristics of these instruction systems is recorded in the series of configuration information words. One configuration information word can be read once the CPUCFG instruction is executed.

When using the CPUCFG instruction, the source operand register rj stores the number of the configuration information word to be accessed, and the configuration information word information read after the instruction is executed is written into the general register rd. In LA64, each configuration information word is 32 bits, which is written into the result register after the sign extension.

The configuration information word contains-series of configuration bits (fields), and its record form is CPUCFG.<configuration word number>.<configuration information mnemonic name>[bit subscript], where the single bit configuration bit is marked as bitXX, which means The XX bit of the configuration word; the bit under the multi-bit configuration field is marked as bitXX:YY, which means the continuous (XX-YY+1) bit from the XX bit to the YY bit of the configuration word. For example, the 0th bit in the configuration word No.1 is used to indicate whether to implement LA32. Record this configuration information as CPUCFG.1.LA32[bit0], where 0x1 indicates that the font size of the configuration information word is No.1, and LA32 indicates this configuration The mnemonic name of the information field is called LA32, and bit 0 means that the field of LA32 is located at bit 0 of the configuration word. The PALEN field of the number of physical address bits supported by the 11th to 4th digits of the configuration word No.1 is recorded as CPUCFG.1.PALEN[itl1:4].

The configuration information accessible by the CPUCFG instruction in the Godson architecture is listed in the table. CPUCFG access to undefined configuration words will read back all 0 values. The undefined field in the defined configuration word can be read back to any value when CPUCFG is executed, and the software should not make any interpretation of it.

Table 3. The configuration information accessible by the CPUCFG instruction

Word number	Bit number	Annotation	Implication
0x0	31:0	PRID	Processor Identity

Word number	Bit number	Annotation	Implication
0x1	1:0	ARCH	2'b00 indicates the implementation of simplified LA32; 2'b01 indicates the implementation of LA32; 2'b10 indicates the implementation of LA64; 2'b11 is reserved.
	2	PGMMU	1 indicates that the MMU supports page mapping mode
	3	IOCSR	1 indicates support for the IOCSR instruction
	11:4	PALEN	The supported physical address bits PALEN value minus 1
	19:12	VALEN	The supported virtual address bits VALEN value minus 1
	20	UAL	1 indicates support for non-aligned memory access
	21	RI	1 indicates support for page attribute of "Read Inhibit"
	22	EP	1 indicates support for page attribute of "Execution Protection"
	23	RPLV	1 indicates support for page attributes of RPLV
	24	HP	1 indicates support for page attributes of huge page
	25	CRC	1 indicates that support CRC instruction That is, information such as "Loongson3A5000 @ 2.5GHz"
	26	MSG_INT	1 indicates that the external interrupt uses the message interrupt mode, otherwise it is the level interrupt line mode

Word number	Bit number	Annotation	Implication
2	0	FP	1 indicates support for basic floating-point instructions
	1	FP_SP	1 indicates support for single-precision floating-point numbers
	2	FP_DP	1 indicates support for double-precision floating-point numbers
	5:3	FP_ver	The version number of the floating-point arithmetic standard. 1 is the initial version number, indicating that it is compatible with the IEEE 754-2008 standard
	6	LSX	1 indicates support for 128-bit vector extension
	7	LASX	1 indicates support for 256-bit vector expansion
	8	COMPLEX	1 indicates support for complex vector operation instructions
	9	CRYPTO	1 indicates support for encryption and decryption vector instructions
	10	LVZ	1 indicates support for virtualization expansion
	13:11	LVZ_ver	The version number of the virtualization hardware acceleration specification. 1 is the initial version number
	14	LLFTP	1 indicates support for constant frequency counter and timer
	17:15	LLFTP_ver	Constant frequency counter and timer version number. 1 is the initial version
	18	LBT_X86	1 indicates support for X86 binary translation extension
	19	LBT_ARM	1 indicates support for ARM binary translation extension
	20	LBT_MIPS	1 indicates support for MIPS binary translation extension
	21	LSPW	1 indicates support for the software page table walking instruction
	22	LAM	1 indicates support AM* atomic memory access instruction
	24	HPTW	1 indicates support Page Table Walker
	25	FRECIPE	1 indicates support FRECIPE.{S/D}、FRSQRTE.{S/D}. If 128-bit vector extension is also supported, VFRECIPE.{S/D}、VFRSQRTE.{S/D} is supported. If 256-bit vector extension is also supported, XVFRECIPE.{S/D}、XVFRSQRTE.{S/D} is supported.
	26	DIV32	1 indicates that DIV.W[U] and MOD.W[U] instructions on 64-bit machines compute only the low 32-bit data of the input register
	27	LAM_BH	1 indicates support AM{SWAP/ADD}[_DB].{B/H}.
	28	LAMCAS	1 indicates support AMCAS[_DB].{B/H/W/D}.
	29	LLACQ_SCREL	1 indicates support LLACQ.{W/D}、SCREL.{W/D}.
	30	SCQ	1 indicates support SC.Q.

Word number	Bit number	Annotation	Implication
3	0	CCDMA	1 indicates support for hardware Cache coherent DMA
	1	SFB	1 indicates support for Store Fill Buffer (SFB)
	2	UCACC	1 indicates support for ucacc win
	3	LLEXC	1 indicates support for LL instruction to fetch exclusive block function_
	4	SCDLY	1 indicates support random delay function after SC
	5	LLDBAR	1 indicates support LL automatic with dbar function
	6	ITLBTHMC	1 indicates that the hardware maintains the consistency between ITLB and TLB
	7	ICHMC	1 indicates that the hardware maintains the data consistency between ICache and DCache in one processor core
	10:8	SPW_LVL	The maximum number of directory levels supported by the page walk instruction
	11	SPW_HP_HF	1 indicates that the page walk instruction fills the TLB in half when it encounters a large page
	12	RVA	1 indicates that the software configuration can be used to shorten the virtual address range
	16:13	RVAMAX-1	The maximum configurable virtual address is shortened by -1
	17	DBAR_hints	1 indicates that the non-0 value of the DBAR is implemented according to the recommended meaning of the manual.
	23	LD_SEQ_SA	1 indicates that the hardware is enabled to guarantee sequential execution of load operations at the same address.
0x4	31:0	CC_FREQ	Constant frequency timer and the crystal frequency corresponding to the clock used by the timer
0x5	15:0	CC_MUL	Constant frequency timer and the corresponding multiplication factor of the clock used by the timer
	31:16	CC_DIV	Constant frequency timer and the division coefficient corresponding to the clock used by the timer
0x6	0	PMP	1 indicates support for the performance counter
	3:1	PMVER	In the performance monitor, the architecture defines the version number of the event, and 1 is the initial version
	7:4	PMNUM	Number of performance monitors minus 1
	13:8	PMBITS	Number of bits of a performance monitor minus 1
	14	UPM	1 indicates support for reading performance counter in user mode

Word number	Bit number	Annotation	Implication
0x10	0	L1 IU_Present	1 indicates that there is a first-level instruction Cache or a first-level unified Cache
	1	L1 IU Unify	1 indicates that the Cache shown by L1 IU_Present is the unified Cache
	2	L1 D Present	1 indicates there is a first-level data Cache
	3	L2 IU Present	1 indicates there is a second-level instruction Cache or a second-level unified Cache
	4	L2 IU Unitfy	1 indicates that the Cache shown by L2 IU_Present is the unified Cache
	5	L2 IU Private	1 indicates that the Cache shown by L2 IU_Present is private to each core
	6	L2 IU Inclusive	1 indicates that the Cache shown by L2 IU_Present has an inclusive relationship to the lower levels (L1)
	7	L2 D Present	1 indicates there is a secondary data Cache
	8	L2 D Private	1 indicates that the secondary data Cache is private to each core
	9	L2 D Inclusive	1 indicates that the secondary data Cache has a containment relationship to the lower level (L1)
	10	L3 IU Present	1 indicates there is a three-level instruction Cache or a three-level system Cache
	11	L3 IU Unify	1 indicates that the Cache shown by L3 IU_Present is unified Cache
	12	L3 IU Private	1 indicates that the Cache shown by L3 IU_Present is private to each core
	13	L3 IU Inclusive	1 indicates that the Cache shown by L3 IU_Present has an inclusive relationship to the lower levels (L1 and L2)
	14	L3 D Present	1 indicates there is a three-level data Cache
	15	L3 D Private	1 indicates that the three-level data Cache is private to each core
	16	L3 D Inclusive	1 indicates that the three-level data Cache has an inclusive relationship to the lower levels (L1 and 12)
0x11	15:0	Way-1	Number of channels minus 1 (Cache corresponding to L1 IU_Present in configuration word 10)
	23:16	Index-log2	log2(number of Cache rows per channel) (Cache corresponding to L1 IU_Present in configuration word 10)
	30:24	Linesize-log2	log2(Cache line bytes) (Cache corresponding to L1 IU_Present in configuration word 10)

Word number	Bit number	Annotation	Implication
`0x12`	`15:0`	`Way-1`	Number of channels minus `1` (Cache corresponding to `L1 D Present` in configuration word `10`)
	`23:16`	`Index-log2`	`log2(number of Cache rows per channel)` (Cache corresponding to `L1 D Present` in configuration word `10`)
	`30:24`	`Linesize-log2`	`log2(Cache row bytes)` (Cache corresponding to `L1 D Present` in configuration word `10`)
`0x13`	`15:0`	`Way-1`	Number of channels minus `1` (Cache corresponding to `L2 IU Present` in configuration word `10`)
	`23:16`	`Index-log2`	`log2(number of Cache rows per channel)` (Cache corresponding to `L2 IU Present` in configuration word `10`)
	`30:24`	`Linesize-log2`	`log2(Cache row bytes)` (Cache corresponding to `L2 IU Present` in configuration word `10`)
`0x14`	`15:0`	`Way-1`	Number of channels minus `1` (Cache corresponding to `L3 IU Present` in configuration word `10`)
	`23:16`	`Index-log2`	`log2(number of Cache rows per channel)` (Cache corresponding to `L3 IU Present` in configuration word `10`)
	`30:24`	`Linesize-log2`	`log2(Cache row bytes)` (Cache corresponding to `L3 IU Present` in configuration word `10`)

Chapter 3. Basic Floating-Point Instructions

This chapter will introduce the floating-point number instructions in the basic part of the non-privileged subset of LoongArch. The function definition of the basic floating-point instructions in LoongArch follows the IEEE 754-2008 standard.

Basic floating-point instructions cannot be implemented separately from basic integer instructions. Generally speaking, it recommends that implementing both basic integer instructions and basic floating-point instructions at the same time. However, for some embedded applications that are cost-sensitive and have extremely low floating-point processing performance requirements, the architecture specification also allows not to implement basic floating-point instructions, or only implement single-precision floating-point numbers and word integers in basic floating-point instructions. Whether the implementation of basic floating-point instructions includes instructions for operating double-precision floating-point numbers and double-word integers has nothing to do with whether the architecture is LA32 or LA64.

3.1. Programming Model of Basic Floating-Point Instructions

The basic floating-point instruction programming model described in this section only involves the content that application software developers need to pay attention to. When software personnel use basic floating-point instructions to program, they are on the basis of the basic integer instruction programming model, and then proceed to involve the content described in this section.

3.1.1. Floating-Point Data Types

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

3.1.1.1. Single-precision Floating-point

Single-precision floating-point numbers have a length of 32 bits and are organized into the following format:

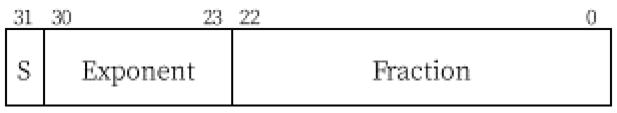

Figure 3. Single-precision floating-point number format

According to the different values of the fields of `S`, `Exponent` and `Fraction`, the floating-point number values represented are shown in the table:

Table 4. Single-precision floating-point number calculation method

`Exponent`	`Fraction`	`S`	`bit[22]`	`V`
`0`	`0`	`0`	`0`	`+0`
		`1`	`0`	`-0`
`0`	`!=0`	`0`	Any value	Denormalized number, the value is `+2`$^{-126}$`×(0.Fraction)`
		`1`	Any value	Denormalized number, the value is `-2`$^{-126}$`×(0.Fraction)`

Exponent	Fraction	S	bit[22]	V
[1,0xFE]	Any value	0	Any value	Normalized number, the value is $+2^{\text{Exponent}-127}\times(1.\text{Fraction})$
		1	Any value	Normalized number, the value is $-2^{\text{Exponent}-127}\times(1.\text{Fraction})$
0xFF	0	0	0	+∞
		1	0	-∞
0xFF	!=0	Any value	0	Signaling Not a Number, SNaN
		Any value	1	Quiet Not a Number, QNaN

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

3.1.1.2. Double-precision Floating-point

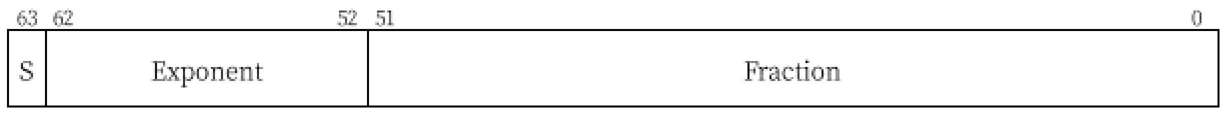

Figure 4. Double-precision floating-point number format

According to the different values of the fields of S, Exponent and Fraction, the floating-point number values represented are shown in the table:

Table 5. Double-precision floating-point number calculation method

Exponent	Fraction	S	bit[51]	V
0	0	0	0	+0
		1	0	-0
0	!=0	0	Any value	Denormalized number, the value is $+2^{-1022}\times(0.\text{Fraction})$
		1	Any value	Denormalized number, the value is $-2^{-1022}\times(0.\text{Fraction})$
[1,0x7FE]	Any value	0	Any value	Normalized number, the value is $+2^{\text{Exponent}-1023}\times(1.\text{Fraction})$
		1	Any value	Normalized number, the value is $-2^{\text{Exponent}-1023}\times(1.\text{Fraction})$
0x7FF	0	0	0	+∞
		1	0	-∞
0x7FF	!=0	Any value	0	Signaling Not a Number, SNaN
		Any value	1	Quiet Not a Number, QNaN

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

3.1.1.3. Non-numerical Result of Instructions

The non-numerical results produced by floating-point number instructions either come from NaN propagation or are directly generated. There are two situations where NaN propagation is required.

Case 1: When the instruction generates an Invalid Operation floating-point exception due to a source operand containing SNaN, but the InvalidOperation floating-point exception enable is invalid, a QNaN result will be generated at this time. The value of this QNaN is to select the SNaN with the highest priority in the source operand and propagate it to the corresponding NaN.

The rule for determining the priority of the source operand is: if there are two source operands `fj` and `fk`, then the priority of `fj` is higher than `fk`; if there are three source operands `fa`, `fj` and `fk`, then the priority of fa is higher than `fj`, `fj` have higher priority than `fk`.

The value generation rules for propagation of SNaN to QNaN are as follows:

- If the result is the same length as the source operand, then the highest position of the SNaN mantissa will be propagated to `1`, and the remaining bits remain unchanged. If the result is narrower than the source operand, then keep the high bits of the mantissa, discard the low bits that exceed the range, and finally set the highest bit of the mantissa to `1`.
- If the result is wider than the source operand, then the lowest bit of the mantissa will be filled with `0`, and finally the highest position of the mantissa will be `1`.

Case 2: When there is no SNaN in the source operand but QNaN exists, the QNaN with the highest priority is selected as the result of this instruction. At this time, the way of judging the priority of the source operand is the same as in the above situation.

Except for the above two cases, other cases that need to produce QNaN results will be directly set to the default QNaN value. The default single-precision QNaN value is `0x7FC00000`, and the default double-precision QNaN value is `0x7FF8000000000000`.

3.1.2. Fixed-Point Data Types

Some floating-point instructions (such as floating-point conversion instructions) also manipulate fixed-point data, including **W**ord (W, length 32b), and **L**ongword (L, length 64b). Both word and longword data types use two's complement encoding.

3.1.3. Registers

Floating-point instruction programming involves registers such as **F**loating-point **R**egister (FR), **C**ondition **F**lag **R**egister (CFR) and **F**loatingpoint **C**ontrol and **S**tatus **R**egister (FCSR).

3.1.3.1. Floating-point Registers

There are 32 FRs, denoted as `f0`-`f31`, each of which can be read and written. Only when only floating-point instructions that manipulate single-precision floating-point numbers and word integers are implemented, the length of FR is 32 bits. Under normal circumstances, the length of FR is 64 bits, regardless of the LA32 or LA64. There is an "orthogonal" relationship between basic floating-point instructions and floating-point registers, that is, from an architectural perspective, any floating-point register operand in these instructions can use any one of the 32 FRs.

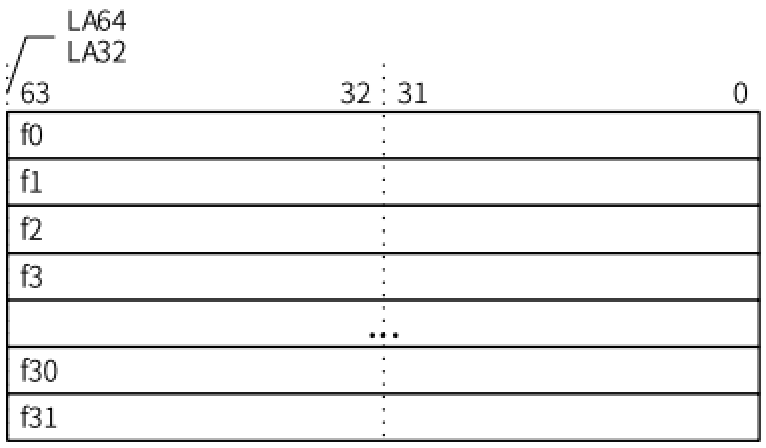

Figure 5. Floating-point Registers

When the floating-point register records a single-precision floating-point number or word integer, the data always appears in the `[31:0]` bits of the floating-point register, at this time the `[63:32]` bits of the floating-point register can be any value.

3.1.3.2. Condition Flag Register

There are 8 CFRs, denoted as `fcc0`-`fcc7`, each of which can be read and written. The length of CFR is 1 bit. The result of the floating-point comparison will be written into the condition flag register. When the comparison result is true, it is set to `1`, otherwise it is set to `0`. The judgment condition of the floating-point branch instruction comes from the condition register.

3.1.3.3. Floating-point Control and Status Register

There are 4 FCSRs, denoted as `fcsr0`-`fcsr3`. Among them, `fcsr1`-`fcsr3` are aliases of some fields in `fcsr0`, that is, accessing `fcsrl`-`fcsr3` is actually accessing some fields of `fcsr0`. When the software writes `fcsr1`-`fcsr3`, the corresponding field in `fcsr0` is modified while the remaining bits remain unchanged. The definition of each field of `fcsr0` is shown in the table.

Table 6. Definitions of FCSR0 Register Fields

Bits	Name	Read&write	Description
`4:0`	Enables	RW	The floating-point operation `VZOUI` exceptions each allow the enable bit to trigger the exception trap. 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`.

`9:8`	RM	RW	Rounding mode control. It contains 4 legal values, each with the following meaning:
			
			`0`: RNE, corresponding to roundTiesToEven in IEEE 754-2008;
			
			`1`: RZ, corresponding to roundTowardZero in IEE 754-2008;
			
			`2`: RP, corresponding to roundTowardsPositive in IEEE 754-2008;
			
			`3`: RM, corresponding to roundTowardsNegative in IEEE 754-2008.

Bits	Name	Read&write	Description
20:16	Flags	RW	Since the last time the Flags field was cleared by the software, the cumulative status of various floating-point operations VZOUI exceptions that were generated but not caught. 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.
28:24	Cause	RW	The VZOUI exception caused by the last 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.

FCSR1 is the alias of the Enables field in FCSR0. Its location is the same as in FCSR0.

FCSR2 is the alias of the Cause and Flags fields in FCSR0. The location of each field is consistent with FCSR0.

FCSR3 is the alias of the RM field in FCSR0. Its location is the same as in FCSR0.

3.1.4. Floating-Point Exceptions

Floating-point exception means that when the floating-point processing unit cannot process the operand or the result of floating-point calculation in a conventional manner, the floating-point functional unit will generate a corresponding exception.

The basic floating-point instructions support five floating-point exceptions defined by IEEE 754-2008:

- **I**nexact (I)
- **U**nderflow (U)
- **O**verflow (O)
- **D**ivision by Zero (Z)
- **I**nvalid Operation (V)

Each bit of the Cause field in FCSR0 corresponds to the above-mentioned exceptions. After the execution of each floating-point instruction, the occurrence of its exception will be updated to the Cause field of FCSR0.

FCSR0 also contains an enable bit (Enables field) for each floating-point exception. The enable bit determines whether an exception generated by the floating-point processing unit will trigger an exception trap or set a status flag. When a floating-point exception occurs, if its corresponding Enable bit is 1, then a floating-point exception trap will be triggered; if its corresponding Enable bit is 0, then the floating-point exception trap will not be triggered, but Set the corresponding position of the Flag field in FCSR0 to 1.

During the execution of a floating-point instruction, multiple floating-point exceptions can be generated at the same time.

When a floating-point exception is generated during the execution of a floating-point instruction but the floating-point exception is not triggered, the floating-point processing unit will generate a default result. Different exceptions produce default results in different ways. The table lists specific generation rules.

Table 7. Default results of floating-point exceptions

Area	Descriptio n	Rounding mode	Default result
`I`	**I**nexact	Any mode	The result after rounding or the result after overflow
`U`	**U**nderflow	RNE	The result after rounding may be `0`, subnormal, the normal number with the smallest absolute value (single-precision: $\pm 2^{-126}$, double-precision: $\pm 2^{-1022}$)
		RZ	The result after rounding, may be `0`, subnormal
		RP	The rounded result may be `0`, subnormal, the smallest positive normal number (single-precision: $+2^{-126}$, double-precision: $+2^{-1022}$)
		RM	The rounded result may be `0`, subnormal, the largest negative normal number (single-precision: -2^{-126}, double-precision: -2^{-1022})
`O`	**O**verflow	RNE	Set the result to `+∞` or `-∞` according to the sign of the intermediate result
		RZ	Set the result to the maximum number according to the sign of the intermediate result
		RP	Correct negative overflow to the smallest negative number, and correct positive overflow to `+∞`
		RM	Correct the positive overflow to the largest positive number, and correct the negative overflow to `-∞`
`Z`	Division by **Z**ero	Any mode	Provide a corresponding signed infinity number
`V`	In**V**alid Operation	Any mode	Provide a QNaN

3.1.4.1. Illegal Operation Exception (`V`)

An invalid operation exception notification signal will be sent if and only if there is no valid defined result. If no exception is triggered, a QNaN will be generated. Please refer to Characteristics of Accessing Control and Status Registers of the IEEE 754-2008 specification for specific determination details of extraordinary operation exceptions.

If an exception is allowed to fall into: the result register is not modified, the source register remains.

If exceptions are prohibited from trapping: If no other exceptions occur, QNaN is written to the target register.

3.1.4.2. Division by Zero Exception (`Z`)

In the division operation, when the divisor is `0` and the dividend is-a limited non-zero data, the division by zero exception is signaled.

If an exception is allowed to fall into: the result register is not modified, the source register remains

If an exception is forbidden to fall into: if no trap occurs, the result is a signed infinite value.

3.1.4.3. Overflow Exception (`O`)

Regarding the exponent field as an unbounded rounding of the intermediate result, when the absolute value of the result obtained exceeds the maximum finite number of the target format, an overflow exception will be notified.(This exception sets both inexact exception and flag bit)

If an exception is allowed to fall into: the result register is not modified, the source register remains.

If exceptions are forbidden to fall into: 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`)

When the detection result is a small non-zero value, an underflow exception will occur. The way to detect small non-zero values is to detect after rounding. that is, for a non-zero result is in `(-2Emin, 2Emin)`, the result is considered to be a small non-zero value (Single-precision number `Emin=-126`, double-precision number `Emin=-1022`). When `FCSR.Enable.U=0`, if the result is detected, a non-zero tiny value:

1. If the final rounded result of the floating-point operation is inaccurate, both `U` and `I` in `FCSR.Cause` should be set to `1`;
2. If the final rounded result of the floating-point operation is accurate, then `U` and `I` in `FCSR.Cause` are not set to `1`.

When `FCSR.Enable.U=1`, if the result is a non-zero tiny value, regardless of whether the final rounded result of the floating-point operation is accurate or inaccurate, it will trigger a floating-point exception trap.

3.1.4.5. Inexact Exception (`I`)

FPU generates inaccurate exceptions when the following situations occur:

- Rounding result is imprecise.
- The rounding result overflows, and the enable bit of the overflow exception is not set.

If an exception is allowed to fall: If an inexact exception trap is enabled, the result register is not modified and the source register is retained. Because this execution mode affects performance, inaccurate exception traps are only enabled when necessary.

If an exception is prohibited, trapping is prohibited: If no other software trap occurs, the rounding or overflow result is sent to the destination register.

3.2. Overview of Floating-Point Instructions

The instructions described in this section, except for `FLDX.{S/D}`, `FSTX.{S/D}`, `FLD{GT/LE}.{S/D}` and `FST{GT/LE}.{S/D}` these 12 The floating-point memory access instructions only belong to the LA64, and all other floating-point instructions are applicable to both LA32 and LA64.

3.2.1. Floating-Point Arithmetic Operation Instructions

3.2.1.1. `F{ADD/SUB/MUL/DIV}.{S/D}`

Instruction formats:

```
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 performs the operation that the single-precision/double-precision floating-point number in the floating-point register `fj` plus the single-precision/double-precision floating-point number in the floating-point register `fk`; then writes the result of the single-precision/double-precision floating-point number to floating-point register `fd`. Floating-point addition operation follows the specification of `addition(x,y)` operation in the IEEE 754-2008 standard.

```
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 performs the operation that the single-precision/double-precision floating-point number in the floating-point register `fj` minus the single-precision/double-precision floating-point number in the floating-point register `fk`, and write the result of the single-precision/double-precision floating-point number to floating-point register `fd`. The floating-point subtraction operation follows the `subtraction(xy)` operation specification in the IEEE 754-2008 standard.

```
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 performs the operation that multiplies the single-precision/double-precision floating-point number in the floating-point register `fj` by the single-precision/double-precision floating-point number in the floating-point register `fk`, and writes the result of the single-precision/double-precision floating-point number To the floating-point register `fd`. The floating-point multiplication operation follows the `multiplication(xy)` operation specification in the IEE 754-2008 standard.

```
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 performs the operation that divides the single-precision/double-precision floating-point number in the floating-point register `fj` by the single-precision/double-precision floating-point number in the floating-point register `fk`, and writes the result of the single-precision/double-precision floating-point number To the floating-point register `fd`. The floating-point division operation follows the `division(x, y)` operation specification in the IEEE 754-2008 standard.

```
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])
```

When the operand is a single-precision floating-point number, the upper 32 bits of the resulting floating-point register can be any value.

3.2.1.2. `F{MADD/MSUB/NMADD/NMSUB}.{S/D}`

Instruction formats:

```
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 performs the operation that multiplies the single-precision/double-precision floating point number in floating point register `fj` with the single-precision/double-precision floating point number in floating point register `fk`. The result is added to the single-precision/double-precision floating point number in the floating point register `fa`. The result of the single-precision/double-precision floating point number is written to 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 performs the operation that multiplies the single-precision/double-precision floating-point number in the floating-point register `fj` with the single-precision/double-precision floating-point number in the floating-point register `fk`, the result minus the floating-point register `fa` Single-precision/double-precision floating-point numbers, the single-precision/double-precision floating-point number results obtained are written into 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 performs the operation that multiplies the single-precision/double-precision floating-point number in the floating-point register `fj` with the single-precision/double-precision floating-point number in the floating-point register `fk`, the result plus the single-precision/double-precision floating-point number in the floating-point register `fa` Precision/double-precision floating-point number, the obtained single-precision/double-precision floating-point number result is negative and written into 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 performs the operation that multiplies the single-precision/double-precision floating-point number in the floating-point register `fj` with the single-precision/double-precision floating-point number in the floating-point register `fk`, the result minus the floating-point register `fa` Single-precision/double-precision floating-point number, the result of the single-precision/double-precision floating-point number obtained is negative and written into 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 fusion multiply-add operations follow the specification of the fusedMultiplyAdd(xy,z) operation in the IEEE 754-2008 standard.

3.2.1.3. `F{MAX/MIN}{S/D}`

Instruction formats:

```
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 the larger of the single-precision/double-precision floating-point number in the floating-point register `fj` and the single-precision/double-precision floating-point number in the floating-point register fk to write into the floating-point register fd. The operation of these two instructions follows the specification of maxNum(x,y) operation in the IEEE 754-2008 standard.

```
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 the smaller of the single-precision/double-precision floating-point number in the floating-point register fj and the single-precision/double-precision floating-point number in the floating-point register fk to write into the floating-point register fd. The operation of these two instructions follows the minNum(x,y) operation specification in the IEEE 754-2008 standard.

```
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])
```

3.2.1.4. F{MAXA/MINA}.{S/D}

Instruction formats:

```
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 the larger absolute value of the single-precision/double-precision floating-point number in the floating-point register fj and the single-precision/double-precision floating-point number in the floating-point register fk to write to the floating-point register fd. The floating-point addition operation follows the specification of maxNumMag(x.v) operation in IEEE 754-2008 standard.

```
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 the smaller absolute value of the single-precision/double-precision floating-point number in the floating-point register fj and the single-precision/double-precision floating-point number in the floating-point register fk to write to the floating-point register fd. The floating-point addition operation follows the specification of `minNumMag(x,y)` operation in IEEE 754-2008 standard.

```
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])
```

3.2.1.5. `F{ABS/NEG}.{S/D}`

Instruction formats:

```
fabs.s      fd, fj
fabs.d      fd, fj
fneg.s      fd, fj
fneg.d      fd, fj
```

The `FABS.{S/D}` instruction selects the single-precision/double-precision floating-point number in the floating-point register `fj`, takes its absolute value(that is, the symbol position is 0, and other parts remain unchanged), and writes it into the floating-point register `fd`. Floating-point addition operations follow the specification of `abs(x)` operation in the EEE 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 the single-precision/double-precision floating-point number in the floating-point register `fj`, takes the opposite number(that is, inverts the sign bit, and other parts remain unchanged), and writes it into the floating-point register `fd`. Floating-point addition operations follow the negate(x) operation specification in the EEE 754-2008 standard.

```
FNEG.S:
    FR[fd][31:0] = FP32_negate(FR[fj][31:0])

FNEG.D:
    FR[fd] = FP64_negate(FR[fj])
```

3.2.1.6. `F{SQRT/RECIP/RSQRT}.{S/D}`

Instruction formats:

```
fsqrt.s         fd, fj
fsqrt.d         fd, fj
frecip.s        fd, fj
frecip.d        fd, fj
frsqrt.s        fd, fj
frsqrt.d        fd, fj
```

These instructions are operations related to square root and reciprocal.

The `FSQRT.{S/D}` instruction selects the single-precision/double-precision floating-point number in the floating-point register `fj`, and writes the single-precision/double-precision floating-point number obtained after the square root to the floating-point register `fd`. The floating-point root operation follows the `squareRoot(x)` operation specification 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 the single-precision/double-precision floating-point number in the floating-point register `fj`, divides the floating-point number by 1.0, and writes the resulting single-precision/double-precision floating-point number into the floating-point register `fd`. It 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 the single-precision/double-precision floating-point number in the floating-point register `fj`, takes its square root and then divides the obtained single-precision/double-precision floating-point number by 1.0, and the obtained single-precision/double-precision floating-point number is written to the floating-point register `fd`. The floating-point squared-inverse operation follows the specification of `rSqrt(x)` operation in IEEE 754-2008 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(R[fj]))
```

3.2.1.7. F{SCALEB/LOGB/COPYSIGN}.{S/D}

Instruction formats:

```
fscaleb.s       fd, fj, fk
fscaleb.d       fd, fj, fk
flogb.s         fd, fj
flogb.d         fd, fj
fcopysign.s     fd, fj, fk
fcopysign.d     fd, fj, fk
```

The `FSCALEB.{S/D}` instruction selects the single-precision/double-precision floating point number a in the floating point register `fj`, Then take the word/double word integer `N` in the floating point register `fk`, and calculate a*2N, The obtained single-precision/double-precision floating point number is written to the floating point register `fd`. These two instructions follow the IEEE754-2008 standard `scaleB(x, N)` operation specification.

```
FSCALEB.S:
    FR[fd][31:0] = FP32_scaleB(FR[fj][31:0], FR[fk][31:0])

FSCALEB.D:
    FR[fd] = FP64_scaleB(FR[fj], FR[fk])
```

The `FLOGB.{S/D}` instruction selects the single-precision/double-precision floating-point number in the floating-point register `fj`, calculates its logarithm based on 2, and writes the obtained single-precision/double-precision floating-point number into the floating-point register `fd` . Floating-point exponential operations follow the specification of `logB(x)` operation in the IEEE 754-2008 standard.

```
FLOGB.S:
    FR[fd][31:0] = FP32_logB(FR[fj][31:0])

FLOGB.D:
    FR[fd] = FP64_logB(FR[fj])
```

The `FCOPYSIGN.{S/D}` instruction selects the single-precision/double-precision floating-point number in the floating-point register `fj`, and changes its sign bit to the sign bit of the single-precision/double-precision floating-point number in the floating-point register `fk`, and the new one is obtained Single-precision/double-precision floating-point numbers are written into the floating-point register `fd`. The floating-point copy sign operation follows the specification of `copySign(x, y)` operation in the IEEE 754-2008 standard.

```
FCOPYSIGN.S:
    FR[fd][31:0] = FP32_copySign(FR[fi][31:01, FR[fk][31:0]])

FCOPYSIGN.D:
    FR[fd] = FP64_copySign(FR[fj], FR[fk])
```

3.2.1.8. FCLASS.{S/D}

Instruction formats:

```
fclass.s    fd, fj
fclass.d    fd, fj
```

This instruction judges the category of the floating-point number in the floating-point register fj. The result of the judgment is composed of 10 bits of information. The meaning of each bit is shown in the following table:

Table 8. Results of floating-point classification

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Bit 9
SNaN	QNaN	Negative value				Positive value			
		∞	Normal	Subnormal	0	∞	Normal	Subnormal	0

When the determined data meets the condition corresponding to a certain bit, the corresponding bit of 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])
    sedMultiplyAdd(FR[fj], FR[fk], FR[fa])
```

3.2.1.9. F{RECIPE/RSQRTE}.{S/D}

Instruction formats:

```
frecipe.s           fd, fj
frecipe.d           fd, fj
frsqrte.s           fd, fj
frsqrte.d           fd, fj
```

The FRECIPE.{S/D} instruction selects the single-precision or double-precision floating-point number in the floating-point register fj, calculates the single-precision or double-precision floating-point number approximation obtained by dividing the floating-point number by 1.0, and writes the approximation to the floating-point register fd . The relative error of the approximation is less than 2^-14.

When the input value is 2^N, the output value is 2^-N. The results when the input value is QNaN, SNaN, ±∞,

±0, the conditions for generating floating-point exceptions, and the default results when floating-point exceptions are generated without triggering exceptions are the same as those of the FRECIP.{S/D} instruction.

```
FRECIPE.S:
    FR[fd][31:0] = FP32_reciprocal_estimate(FR[fj][31:0])
FRECIPE.D:
    FR[fd] = FP64_reciprocal_estimate(FR[fj])
```

FRSQRTE.{S/D} instruction selects the single/double precision floating point number in the floating point register fj, first extract the Square Root it, and then divides the approximate result by 1.0, and then writes the obtained single/double precision floating point number into the floating point register fd. The relative error of the obtained approximation is less than 2^-14.

When the input value is 2^2N, the output value is 2^-N. The results when the inputs are QNaN, SNaN, ±∞, and ±0, the conditions for generating floating-point exceptions, and the default results when floating-point exceptions are generated but not triggered are the same as those of the FRSQRT.{S/D} instruction.

```
FRSQRTE.S:
    FR[fd][31:0] = FP32_reciprocal_squareroot_estimate(FR[fj][31:0])
FRSQRTE.D:
    FR[fd] = FP64_reciprocal_squareroot_estimate(FR[fj])
```

3.2.2. Floating-Point Comparison Instructions

3.2.2.1. FCMP.cond.{S/D}

Instruction formats:

```
fcmp.cond.s    cc, fj, fk
fcmp.cond.d    cc, fj, fk
```

This is a floating-point comparison instruction, which stores the result of the comparison into the specified status code (CC). There are 22 types of cond for this instruction. These comparison conditions and judgment standards are listed in the following table.

Table 9. Floating-point comparison conditions and judgment standards

Mnemonic	Cond	Meaning	True Condition	QNaN Exception	IEEE 754-2008 Funtion
CAF	0x0	None	None	No	
CUN	0x8	Incomparable	UN		compareQuietUnordered
CEQ	0x4	Equal	EQ		compareQuietEqual
CUEQ	0xC	Equal or incomparable	UN EQ		
CLT	0x2	Less than	IT		compareQuietLess
CULT	0xA	Less than or incomparable	UN LT		compareQuietLessUnordered
CLE	0x6	Less than or equal to	LT EQ		compareQuietLessEqual
CULE	0xE	Less than or equal to or incomparable	UN LT EQ		compareQuietNotGreater
CNE	0x10	Vary	GT LT		
COR	0x14	Orderly	GT LT EQ		
CUNE	0x18	Incomparable or unequal	UN GT LT		compareSignalingNotEqual
SAF	0x1	None	None	Yes	
SUN	0x9	Is not greater than or equal to	UN		
SEQ	0x5	equal	EQ		compareSignalingEqual
SUEQ	0xD	Not greater than or less than	UN EQ		
SLT	0x3	Less than	IT		compareSignalingLess
SULT	0xB	Is not greater than or equal to	UN LT		compareSignalingLessUnordered
SLE	0x7	Less than or equal to	IT EQ		compareSignalingLessEqual
SULE	0xF	Not greater than	UN LT EQ		compareSignalingNotGreater
SNE	0x11	Vary	GT LT		
SOR	0x15	Orderly	GT LT EQ		
SUNE:	0x19	Incomparable or unequal	UN GT LT		

Note: `UN` means no comparison, `EQ` means equal, `IT` means less than. When there is at least one `NaN` in two operands, the two numbers cannot be compared.

3.2.3. Floating-Point Conversion Instructions

3.2.3.1. `FCVT.S.D`, `FCVT.D.S`

Instruction formats:

```
fcvt.s.d    fd, fj
fcvt.d.s    fd, fj
```

The `FCVT.S.D` instruction performs the operation that the double-precision floating-point number in the floating-point register `fj` to be converted into a single-precision floating-point number, and the obtained single-precision floating-point number is written into the floating-point register `fd`.

```
FCVT.S.D:
    FR[fd][31:0] = FP32_convertFormat(FR[fj], FP64)
```

The `FCVT.D.S` instruction performs the operation that the single-precision floating-point number in the floating-point register `fj` to be converted into a double-precision floating-point number, and the obtained double-precision floating-point number is written into the floating-point register `fd`.

```
FCVT.D.S:
    FR[fd] = FP64_convertFormat(FR[fj][31:0], FP32)
```

The floating-point format conversion operation follows the specification of the `convertFormat(x)` operation in the IEEE 754-2008 standard.

3.2.3.2. `FFINT{S/D}.{W/L}`, `FTINT.{W/L}.{S/D}`

Instruction formats:

```
ffint.s.w       fj
ffint.s.I       fj
ffint.d.w       fj
ffint.d.I       fj
ftint.w.s       fj
ftint.w.d       fj
ftint.l.s       fj
ftint.l.d       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 into a single-degree/double-precision floating-point number, and the obtained single-precision/double-precision floating-point number is written to Floating-point register `fd`. This floating-point format conversion operation follows the `convertFromInt(x)` operation specification in the EEE 754-2008 standard.

```
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)
```

`FTINT{W/L}.{S/D}` instruction selects the single-degree/double-precision floating-point number in the floating-point register `fj` to be converted into an integer/long-integer fixed-point number, and the obtained integer/long-integer fixed-point number is written To the floating-point memory `fd`. According to the different states in `FCSR`, the operations in the IEEE 754-2008 standard followed by this floating-point format conversion operation are shown in the following table.

Table 10. Standard for converting to integer

Rounding mode	Whether to report floating-point imprecision exceptions	IEEE 754-2008 Function
Round to the nearest even number	Yes	`convertToIntegerTiesToEven(X)`
Round towards zero		`convertToIntegerTowardZero(x)`
Round towards positive infinity		`convertToIntegerTowardPositive(x)`
Round towards negative infinity		`converrtToIntegerTowardNegative(x)`
Round to the nearest even number	No	`convertToIntegerExactTiesToEven(x)`
Round towards zero		`convertToIntegerExactTowardZero(x)`
Round towards positive infinity		`convertToIntegerExactTowardPositive(x)`
Round towards negative infinity		`convertToIntegerExactTowardNegative(x)`

```
FTINT.W.S:
    FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], FCSR.Enables.I,
FCSR.RM)

FTINT.W.D:
    FR[fd] = FP64convertToSint32(FR[fj], FCSR.Enables.I, FCSR.RM)

FTINT.L.S:
    FR[fd][31:0] = FP32convertToSint64(FR[fj][31:0], FCSR.Enables.I,
FCSR.RM)

FTINT.L.D:
    FR[fd] = FP64convertToSint64(FR[fj], FCSR.Enables.I, FCSR.RM)
```

3.2.3.3. `FTINT{RM/RP/RZ/RNE}.{W/L}.{S/D}`

Instruction formats:

```
ftintrm.w.s     fd, fj
ftintrm.w.d     fd, fj
ftintrm.l.s     fd, fj
ftintrm.l.d     fd, fj
ftintrp.w.s     fd, fj
ftintrp.w.d     fd, fj
ftintrp.l.s     fd, fj
ftintrp.l.d     fd, fj
ftintrz.w.s     fd, fj
ftintrz.w.d     fd, fj
ftintrz.l.s     fd, fj
ftintrz.l.d     fd, fj
ftintrne.w.s    fd, fj
ftintrne.w.d    fd, fj
ftintrne.l.s    fd, fj
ftintrne.l.d    fd, fj
```

These instructions convert floating-point numbers to fixed-point numbers with the specified rounding pattern. `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 integer-type long integer-type fixed point number, and the resulting integer-type/long integer-type fixed point number is written to the floating-point register `fd`, using the “round to negative infinity” mode.

```
FTINTRM.W.S:
    FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], FCSR.Enables.I, 3)

FTINTRM.W.D:
    FR[fd] = FP64convertToSint32(FR[fj], FCSR.Enables.I, 3)

FTINTRM.L.S:
    FR[fd][31:0] = FP32convertToSint64(FR[fj][31:0], FCSR.Enables.I, 3)

FTINTRM.L.D:
    FR[fd] = FP64convertToSint64(FR[fj], FCSR.Enables.I, 3)
```

`FTINTRP.{W/L}.{S/D}` instruction selects the single-precision/double-precision floating-point number in the floating-point register `fj`, converts it to integer/long-integer fixed point number, and writes the integer/long-integer fixed point number into the floating-point register `fd`, using the "rounding to positive infinity" method.

```
FTINTRP.W.S:
    FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], FCSR.Enables.I, 2)

FTINTRP.W.D:
    FR[fd] = FP64convertToSint32(FR[fj], FCSR.Enables.I, 2)

FTINTRP.L.S:
    FR[fd][31:0] = FP32convertToSint64(FR[fj][31:0], FCSR.Enables.I, 2)

FTINTRP.L.D:
    FR[fd] = FP64convertToSint64(FR[fj], FCSR.Enables.I, 2)
```

`FTINTRZ.{W/L}.{S/D}` instruction selects the single-degree/double-precision floating-point number in floating-point register `fj`, converts it to integer/long-integer fixed-point number, and writes the obtained integer/long-integer fixed-point number to floating-point register `fd`, using the "rounding to zero" method.

```
FTINTRZ.W.S:
    FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], FCSR.Enables.I, 1)

FTINTRZ.W.D:
    FR[fd] = FP64convertToSint32(FR[fj], FCSR.Enables.I, 1)

FTINTRZ.L.S:
    FR[fd][31:0] = FP32convertToSint64(FR[fj][31:0], FCSR.Enables.I, 1)

FTINTRZ.L.D:
    FR[fd] = FP64convertToSint64(FR[fj], FCSR.Enables.I, 1)
```

`FTINTRNE.{W/L}{S/D}` instruction selects the single-precision/double-precision floating-point number in floating-point register `fj`, converts it to integer long integer fixed point number, and writes the obtained integer/long-integer fixed point number to floating-point register `fd`, using the "rounding to the nearest even number" method.

```
FTINTRNE.W.S:
    FR[fd][31:0] = FP32convertToSint32(FR[fj][31:0], FCSR.Enables.I, 0)

FTINTRNE.W.D:
    FR[fd] = FP64convertToSint32(FR[fj], FCSR.Enables.I, 0)

FTINTRNE.L.S:
    FR[fd][31:0] = FP32convertToSint64(FR[fj][31:0], FCSR.Enables.I, 0)

FTINTRNE.L.D:
    FR[fd] = FP64convertToSint64(FR[fj], FCSR.Enables.I, 0)
```

The operations in the IEEE 754-2008 standard that the above four floating-point format conversion operations follow are shown in the following table.

Table 11. Standard for floating-point conversion

Instruction name	Whether to report floating-point imprecision exceptions	IEEE 754-2008 Function
`FTINTRNE.{W/L}.{S/D}`	Yes	`convertToIntegerExactTiesToEven(x)`
`FTINTRZ.{W/L}.{S/D}`		`convertToIntegerExactTowardZero(x)`
`FTINTRP.{W/L}.{S/D}`		`convertToIntegerExactTowardPositive(x)`
`FTINTRM.{W/L}{S/D}`		`convertToIntegerExactTowardNegative(x)`
`FTINTRNE.{W/L}.{S/D}`	No	`convertToIntegerTiesToEven(x)`
`FTINTRZ.{W/L}.{S/D}`		`convertToIntegerTowardZero(x)`
`FTINTRP{W/L}.{S/D}`		`convertToIntegerTowardPositive(x)`
`FTINTRM.{W/L}.{S/D}`		`convertToIntegerTowardNegative(x)`

3.2.3.4. `FRINT.{S/D}`

Instruction formats:

```
frint.s    fd, fj
frint.d    fd, fj
```

The `FRINT.{S/D}` instruction selects the single-precision/double-precision floating-point number in the floating-point register `fj` and converts it to a single-precision/double-precision floating-point number with integer value, and the resulting single-precision/double-precision floating-point number is written to the floating-point register `fd`. According to the different states in `FCSR`, this floating-point format conversion operation follows the operation in IEEE 7542008 standard as shown in the following table.

Table 12. Standard for rounding to integer

Rounding mode	Whether to report floating-point imprecision exceptions	IEEE 754-2008 Function
Round to the nearest even number	Yes	`roundToIntegralExact(x)`
Round towards zero		
Round towards positive infinity		
Round towards negative infinity		
Round to the nearest even number	No	`roundToIntegerTiesToEven(x)`
Round towards zero		`roundToIntegerTowardZero(x)`
Round towards positive infinity		`roundToIntegerTowardPositive(x)`
Round towards negative infinity		`roundToInteger TowardNegative(x)`

```
FRINT.S:
    FR[fd][31:0] = FP32_roundToInteger(FR[fj], FCSR.Enables.I, FCSR.RM)

FRINT.D:
    FR[fd] = FP64_roundToInteger(FR[fj], FCSR.Enables.I, FCSR.RM)
```

3.2.4. Floating-Point Move Instructions

3.2.4.1. `FMOV.{S/D}`

Instruction formats:

```
fmov.s      fd, fj
fmov.d      fd, fj
```

`FMOV{S/D}` writes the value of the floating-point register `fj` into the floating-point register `fd` in the single-precision/double-precision floating-point number format. If the value of `fj` is not in the single-precision/double-precision floating-point number format, the result is uncertain.

```
FMOV.S:
    FR[fd][31:0] = FR[fj][31:0]

FMOV.D:
    FR[fd] = FR[fj]
```

The above instruction operations are non-arithmetic and will not cause IEEE 754 exceptions, nor will they modify the Cause and Flags fields of the floating-point control and status register.

3.2.4.2. FSEL

Instruction formats:

```
fsel        fd, fj, fk, ca
```

The FSEL instruction performs conditional assignment operations.

When FSEL is executed, if the value of the condition flag register ca is equal to 0, the value of the floating-point register fj is written into the floating-point register fd, otherwise the value of the floating-point register fk is written into the floating-point register fd.

```
FSEL:
    FR[fd] = CFR[ca] ? FR[fk] : FR[fj]
```

3.2.4.3. MOVGR2FR.{W/D}, MOVGR2FRH.W

Instruction formats:

```
movgr2fr.w      fd, rj
movgr2fr.d      fd, rj
movgr2frh.w     fd, rj
```

MOVGR2FR.W writes the low 32-bit value of the general register rj into the low 32-bit of the floating-point register fd. If the length of the floating-point register is 64 bits, the high 32-bit value of fd is uncertain.

```
MOVGR2FR.W:
    FR[fd][31:0] = GR[rj][31:0]
```

MOVGR2FRH.W writes the low 32-bit value of the general register rj into the high 32-bit of the floating-point register fd, and the low 32-bit value of the floating-point register fd remains unchanged.

```
MOVGR2FRH.W:
    FR[fd][63:32] = GR[rj][31:0]
    FR[fd][31: 0] = FR[fd][31:0]
```

MOVGR2FR.D writes the 64-bit value of general register rj into floating-point register fd.

```
MOVGR2FR.D:
    FR[fd] = GR[rj]
```

3.2.4.4. MOVFR2GR.{S/D}, MOVFRH2GR.S

Instruction formats:

```
movfr2gr.s      rd, fj
movfr2gr.d      rd, fj
movfrh2gr.s     rd, fj
```

MOVFR2GRMOVFRH2GR.S sign extensions the low/high 32-bit value of the floating-point register fj and writes it into the general register rd.

```
MOVFR2GR.S:
    GR[rd] = SignExtend(FR[fj][31: 0], GRLEN)

MOVFRH2GR.S:
    GR[rd] = SignExtend(FR[fj][63:32], GRLEN)
```

MOVFR2GR.D writes the 64-bit value of the floating-point register fj into the general register rd.

```
MOVFR2GR.D:
    GR[rd] = FR[fj]
```

3.2.4.5. MOVGR2FCSR, MOVFCSR2GR

Instruction formats:

```
movgr2fcsr      fcsr, rj
movfcsr2gr      rd,   fcsr
```

MOVGR2FCSR modifies the value of the software writable field corresponding to the floating-point control and status register indicated by fcsr according to the value of the lower 32 bits of the general register rj. If the MOVGR2FCSR instruction modifies FCSR0 so that the bits of the Cause field and the corresponding Enables bit are both 1, or modify the Enables field of FCSR1 and the Cause field of FCSR2 so that the Cause bit and the corresponding Enables bit are both 1, the M0VGR2FCSR instruction itself No floating-point exception will be triggered.

```
MOVGR2FCSR:
    FCSR[fcsr] = GR[rd][31:0]
```

MOVFCSR2GR sign extensions the 32-bit value of the floating-point control and status register indicated by fcsr and writes it into the general register rd.

```
MOVFCSR2GR:
    GR[rd] = SignExtend(FCSR[fcsr], GRLEN)
```

If the floating-point control and status register indicated by fcsr in the above instruction does not exist, the result is uncertain.

3.2.4.6. `MOVFR2CF`, `MOVCF2FR`

Instruction formats:

```
movfr2cf        cd, fj
movcf2fr        fd, cj
```

`MOVFR2CF` writes the value of the lowest bit of the floating-point register `fj` into the condition flag register `cd`.

```
MOVFR2CF:
    CFR[cd] = FR[fj][0]
```

`MOVCF2FR` writes the value of the condition flag register `cj` into the lowest bit of the floating-point register `fd`.

```
MOVCF2FR:
    FR[fd][0] = CFR[cj]
```

3.2.4.7. `MOVGR2CF`, `MOVCF2GR`

Instruction formats:

```
movgr2cf    cd, rj
movcf2gr    rd, cj
```

`MOVGR2CF` writes the value of the lowest bit of the general register `rj` into the condition flag register `cd`.

```
MOVGR2CF:
    CFR[cd] = GR[rj][0]
```

`MOVCF2GR` writes the value of the condition flag register `cj` into the lowest bit of the general register `rd` and clears the other bits.

```
MOVCF2GR:
    GR[rd][0] = CFR[cj]
```

3.2.5. Floating-Point Branch Instructions

3.2.5.1. `BCEQZ, BCNEZ`

Instruction formats:

```
bceqz   cj, offs21
bcnez   cj, offs21
```

`BCEQZ` judges the value of the condition flag register `cj`, if it is equal to 0, jump to the target address, otherwise it does not jump. `BCNEZ` judges the value of the condition flag register `cj`, if it is not equal to 0, jump to the target address, otherwise it does not jump. The jump target address of the above two branch instructions is to logically shift the 21-bit immediate offs21 in the instruction code to the left by 2 bits and then sign extension, and the resulting offset value plus the `PC` of the branch instruction.

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

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

TIP When writing assembly, you need to fill in the immediate field with the **real offset value** in bytes, i.e. `(offs21<<2)`.

3.2.6. Floating-Point Common Memory Access Instructions

3.2.6.1. `FLD.{S/D}, FST.{S/D}`

Instruction formats:

```
flds    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 the internal memory and writes it into the lower 32 bits of the floating-point register `fd`. If the length of the floating-point register is 64 bits, the high 32-bit value of fd is uncertain.

`FLD.D` retrieves a double word from the internal memory and writes it into the floating-point register `fd`.

`FST.S` writes the low 32-bit word data in the floating-point register `fd` into the memory.

`FST.D` writes double-word data in the floating-point register `fd` into the memory.

The access address of the above instruction is calculated by summing the value in the general register `rj` with the symbolically expanded 12-bit immediate number `si12`.

`FLD.{S/D}` and `FST.{S/D}` instructions, regardless of the hardware implementation and environment configuration, as long as the access address is naturally aligned, the non-alignment exception will not be triggered; when the access address is not naturally aligned, if the hardware implementation supports non-aligned access and the current computing environment is configured to allow non-aligned access, then the non-alignment exception will not be triggered; otherwise, the non-alignment exception will be triggered. Otherwise, the non-alignment exception will be triggered.

```
FLD.S:
    vaddr = GR[rj] + SignExtend(si12, GRLEN)
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    word = MemoryLoad(paddr, WORD)
    FR[fd][31:0] = word

FLD.D:
    vaddr = GR[rj] + SignExtend(si12, GRLEN)
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    doubleword = MemoryLoad(paddr, DOUBLEWORD)
    FR[fd] = doubleword

FST.S:
    vaddr = GR[rj] + SignExtend(si12, GRLEN)
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    MemoryStore(FR[fd][31:0], paddr, WORD)

FST.D:
    vaddr = GR[rj] + SignExtend(si12, GRLEN)
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    MemoryStore(FR[fd][63:0], paddr, DOUBLEWORD)
```

3.2.6.2. `FLDX.{S/D}, FSTX.{S/D}`

Instruction formats:

```
fldx.s  fd, rj, rk
fldx.d  fd, rj, rk
fstx.s  fd, rj, rk
fstx.d  fd, rj, rk
```

FLDX.S retrieves a word of data from the memory and writes it into the lower 32 bits of the floating-point register fd. If the length of the floating-point register is 64 bits, the high 32-bit value of fd is uncertain.

FLDX.D retrieves a double word of data from the memory and writes it into the floating-point register fd.

FSTX.S writes the low 32-bit word data in the floating-point register fd into the memory.

FSTX.D writes the double word data in the floating-point register fd into the memory.

The memory access address calculation method of the above instruction is to add sum the value in the general register rj and the value in the general register rk.

For FLDX.{S/D} and FSTX.{S/D} instructions, no matter what kind of hardware implementation and environmental configuration, as long as the memory access address is naturally aligned, the non-aligned exception will not be triggered; When the memory address is not naturally aligned, if the hardware implementation supports unaligned memory access and the current computing environment is configured to allow unaligned memory access, then the unaligned exception will not be triggered, otherwise it will trigger the unaligned exception.

```
FLDX.S:
    vaddr = GR[rj] + GR[rk]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    word = MemoryLoad(paddr, WORD)
    FR[fd][31:0] = word

FLDX.D:
    vaddr = GR[rj] + GR[rk]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    doubleword = MemoryLoad(paddr, DOUBLEWORD)
    FR[fd] = doubleword

FSTX.S:
    vaddr = GR[rj] + GR[rk]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    MemoryStore(FR[fd][31:0], paddr, WORD)

FSTX.D:
    vaddr = GR[rj] + GR[rk]
    AddressCompli anceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    MemoryStore(FR[fd][63:0], paddr, DOUBLEWORD)
```

3.2.7. Floating-Point Bound Check Memory Access Instructions

3.2.7.1. FLD{GT/LE}.{S/D}, FST{GT/LE}.{S/D}

Instruction formats:

```
fldgt.s     fd, rj, rk
fldgt.d     fd, rj, rk
fldle.s     fd, rj, rk
fldle.d     fd, rj, rk
fstgt.s     fd, rj, rk
fstgt.d     fd, rj, rk
fstle.s     fd, rj, rk
fstle.d     fd, rj, rk
```

`FLD{GT/LE}.{S/D}` determines if the valid address is out of bounds and writes the value from memory to the floating-point register.

`FLD{GT/LE}.S` checks if the value in general register `rj` is greater/less than/equal to the value in general register `rk`, and if the condition is met, fetches a word of data from memory and writes it to the lower 32 bits of floating-point register `fd`. If the floating-point register is 64 bits wide, the high 32-bit value of fd is not determined.

`FLD{GT/LE}.D` checks if the value in general register `rj` is greater than/less than/equal to the value in general register `rk`, and if the condition is met, fetches a double word of data from memory and writes it to floating-point register `fd`.

`FST{GT/LE}.{S/D}` determines if the valid address is out of bounds, and writes the value of the floating-point register to memory.

`FST{GT/LE}.S` checks if the value in general register `rj` is greater/less than/equal to the value in general register `rk`, and if the condition is met, writes the low 32-bit word data in floating-point register `fd` to memory.

`FST{GT/LE}.D` checks if the value in general register `rj` is greater than/less than or equal to the value in general register `rk`, and if the condition is satisfied, writes the double word data in floating-point register `fd` to memory.

The access address of the above instruction comes directly from the value in general register `rj`. The access addresses of the above instructions are required to be naturally aligned, otherwise a non-alignment exception will be triggered. The above instruction terminates the access operation and triggers the bound check exception if the check condition is not satisfied.

```
FLDGT.S:
    vaddr = GR[rj]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    if GR[rj] > GR[rk]:
        word = MemoryLoad(paddr, WORD)
        FR[fd][31:0] = word
    else:
        RaiseException(BCE)    # Bound Check Exception

FLDGT.D:
    vaddr = GR [rj]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    if GR[rj] > GR[rk]:
        FR[fd] = MemoryLoad(paddr, DOUBLEWORD)
    else:
        RaiseException(BCE)    # Bound Check Exception

FLDLE.S:
    vaddr = GR[rj]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    if GR[rj] <= GR[rk]:
        word = MemoryLoad(paddr, WORD)
        FR[fd][31:0] = word
    else:
        RaiseException(BCE)    # Bound Check Exception

FLDLE.D:
    vaddr = GR[rj]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    if GR[rj] <= GR[rk]:
        FR[fd] = MemoryLoad(paddr, DOUBLEWORD)
    else:
        RaiseException(BCE)    # Bound Check Exception

FSTGT.S:
    vaddr = GR[rj]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    if GR[rj] > GR[rk]:
        MemoryStore(FR[fd][31:0], paddr, WORD)
    else:
        RaiseException(BCE)    # Bound Check Exception

FSTGT.D:
    vaddr = GR[rij]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    if GR[rj] > GR[rk]:
        MemoryStore(FR[fd][63:0], paddr, DOUBLEWORD)
    else:
        RaiseException(BCE)    # Bound Check Exception

FSTLE.S:
    vaddr = GR[rj]
    AddressComplianceCheck(vaddr)
    paddr = AddressTranslation(vaddr)
    if GR[rj] <= GR[rk]:
        MemoryStore(FR[fd][31:0], paddr, WORD)
    else:
        RaiseException(BCE)    # Bound Check Exception

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

Chapter 4. Overview of Privileged Resources

4.1. Privilege Levels

The processor cores are divided into four privilege levels (PLV0 to PLV3), which are uniquely determined by the value of the `PLV` field in `CSR.CRMD`.

Among all privilege levels, PLV0 is the privilege level with the highest privilege and is the only privilege level that can use privileged instructions and access all privileged resources. The three privilege levels, PLV1 to PLV3, cannot execute privileged instructions to access privileged resources, but the three privilege levels have different access rights under the MMU's mapped address translation mode.

For Linux systems, only the PLV0 level can correspond to the kernel state in the architecture, while the PLV3 level is recommended for the user state.

4.2. Overview of Privilege Instructions

All privileged instructions are accessible only at the PLV0 privilege level. The only exception is that when the `RPERF1`/`RPERF2`/`RPERF3` in `CSR.MISC` is configured to `1`, the `CSRRD` instruction can be executed at PLV1/PLV2/PLV3 privilege level to read the performance counter.

4.2.1. CSR Access Instructions

Instruction formats:

```
csrrd       rd, csr_num
csrwr       rd, csr_num
csrxchg     rd, rj, csr_num
```

The `CSRRD`, `CSRWR`, and `CSRXCHG` instructions are used to access the CSRs in software. The `CSRRD` instruction writes the value of the specified CSR to the general register rd. The `CSRWR` instruction writes the old value of the general register `rd` to the specified CSR and updates the old value of the specified CSR to the general register `rd`. The `CSRXCHG` instruction writes the old value of the general register `rd` to the bits of the specified CSR corresponding to the write mask `1` according to the write mask information stored in the general register `rj`. The `CSRXCHG` instruction writes the old value of the general register `rd` to the bits of the specified CSR corresponding to the write mask of `1` according to the write mask information stored in the general register `rj`. The rest of the bits in the CSR remain unchanged, and the old value of the CSR is updated to the general register `rd`.

All CSRs are addressed independently. The addressable value of the CSRs in the above instruction is derived from the 14-bit immediate `csr_num` in the instruction. `csr_num` for CSR 0 is `0`, `csr_num` for CSR 1 is `1`, and so on.

The length of all CSR registers is either `32` bits or equal to the length of `GR` in the architecture, so CSR access instructions do not distinguish between lengths. In LA32, all CSRs are naturally `32` bits wide. In LA64, CSRs with a fixed 32-bit length in the definition are always written to the general purpose register `rd` after symbolic expansion.

When a CSR access instruction accesses a CSR that is not defined in the architecture or not implemented in hardware, the read operation returns an all `0` values and the write operation does not modify any software-visible state of the processor.

4.2.2. IOCSR Access Instructions

4.2.2.1. IOCSR{RD/WR}.{B/H/W/D}

Instruction formats:

```
iocsrrd.b       rd, rj
iocsrrd.h       rd, rj
iocsrrd.w       rd, rj
iocsrrd.d       rd, rj
iocsrwr.b       rd, rj
iocsrwr.h       rd, rj
iocsrwr.w       rd, rj
iocsrwr.d       rd, rj
```

`IOCSR{RD/WR}.{B/H/W/D}` instructions are used to access the IOCSR.

All IOCSR registers use independent addressing space, and the basic unit of addressing is byte. All data is stored in the IOCSR space in a little-endian storing `{B/H/W/D}` instruction's IOCSR address is from the general register `rj`.

The `IOCSRRD.{B/H/W/D}` instruction fetches byte/half-word/word/double-word length data from the specified address in the IOCSR space, and writes it to the general register `rd` after symbolic expansion.

The `IOCSRWR.{B/H/W/D}` instruction writes the `[7:0]`/`[15:0]`/`[31:0]`/`[63:0]` bits of data in the general register `rd` to the beginning of the specified address in the IOCSR space.

The `IOCSRRD.D` and `IOCSRWR.D` instructions appear only in LA64.

IOCSR registers can typically be accessed by multiple processor cores simultaneously. The execution of IOCSR access instructions on multiple processor cores satisfies the sequential consistency condition.

4.2.3. Cache Maintenance Instructions

4.2.3.1. CACOP

Instruction formats:

```
cacop   op, rj, si12
```

The `CACOP` instruction is mainly used for Cache initialization and cache-consistency maintenance.

The value of the general register `rj`, plus the sign-extended 12-bit immediate number `si12`, gives the virtual address `VA` used by the `CACOP` instruction, which is used to indicate the location of the Cache line being operated on.

Which Cache is accessed by the `CACOP` instruction and what Cache operation is performed is determined by the 5-bit op in the instruction. `op[2:0]` indicates the Cache object to be operated on, and `op[4:3]` indicates the type of operation.

The Cache object indicated by op[2:0] is in the same order as the Cache identified in CPUCFG10. For example, when CPUCFG10=0x02C3D, op[2:0]=0 indicates operation of the first-level private instruction Cache, op[2:0]=1 indicates operation of the first-level private data Cache, op[2:0]=2 indicates operation of the second-level private mixed Cache, and op[2:0]=3 indicates operation of the third-level shared mixed Cache.

op[4:3]=0 is used for Cache initialization (StoreTag), mainly to write the contents of the CSR.CTAG to the tag of the specified Cache row using direct address indexing. Suppose the Cache to be accessed has (1<<Way) ways, each ways has (1<<Index) Cache line, and each Cache line size is (1<<Offset) bytes, then the direct address indexing method means that the VA[Index+offset1.Offset] of the VA[Way-1:0] way of the Cache is [operated: Offset] line of the Cache.

op[4:3]=1 means that the cache-consistency (Index Invalidate / Invalidate and Writeback) is maintained by direct address indexing. See the previous paragraph for a definition of the direct address indexing method. The operation to maintain consistency is an invalidate and writeback operation on the specified Cache. If the operation is on the instruction Cache, then only the invalidation operation is performed, not the writing back of the data in the Cache row. The data written back into which level of memory is determined by the specific implementation of the Cache hierarchy and the inclusion or mutually exclusive relationship between the levels. For data Cache or mixed Cache, it is up to the implementation to decide whether to write back the data in the Cacche row only if it is dirty.

op[4:3]=2 means that Cache coherency is maintained by query indexing (Hit Invalidate / Invalidate and Writeback).

The operation of maintaining Cache coherency here is the same as described in the above paragraph. The so-called query index approach treats the VA of the CACOP instruction as a normal load instruction to access the Cache to be operated on, and if it hits, it operates on the hit Cache row, otherwise it does not do any operation. Since this query process may involve virtual-to-real address translation, the CACOP instruction may trigger TLB-related exceptions in this case. However, since the CACOP instruction operates on Cache rows, there is no need to consider address alignment or not in this case.

op[4:3]=3 is an implementation of a custom Cache operation and is not explicitly functionally defined in the architecture specification.

4.2.4. TLB Maintenance Instructions

4.2.4.1. TLBSRCH

Instruction formats:

```
tlbsrch
```

The functional definition of the TLBSRCH instruction without implementing the LVZ extension is given here.

Use the information of CSR.ASID and CSR.TLBEHI to query TLB. If there is a hit entry, the index of the hit entry is written to the Index field of CSR.TLBIDX, and the INV field of CSR.TLBIDX is set to 0; if there is no hit entry, the INV field of CSR.TLBIDX is set to 1.

The rules for calculating the index of each entry in the TLB are, starting from 0, incremental numbering, first STLB and then MTLB, STLB from the 0th line to the last line of the 0th way, then the 0th line to the last line of the 1st way, until the last line of the last way, MTLB from the 0th line to the last line.

4.2.4.2. TLBRD

Instruction formats:

```
tlbrd
```

The functional definition of the TLBRD instruction without implementing the LVZ extension is given here.

The value of the Index field of CSR.TLBIDX is used as the index to read the specified entry in the TLB. If the specified location is a valid TLB entry, the page table information of the TLB entry is written to CSR.TLBEHI, CSR.TLBELO0, CSR.TLBELO1 and CSR.TLBIDX.PS, and the INV field of CSR.TLBIDX is set to 0; if the specified location is an invalid TLB entry, then CSR.TLBEHI, CSR.TLBELO0 and CSR.TLBELO1 is set to 0; and the INV field of CSR.TLBIDX is set to 1; TLBIDX.PS is set to 0 and the INV field of CSR.TLBIDX is set to 1.

Note that valid/invalid TLB entries and valid/invalid page table entries in the TLB are two concepts.

If the index used for the access exceeds the range of the TLB, the behavior of the processor is undefinded.

4.2.4.3. TLBWR

Instruction formats:

```
tlbwr
```

The functional definition of the TLBWR instruction without implementing the LVZ extension is given here.

The TLBWR instruction fills the page table entry information stored in the TLB-related CSRs into the TLB. The page table entry information to be populated comes from CSR.TLBEHI, CSR.TLBELO0, CSR.TLBELO1 and CSR.TLBIDX_PS. If CSR.TLBIDX.NE=1, then the TLB is populated with an invalid TLB entry; only if CSR.TLBIDX.NE=0, the TLB is populated with a valid TLB entry.

The location where the page table entry is written to the TLB is specified by the value of the Index field of CSR.TLBIDX. Please refer to the TLBSRCH instruction for the calculation rules of each index in the TLB for the specific corresponding rules. If a page table entry is to be written to the STLB, but a conflict occurs between the value of the Index field of CSR.TLBIDX and VPPN and CSR.TLBIDX.PS in CSR.TLBEHI, the behavior of the processor is undefinded.

4.2.4.4. TLBFILL

The functional definition of the TLBFILL instruction without implementing the LVZ extension is given here.

The TLBFILL instruction fills the page table entry information stored in the TLB-related CSRs into the TLB. The page table entry information to be populated comes from CSR.TLBEHI, CSR.TLBELO0, CSR.TLBELO1 and CSR.TLBIDX_PS. If CSR.TLBIDX.NE=1, then the TLB is populated with an invalid TLB entry; only if CSR.TLBIDX.NE=0, the TLB is populated with a valid TLB entry.

Whether to write to STLB or MTLB is first made based on the page size of the page table entry being filled. When the page size of the page table entry being filled is equal to the page size configured for STLB (CSR.STLBPS) it will be filled to STLB, otherwise it will be filled to MTLB. Which way the page table entry is filled to STLB, or which entry is filled to MTLB is randomly selected by the hardware.

4.2.4.5. TLBCLR

Instruction formats:

```
tlbclr
```

The contents of the TLB are invalidated according to the information of the TLB-related CSRs to maintain the consistency of the page table data between the TLB and the memory. The functional definition of the `TLBCLR` instruction without implementing the LVZ extension is given here.

When `CSR.index.index` falls within the range of MTLB (greater than or equal to the number of STLB entries), `TLBCLR` is executed to invalidate all page table entries in MTLB with `G=0` and `ASID` equal to `CSR.ASID.ASID`.

When `CSR.index.index` falls within the STLB range (less than the number of STLB entries), execute a `TLBCLR` to invalidate all page table entries in the STLB that are equal to `G=0` and `ASID` equal to `CSR.ASID.ASID` in the group indicated by the low bit of `CSR.index.index`.

4.2.4.6. TLBFLUSH

Instruction formats:

```
tlbflush
```

The contents of the TLB are invalidated according to the information of the TLB-related CSRs to maintain the consistency of the page table data between the TLB and the memory. The functional definition of `TLBCLR` instruction without implementing LVZ extension is given here.

When `CSR.index.index` falls within the MTLB range (greater than or equal to the number of STLB entries), `TLBCLR` is executed to invalidate all page table entries in the MTLB.

When `CSR.index.index` falls within the STLB range (less than the number of STLB entries), a `TLBCLR` is executed to invalidate all page table entries in the group indicated by the low `CSR.index.index` in the STLB.

4.2.4.7. INVTLB

Instruction formats:

```
invtlb  op, rj, rk
```

The `INVTLB` instruction is used to invalidate the contents of the TLB to maintain consistency of the page table data between the TLB and memory. The functional definition of the `INVTLB` instruction is given here for the case where the LVZ extension is not implemented.

Of the three source operands of the instruction, op is a 5-bit immediate number to indicate the type of operation.

The `[9:0]` bits of the general register `rj` hold the `ASID` information required for the invalid operation (called “register specified `ASID`”), and the remaining bits must be filled with `0`. When the operation indicated by op does not require an ASID, the general register `rj` should be set to `r0`.

The general register `rk` is used to store the virtual address information required for invalid operations (called “register specified `VA`”). When the operation indicated by the op does not require virtual address information, the general register `rk` should be set to `r0`.

The operations corresponding to each op are shown in the following table, and the op that does not appear in the table will trigger a reserved instruction exception.

Table 13. Operations corresponding to each `op` in the `INVTLB` instruction

op	Operation
`0x0`	Clear all page table entries
`0x1`	Clears all page table entries. The effect of this operation is exactly the same as op=0.
`0x2`	Clears all `G=1` page table entries.
`0x3`	Clears all page table entries with `G=0`.
`0x4`	Clears all page table entries with `G=0` and `ASID` equal to the `ASID` specified in the register.
`0x5`	Clear all page table entries with `G=0` and `ASID` equal to the register specified `ASID`, and VA equal to the register specified `VA`.
`0x6`	Clear all page table entries where `G=1` or `ASID` is equal to the `ASID` specified in the register and `VA` is equal to the `VA` specified in the register.

4.2.5. Software page walking Instructions

4.2.5.1. `LDDIR`

Instruction formats:

```
lddir   rd, rj, level
```

The `LDDIR` instruction is used for accessing directory entries during software page table walking.

The 5-bit immediate level in the `LDDIR` instruction indicates which page table is currently being accessed. `level=1` corresponds to `Dir0` in `PWCL`, `level=2` corresponds to `Dirl` in `PWCL`, `level=3` corresponds to `Dir2` in `PWCH`, and `level=4` corresponds to `Dir3` in `PWCH`.

If bit `[6]` of the general register `rj` is `0`, it means that the content of `rj` is the physical address of the base address of the level page table at this time. In this case, the `LDDIR` instruction will access the level page table according to the current TLB refill address, retrieve the base address of the corresponding `level+1` page table, and write it to the general register `rd`.

If bit `[6]` of general register `rj` is `1`, it means that the content in `rj` is a large page (Huge Page) page table entry. In this case, after executing the `LDDRI` instruction, the value in the general register `rj` will be written directly to the general register `rd`.

4.2.5.2. `LDPTE`

Instruction formats:

```
ldpte   rj, seq
```

The LDPTE instruction is used for page table entry accesses during software page table walking.

The immediate number seq in the LDPTE instruction is used to indicate whether an even or odd number of pages are being accessed. The result is written to CSR.TLBRELO0 when an even page is accessed. The result will be written to CSR.TLBRELO1 when an odd page is accessed.

If bit [6] of the general register rj is 0, the content of rj is the physical address of the base address of the page table at that level of the PTE. In this case, the LDPTE instruction will access the PTE level page table according to the currently processed TLB refill address, retrieve the page table entry and write it to the corresponding CSR.

If bit [6] of the general register rj is 1, it means that the content of rj is a large page (Huge Page) page table entry. In this case, the LDPTE instruction is executed, and the value in general register rj is directly converted into the final page table entry format and written to the corresponding CSR.

4.2.6. Other Miscellaneous Instructions

4.2.6.1. ERTN

Instruction formats:

```
ertn
```

The ERTN instruction is used to return from exception processing.

If the exception being processed is a debug exception clear the DM bit in the CSR.DEBUG to 0, and jump to the address stored in the CSR.DEBUG to start fetching.

If the exception being processed is something other than a debug exception, update the PPLV, PIE, and PWE information corresponding to the exception to CSR.CRMD, update the PVM in CSR.VMCTL to CSR.VMCTL.VM, and jump to the ERA corresponding to the exception to start fetching instructions.

If the exception processed is an error-related exception, the PPLV, PIE and PWE information corresponding to the exception is from CSR.MERRCTL, and the ERA corresponding to the exception is from CSR.MERRERA. In addition, the PDA, PPG, PDCAF and PDCAM information in CSR.MERRCTL should be updated to CSR.CRMD.

If the exception being processed is a TLB refill exception, the PPLV, PIE, and PWE information corresponding to the exception is from CSR.TLBRPRMD, and the ERA corresponding to the exception is from CSR.TLBRERA. In addition, it is necessary to clear DA field 0 and PG field 1 in CSR.CRMD.

If the exception being handled is not a debug exception, an error-related exception, or a TLB refill exception, then the PPLV, PIE and PWE information corresponding to the exception is from CSR.PRMD, and the ERA corresponding to the exception is from CSR.ERA.

When executing the ERTN instruction, if the KL0 bit in CSR.LLBCTL is not equal to 1, then the LLbit is set to 0, otherwise the LLbit is not modified.

4.2.6.2. DBCL

Instruction formats:

```
dbcl        code
```

Executing DBCL instruction will immediately enter debug mode.

4.2.6.3. IDLE

Instruction formats:

```
idle        hint
```

After executing the IDLE instruction, the processor core will stop fetching instructions and enter the wait state until it is woken up by an interrupt or is reset. After waking up from the wait state by an interrupt, the first instruction executed by the processor core is the one after IDLE.

Chapter 5. Memory Management

5.1. Physical Address Space

The physical address space range of memory is `0`-2^{PALEN}`-1`.

In LA32, `PALEN` is theoretically a positive integer not exceeding `32`, and its specific value is determined by the implementation, which is usually recommended to be `32`.

In LA64, `PALEN` is theoretically a positive integer not exceeding `60`, and its specific value is determined by the implementation.

The system software can determine the specific value of `PALEN` by reading the `PALEN` field of the `0x1` configuration word with the `CPUCFG` instruction.

5.2. Virtual Address Space and Address Translation Mode

The virtual address space is linear/flat in LoongArch. For PLV0 level, the virtual address space size is 2^{32} bytes in LA32 and 2^{64} bytes in LA64. However, the 2^{64}-byte virtual address space is not always legal in LA64. It can be assumed that there are some virtual address holes. The legal virtual address space is closely related to the address translation mode, which is described in the next section in conjunction with the definition of the address translation mode.

The MMU in LoongArch supports two modes of translating virtual addresses to physical addresses: direct address translation mode and mapped address translation mode.

When `CSR.CRMD.DA=1` and `CSR.CRMD.PG=0`, the MMU of the processor core is in direct address translation mode. In this mode, the physical address is by default equal to the `[PALEN-1:0]` bits of the virtual address (zero extension if necessary), unless the implementation uses other higher priority translation rules. The entire virtual address space is legal at this point. The processor will enter the direct address translation mode after reset.

When `CSR.CRMD.DA=0` and `CSR.CRMD.PG=1`, the MMU of the processor core is in mapped address translation mode. Specifically, there are two types of address translation modes: direct mapped address translation mode (direct mapped mode) and page table mapped address translation mode (page table mapped mode). When translating addresses, the direct mapped mode is preferred. Only when the address cannot be translated by the direct mapped mode, the page table mapped mode is used for translation. See Direct Mapped Address Translation Mode for details on the direct mapped mode and Memory Management of Page Table Mapping for details on the page table mapped mode. The rules for virtual address space legality during using the page table mapped mode in LA64 are presented here. The `[63:PALEN]` bits of the legal virtual address must be the same as the `[PALEN-1]` bits, otherwise an **AD**dress error **E**xception (ADE) will be triggered. In direct mapped mode, however, this address illegality check is not required.

5.2.1. Direct Mapped Address Translation Mode

When the MMU of the processor core is in mapped address translation mode, direct mapping of virtual and physical addresses can also be accomplished through the mechanism of direct mapping configuration windows. There are four direct mapping configuration windows. The first two windows can be used for both fetch and load/store operations, and the last two windows are used for load/store operations only.

The system software sets each of the four direct mapping configuration windows by configuring the `CSR.DMW0`-`CSR.DMW3` configuration window registers. Each window can be used to configure not only for the address range, but also for the privilege levels under which the window is available, as well as the type of memory access for virtual address within the address range.

In LA64, each direct mapping configuration window can be configured with a virtual address space which length is PALEN bytes. When a virtual address hits a valid direct mapping configuration window, its physical address is equal to the `[PALEN-1:0]` bits of itself. The hit is determined as follows: the highest 8 bits of the virtual address (`[63:60]` bits) are equal to the `VSEG` field in the configuration window register, and the current privilege level is available.

For example, if `PALEN` is equal to `48` and `DMWO` is set to `0x9000000000000011`, virtual address space `0x9000000000000000`-`0x9000FFFFFFFFFFFF` will be directly mapped to physical address space `0x0`-`0xFFFFFFFFFFFF` at the PLV0 privilege level, the memory access type of which is consistent and cacheable.

In LA32, each direct mapping configuration window can be configured with a virtual address space which length is 2^{29} bytes. When a virtual address hits a valid direct mapping configuration window, its physical address is equal to the combination of the `[28:0]` bits of itself and the high bits of the the configuration window register. The hit is determined as follows: the highest 4 bits of the virtual address (`[31:29]` bits) are equal to the `[31:29]` bits in the configuration window register, and the current privilege level is available.

For example, if `DMW0` is set to `0x80000011`, virtual address space `0x80000000`-`0x8FFFFFFF` will be directly mapped to physical address space `0x0`-`0x1FFFFFFF` at the PLV0 privilege level, the memory access type of which is consistent and cacheable.

5.2.2. 32-bit Address Mode in LA64

When the binary application in LA32 runs on the processor that implements LA64, the calculation involving address in the instruction needs to be handled specially in order to obtain the same operation result, which is the 32-bit address mode control in LA64. When `VA32L1`/`VA32L2`/`VA32L3` in `CSR.MISC` is set to `1`, the software running at PLV1/PLV2/PLV3 level will run in 32-bit address mode. At this time, the virtual address will be zero extended to 64 bits. The 32-bit results of executing instructions like `BL`, `JIRL` and `PCADD` will also be sign extended to 64 bits.

5.2.3. Virtual Address Reduction Mode in LA64

In order to reduce the number of page table levels in some occasions, the virtual address reduction mode is also provided in LA64. When the system software set `RDVA` in the `CSR.RVACFG` register to a value from `1` to `8`, the valid bits of the virtual address in mapped address translation mode are treated as `(VALEN-RDVA)` bits. For example, when `VALEN=48` and `RDVA` is set to `8`, the `[63:40]` bits of the legal address must be a sign expansion of the `[39]` bit.

5.3. Memory Access Types

As mentioned in Memory Access Types, there are three types of memory access in LoongArch, including CC, SUC, and WUC.

When the MMU of the processor core is in direct address translation mode, the memory access types of all fetch operations are determined by `CSR.CRMD.DATF`, and the memory access types of all load/store operations are determined by `CSR.CRMD.DATM`.

When the MMU of the processor core is in mapped address translation mode, the memory access types are divided into two cases. If the address of a fetch or load/store operation falls on one of the direct mapping configuration windows, then its memory access type is determined by the MAT field in the CSR register that is configured in the window. If the fetch or load/store can only be mapped through the page table, then its memory access type is determined by the MAT field in the page table entry.

In any case, the definition of the control value for the memory access type is always the same: `0` for strongly-ordered uncached, `1` for coherent cached, `2` for weakly-ordered uncached, and `3` for reserved.

5.4. Memory Management of Page Table Mapping

In mapped address translation mode, all legal addresses, except those that fall in the direct mapping configuration window, must be mapped through the page table to complete the translation of virtual addresses to physical addresses. As a temporary Cache for the processor to store information about page tables in the operating system, TLB is used to speed up the translation of virtual addresses to physical addresses for fetch and load/store operations in mapped address translation mode.

5.4.1. TLB Organizational Structure

The TLB in LoongArch is divided into two parts, one is **S**ingular-Page-Size **TLB** (STLB) which has the same page size for all table entries, and the other is **M**ultiple page size **TLB** (MTLB) which supports different page sizes for different table entries.

The page size is the same as the page size configured in the STLB, and it is up to the implementation to decide whether a page table entry can enter the MTLB, with no restrictions in the architecture specification.

During the translation of a virtual address to a physical address, the STLB and the MTLB look up simultaneously. Accordingly, the software needs to ensure that there are no simultaneous hits of MTLB and STLB, otherwise the processor behavior will be undefined.

The MTLBs are fully associative, and the STLBs are multi-way set associative. For STLB, if it has 2^{INDEX} groups and the configured page size is 2^{PS} bytes, the hardware querying STLB is using the `[PS+INDEX:PS]` bits of the virtual address as the index of each way.

5.4.2. TLB Entry

The table entry formats of STLB and MTLB is basically the same, the only difference is that each table entry of MTLB contains the page size information, while STLB does not need to store the page size information repeatedly because it is the same page size. For STLB, the page size of the page table entry is configured by the system software in the `PS` field of the `CSR.STLBPS` register.

The format of each TLB table entry is shown in the figure and contains two parts: the comparison part and the physical translation part.

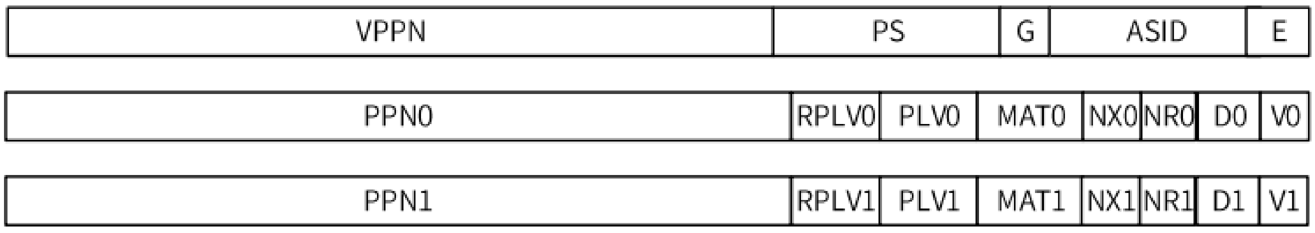

Figure 6. TLB entry formats

The comparison part of TLB table entries includes:

- **E**xistence bit (`E`), `1` bit. When this bit is set, it indicates that the page table entry exists and can participate in lookup matching.
- **A**ddress **S**pace **ID**entifier (`ASID`), `10` bits. `ASID` is used to distinguish the same virtual address in different processes and to avoid performance loss caused by clearing the entire TLB during process switching. The operating system assigns a unique `ASID` to each process, and the TLB needs to match the `ASID` in addition to the address when performing lookups.

- **G**lobal flag bit (`G`), `1` bit. When this bit is set, the lookup is not checked for ASID consistency. If the operating system needs to share the same virtual address among all processes, this bit can be set.
- **P**age **S**ize (`PS`), `6` bits. `PS` appears only in the MTLB. It is used to specify the size of the pages stored in this page table entry. The value is a power of `2` of the page size. That is, for a page size of `16KB`, `PS=14`.
- **V**irtual **P**air of **P**age frames **N**umber (`VPPN`), `(VALEN-13)` bits. The physical translation part holds the translation information for a adjacent odd even pair of page tables, so the virtual page number stored in the TLB page table entry is the content of the virtual page number divided by `2` in the operating system. The lowest bit of the virtual page number does not need to be stored. When searching for the TLB, the lowest bit of the virtual page number is used to decide whether to select the odd-numbered page or the even-numbered page for physical translation.

The physical translation part of the table entry holds the translation information for a adjacent odd even pair of page tables, and the information for each page includes:

- **V**alid bit (`V`), `1` bit. This bit is set when the page table entry is valid. Note the difference between the P bit when performing lookups. The P bit refers to whether a page table entry on the TLB table entry is present. A page table entry is present even if it is invalid (`V=0`).
- **D**irty bit (`D`), `1` bit. This bit is set when there is dirty data on the address space where the page table entry is located.
- **N**on-**R**eadable bit (`NR`), `1` bit. This bit is set when no load operation is allowed on the address space where this page table entry is located. This control bit is only exist in LA64.
- **N**on-e**X**ecutable bit (`NX`), `1` bit. This bit is set when a fetch operation is not allowed on the address space where this page table entry is located. This control bit is only exist in LA64.
- **M**emory **A**ccess **T**ype (`MAT`), `2` bits. `MAT` controls the type of memory access that falls on the address space where the page table entry is located. See Memory Access Types for the specific meaning of each value.
- **P**rivilege **L**e**V**el (`PLV`), `2` bits. `PLV` refers to the privilege level corresponding to this page table entry. When `RPLV=0`, the page table entry can be accessed by any program whose privilege level is not lower than `PLV`; when `RPLV=1`, the page table entry can only be accessed by programs whose privilege level is equal to `PLV`.
- **R**estricted **P**rivilege **L**e**V**el (`RPLV`), `1` bit. `RPLV` refers to whether a page table entry is accessed only by programs corresponding to the privilege level. See above in `PLV`. This control bit is only exist in LA64.
- **P**hysical **P**age **N**umber (`PPN`), `(PALEN-12)` bits. When the page size is larger than `4KB`, the $[\log_2 PS-1:12]$ bits of the `PPN` stored in the TLB can be any value.

5.4.3. Software Management of TLB

The management of TLBs in LoongArch involves software work. In the current version of this architecture specification, TLB refill and consistent maintenance between TLB and page tables are still all led by software.

5.4.3.1. TLB-related Exceptions

The TLB performs translation of virtual addresses to physical addresses automatically by hardware. However, when there is no match in the TLB, or when the page table entry is invalid or illegally accessed despite the match, an exception needs to be triggered and handed over to the OS kernel or other supervisory programs. The exception is further handled by software to maintain the content of the TLB or to make a final ruling on the legality of the program execution. The exceptions related to TLB management in LoongArch are as follows:

- TLB refill exception: This exception is triggered when the virtual address of an access operation does not have a match in the TLB, which notifies the system software to perform a TLB refill. This exception has a separate exception entry, a separate CSR for maintaining the exception context, and a separate set of CSRs as TLB access interface; that means the exception is allowed to be triggered during the processing of other exceptions. While the TLB refill exception being caught, `CRMD` will be set to `1` and `PG` will be set to `0`. This means the hardware will enter the direct address translation mode automatically, so that the TLB refill exception handler itself will not trigger the TLB refill exception again, and the exception context will not be saved and recovered. In order to distinguish CSRs used by the TLB refill exception and CSRs available for other exceptions, the hardware will automatically set `CSR.TLBRERA.ISTLBR` to `1` while the exception is caught.
- Page invalid exception for load operation: This exception is triggered when the virtual address of the load operation finds a match in the TLB with `V=0`.
- Page invalid exception for store operation: This exception is triggered when the virtual address of the store operation finds a match in the TLB with `V=0`.
- Page invalid exception for fetch operation: This exception is triggered when the virtual address of the fetch operation finds a match in the TLB with `V=0`.
- Page privilege level ilegal exception: This exception is triggered when the virtual address of the access operation finds a matching entry in the TLB with `V=1`, but the privilege level of the access is illegal. The privilege level is illegal when `RPLV=0` and `CSR.CRMD.PLV` is greater than the `PLV` in the page table entry, or when `RPLV=1` and `CSR.CRMD.PLV` is not equal to the `PLV` in the page table entry.
- Page modify exception: This exception is triggered when the virtual address of the store operation finds a match in the TLB with `V=1` and privilege level is legal and `D=0`.
- Page non-readable exception: This exception is triggered when the virtual address of the load operation finds a match in the TLB with `V=1` and privilege level is legal and `NR=1`.
- Page non-executable exception: This exception is triggered when the virtual address of the fetch operation finds a match in the TLB with `V=1` and privilege level is legal and `NX=1`.

5.4.3.2. TLB-related Instructions

The TLB-related instructions mainly involve operations such as lookup, read, write, and invalidate the TLB for filling, updating, and consistency maintenance of the TLB. See TLB Maintenance Instructions and Software page walking Instructions for specific instruction definitions.

5.4.3.3. TLB-related CSRs

TLB-related CSRs are divided into three categories according to their functions. The first category is used for the interactive interface of TLBs other than TLB refill exceptions. The second category is used for software and hardware page walking. The third category is used for TLB refill exceptions.

The first category includes:

- `BADV`
- `TLBEHI`
- `TLBELO0`
- `TLBELO1`
- `TLBIDX`
- `ASID`
- `STLBPS`

The second category includes:

- `PGDL`
- `PGDH`
- `PGD`
- `PWCL`
- `PWCH`

The third category includes:

- `TLBRENTRY`
- `TLBRERA`
- `TLBRBADV`
- `TLBREHI`
- `TLBRELO0`
- `TLBRELO1`
- `TLBRPRMD`
- `TLBRSAVE`

See Basic Control and Status Registers for details of how each CSR register above interacts with the TLB.

5.4.3.4. Initialization of TLB

LoongArch allows not to implement the hardware initialization of the TLB, but to let the software in the boot phase perform this function by executing `INVTLB r0, r0`.

5.4.4. TLB-based Translation of Virtual Addresses to Physical Addresses

The TLB-based translation of virtual addresses to physical addresses is described here. For the convenience of description, the following is presented in pseudocode form with STLB first and MTLB second, while the hardware implementation of the processor can look up STLB and MTLB at the same time.

```
# va: virtual address to be found.
# mem_type: memory acess type. FETCH refers to fetch operation, LOAD
refers to load operation, and STORE refers to store operation.
# plv: current privilege level, i.e., CSR.CRMD.PLV.
# pa: physical addresses after translation.
# mat: memory acess type after translation.
# VALEN: number of valid bits of the virtual address.
# PALEN: number of valid bits of the physical address.
# STLB[][]: STLB[N][M] refers to the Nth way and the Mth entry of STLB.
# STLB_WAY: number of ways of STLB.
# STLB_INDEX: the power of 2 of the number of groups in each way of
STLB, i.e., each way has 2STLB_INDEX groups.
# MTLB[]: MTLB[N] refers to the Nth entry of MTLB.
# MTLB_ENTRIES: number of entries of MTLB.

# look up STLB
stlb_found = 0
stlb_ps = CSR.STLBPS.PS
stlb_idx = va[stlb_ps+STLB_INDEX-1:stlb_ps]
for way in range(STLB_WAY):
    if (STLB[way][stlb_idx].E == 1) and
    ((STLB[way][stlb_idx].G == 1) or (STLB[way][stlb_idx].ASID ==
CSR.ASID.ASID))
 and
    (STLB[way][stlb_idx].VPPN[VALEN-1:stlb_ps+1]==va[VALEN-
1:stlb_ps+1]):
    if (stlb_found == 0):
        stlb_found = 1
        if (va[stlb_s] == 0):
            sfound_v = STLB[way][stlb_idx].V0
            sfound_d = STLB[way][stlb_idx].D0
            sfoundnr = STLB[way][stlb_idx].NR0
            sfound_ne = STLB[way][stlb_idx].NE0
            sfound_mat = STLB[way][stlb_idx].MAT0
            sfound_plv = STLB[way][stlb_idx].PLV0
            sfound_rplv = STLB[way][stlb_idx].RPLV0
            sfound_pfn = STLB[way][stlb_idx].PFN0
        else:
            sfound_v = STLB[way][stlb_idx].V1
            sfound_d = STLB[way][stlb_idx].D1
            sfound_nr = STLB[way][stlb_idx].NR1
            sfound_ne = STLB[way][stlb_idx].NE1
            sfound_mat = STLB[way][stlb_idx].MAT1
            sfound_plv = STLB[way][stlb_idx].PLV1
            sfound_rplv = STLB[way][stlb_idx].RPLV1
            sfound_pfn = STLB[way][stlb_idx].PFN1
        else:
            # There are multiple hits, so the processor behavior will be
undefined.

# look up MTLB
mtlb_found = 0
for i in range (MTLB_ENTRIES):
    if (MTLB[i].E == 1) and
    ((MTLB[i].G == 1) or (MTLB[i].ASID == CSR.ASID.ASID)) and
    (MTLB[i].VPPN[VALEN-1:MTLB[i].PS+1] == va[VALEN-1:MTLB[i].PS+1]):
    if (mtlb_found == 0):
            mtlb_found = 1
            mfound_ps - MTLB[i].PS
        if (va[mfound_ps] == 0):
            mfound_v = MTLB[i].V0
            mfound_d = MTLB[i].DO
            mfound_nr = MTLB[i].NRO
            mfound_ne - MTLB[i].NEO
            mfound_mat = MTLB[i].MATO
            mfound_plv = MTLB[i].PLV0
            mfound_rplv = MTLB[i].RPLVO
            mfound_pfn = MTLB[i].PFNO
        else:
            mfound_v = MTLB[i].V1
            mfound_d = MTLB[i].D1
            mfound_nr = MTLB[i].NR1
            mfound_ne = MTLB[i].NE1
            mfound_mat = MTLB[i].MAT1
            mfound_plv = MTLB[i].PLV1
            mfound_rplv = MTLB[i].RPLV1
            mfound_pfn = MTLB[i].PFN1
    else:
        # There are multiple hits, so the processor behavior will be
undefined.

if (stlb_found == 1) and (mtlb_found == 1):
    # There are multiple hits, so the processor behavior will be
undefined.
elif (stlb_found == 1):
    found_v = sfound_v
    found_d = sfound_d
    found_nr = sfound_nr
    found_ne = sfound_ne
    found_mat = sfound_mat
    found_plv = sfound_plv
    found_rplv = sfound_rplv
    found_pfn = sfound_pfn
    found_ps = stlb_ps
elif (mtlb_found == 1):
    found_v = mfound_v
    found_d = mfound_d
    found_nr = mfound_nr
    found_ne = mfound_ne
    found_mat = mfound_mat
    found_plv = mfound_plv
    found_rplv = mfound_rplv
    found_pfn = mfound_pfn
    found_ps = mfound_ps
else:
    SignalException(TLBRD)    # Trigger TLB refill exception.

if (found_v == 0):
    case mem_type:
    FETCH : SignalException(PIF)    # Trigger page invalid exception for
fetch operation.
    LOAD : SignalException(PIL)     # Trigger page invalid exception for
load operation.
    STORE : SignalException(PIS)    # Trigger page invalid exception for
store operation.
elif (mem_type == FETCH) and (found_ne == 1):
    SignalException(PNX)    # Trigger page non-executable exception.
elif ((found_rplv == 0) and (plv > found_plv)) or
     ((found_rplv == 1) and (plv != found_plv)):
    SignalException(PPE)    # Trigger page privilege level ilegal
exception.
elif (mem_type == L0AD) and (found_nr == 1):
    SignalException(PNR)    # Trigger page non-readable exception.
elif (mem_type == STORE) and (found_d == 0)
     and ((plv == 3) or (CSR.MISC[16+plv] == 0)) :    # The function
that disable the check of write protection is not enabled.
    SignalException(PME)    # Trigger page modify exception.
else:
    pa = {found_pfn[PALEN-1:found_ps], va[found.ps-1:0]}
    mat = found_mat
```

5.4.5. Multi-level Page Table Structure Supported by page walking

Whether the LDDIR and LDPTE instructions are used to implement software page walking or hardware page walking, the supported multi-level page table structure is the same, as shown in the figure.

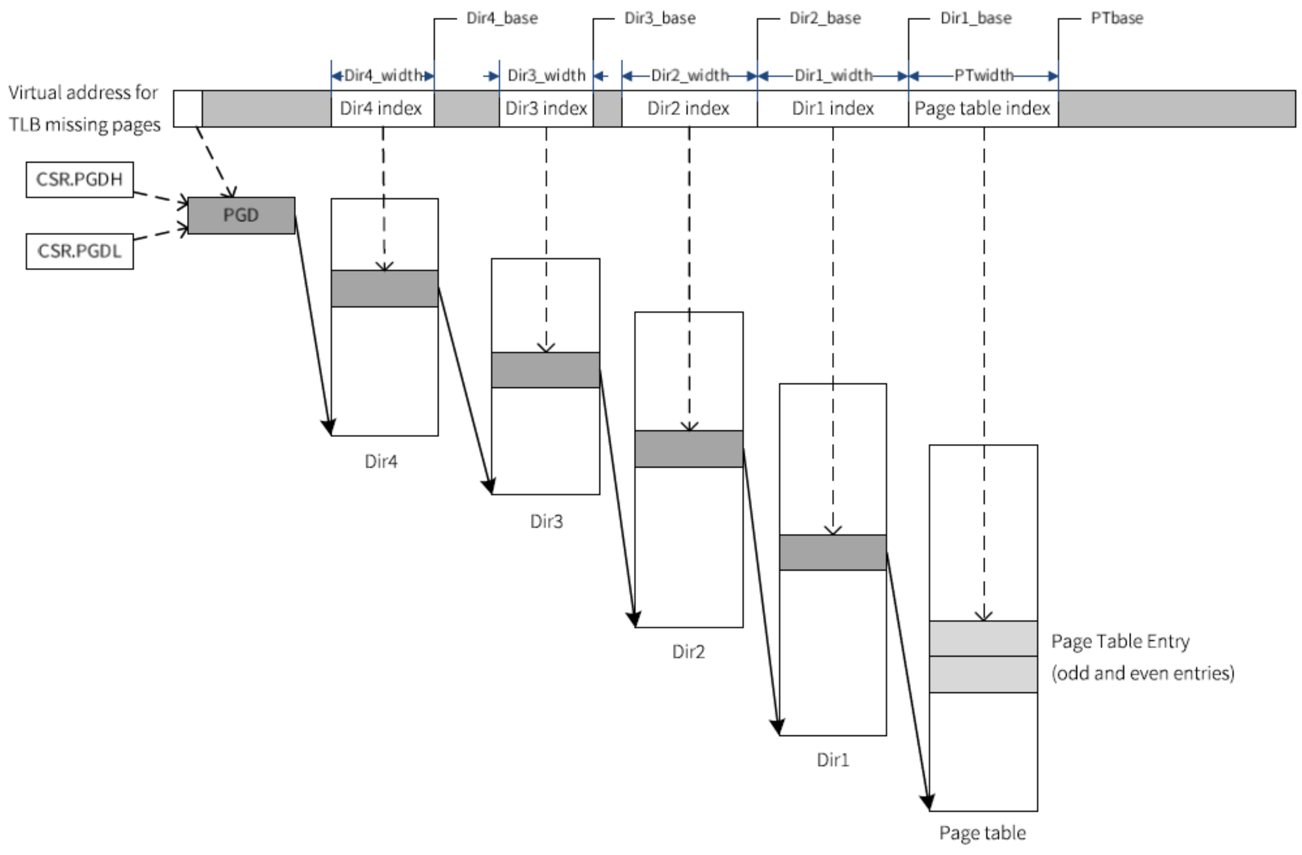

Figure 7. Multi-level page table structure supported by page walking

The base address of the top-level directory (global directory) of the traversed page table called PGD is determined by the `(PALEN-1)` bit of the queried virtual address. When this bit is `0`, the PGD comes from `CSR.PGDL`; when this bit is 1, the PGD comes from `CSR.PGDH`. This means that the entire page table structure is `(PALEN-1)` bits.

The specifications of each level of directory entries and page table entries are configured by the system software in `CSR.PWCL` and `CSR.PWCH`.

Whether the LDDIR and LDPTE instructions are used to implement software page walking or hardware page walking, the system software needs to define the page table entries in the following format.

63	62	61		PALEN-1 … 12		8	7	6	5 4	3 2	1	0
RPLV	NX	NR		PA[PALEN-1:12]		W	P	G	MAT	PLV	D	V

Figure 8. Table entry format for common pages

63	62	61		PALEN-1 … 24		12		8	7	6	5 4	3 2	1	0
RPLV	NX	NR		PA[PALEN-1:24]		G		W	P	H	MAT	PLV	D	V

Figure 9. Table entry format for huge pages

In the above definition of the page table entry format, the main differences between the page table entry of a huge page and the page table entry of a common page are:

1. Bit `6` of the directory entry is the huge page table entry flag bit, and `1` indicates that the directory entry actually stores the page table entry of a huge page at this time;
2. The `G` bit of the common page table entry is in bit `6`, while the `G` bit of the huge page table entry is in bit `12`.

Bits not defined in either of these formats are automatically ignored by the `LDDIR` and `LDPTE` instructions or hardware page walking.

The `P` field defined in the above page table entry format represents whether the physical page exists, and the `W` field represents whether the page is writable. This information is not filled in the TLB table entry, but is used during the page walking.

Due to the double-page memory structure of the TLB table entries, for the huge page table entries (which has only one), the hardware page table refill or the software LDPTE instruction will automatically split the two page table entries in half according to the information of the huge page table entries and then fill in the TLB. For example, if the standard page size is `16KB`, the size of the first-level huge page size is usually `32MB`. After the `LDPTE rj, 0` and `LDPTE rj, 1` instructions are executed during page walking, The TLB will be filled with two page table entries (page size is `16MB`) without special software intervention.

Because the address mapping is in direct address translation mode during **TLB R**efill exception (TLBR), the addresses configured in the `PGD` and in the directory entries of the page table in memory must be physical addresses.

Chapter 6. Exceptions and interrupts

6.1. Interrupts

6.1.1. Interrupt Types

Interrupts in LoongArch take the form of line-based interrupts. Each processor core can record 13 line-based interrupts: one **I**nter-**P**rocessor **I**nterrupt (IPI), one **T**imer **I**nterrupt (TI), one **P**erformance **M**onitor **C**ounter **O**verflow **I**nterrupt (PMCOV), eight **H**ard**W**are **I**nterrupts (HWI0-HWI7), and two **S**oft**W**are **I**nterrupts (SWI0-SWI1). All line-based interrupts are level-triggered and are high level triggered.

The interrupt source for inter-processor interrupts comes from an interrupt controller outside the core, which is recorded by the processor core in the `CSR.ESTA.IS[12]` bit.

The interrupt source for the timer interrupt is from the constant frequency timer in the core. This interrupt is triggered when the constant frequency timer counts down to zero. The timer interrupt is recorded by the processor core in the `CSR.ESTA.IS[11]` bit. Clearing the timer interrupt is accomplished by the software via writing `1` to `CSR.TICLR.TI`.

The interrupt source for the performance monitor counter overflow interrupt comes from the performance monitor counter in the core. This interrupt is triggered when the `[63]` bit of the performance counter of any enabled interrupt is `1`. The performance monitor counter overflow interrupt is recorded by the processor core in the `CSR.ESTA.IS[10]` bit. To clear a performance monitor counter overflow interrupt, set the performance monitor counter of the interrupt that is triggered to `0` at the `[63]` bit, or disable the interrupt for that performance monitor counter.

The interrupt source for hardware interrupts comes from outside the processor core, and its direct source is usually an interrupt controller outside the core. 8 hardware interrupts (`HWI[7:0]`) are recorded by the processor core in the `CSR.ESTA.IS[9:2]` bits .

The source of the software interrupt comes from the internal core of the processor, and the software writes `1` to `CSR.ESTA.IS[1:0]` to set up the software interrupt and `0` to clear it.

The index of the location of the interrupt recorded by the `CSR.ESTA.IS` field is also called the **Int**errupt **Number** (Int Number). Int number for SWI0 is equal to `0`, int number for SWI1 is equal to `1`, … , int number of IPI is equal to `12`.

6.1.2. Interrupt Priority

The response to multiple interrupts at the same time is arbitrated by a fixed priority. The higher the int number, the higher the priority. Therefore, IPI has the highest priority, TI the second highest, … , SWI0 has the lowest priority.

6.1.3. Interrupt Entry

Interrupts are treated as an exception once they are marked to the instruction by the processor, so the calculation of interrupt entries follows the rules for calculating general exception entries. See Exception Entry for the rules of calculating the general exception entries. The exception number for an interrupt is its own int number plus `64`. The exception number for interrupt SWI0 is `64`, the exception number for interrupt SWI1 is `65`, … , and so on.

6.1.4. Process of Processor Responding to Interrupts

The interrupt signal from each interrupt source is recorded by the processor in the `CSR.ESTA.IS` field. The value of this field and the value of the local interrupt enable field configured by software in the `CSR.ECFG.LIE` field perform the bitwise AND operation to obtain a 13-bit interrupt vector (`int_vec`). When `CSR.CRMD.IE=1` and `int_vec` is not all `0` values, the processor considers that there is an interrupt that needs to be responded to. So the processor picks an instruction from the executed instruction stream and marks it with a special kind of exception — interrupt exception.

The subsequent process of the processor is the same as that of the general exception, see the description in General Hardware Exception Handling of General Exceptions.

6.2. Message-Interrupts

6.2.1. Message-Interrupt Types

In the Loongson architecture, 256 message interrupts can be recorded inside each logical processor core, which can include message-type intercore interrupts and message-type hard interrupts input from outside the processor core. Within a processor core, four 64-bit CSRS, CSR.MSGIS0 to CSR.MSGIS3, record in turn whether messages from 0 to 255 are interrupted or not. The exact source of the 256 message interrupts inside each processor core is determined by the implementation, and software developers need to refer to the specific chip user manual for information.

6.2.2. Message-Interrupt-Priority

Loongson architecture adopts a fixed priority among 256 message interrupts in each logical processor core. The larger the interrupt number, the higher the priority, that is, message number 255 has the highest priority, message number 254 has the second,…… , message 0 has the lowest interrupt priority.

Only message interrupts recorded inside each logical processor core whose priority is not lower than the message interrupt enable priority threshold (recorded in the CSR.MSGIE.PT field) can be further selected and set by the hardware.

When there are both message interrupt request and line interrupt request in a processor core, the message interrupt request has higher priority than the line interrupt request.

6.2.3. Message-Interrupt-Entry

All message interrupts adopt a uniform entry, and the "entry page number" of their computed entry address coincides with the line interrupt, coming from an "in-page offset" equal to 2(CSR.ECFG.VS+2)×78(0x4E) of their computed entry address.

6.2.4. Message-Interrupt-Response-Processing

When the message interrupt is routed to the specified processor core, the processor core will set the corresponding status position of internal CSR.MSGIS0~CSR.MSGIS3 to 1 according to the interrupt number, and this process is recorded for the message interrupt. Then the processor core selects the interrupt with the highest priority among the recorded message interrupts whose interrupt number is not lower than the message interrupt enable priority threshold (recorded in the CSR.MSGIE.PT field), and records its message interrupt number in the CSR.MSGOR.INTNUM field, and sets the CSR.MSGOR.NULL bit to zero CSR.ESTAT.MSGINT position 1, this process picks and sets the message interrupt request for the message interrupt. When the CSR.ESTAT.MSGINT bit is 1, only the global interrupt enables CSR.CRMD.IE to block the message interrupt requests that are picked and set.

In the case that the CSR.ESTAT.MsgInt bit is 1, if the software reads the CSR.MSGIR register, the hardware will automatically clear the corresponding status of CSR.MSGIS0 to CSR.MSGIS3 according to the message interrupt number currently recorded in the CSR.MSGIR.INTUM field 0. If there is no more selected message interrupt in CSR.MSGIS0~CSR.MSGIS3 after the interrupt status bit of this message is cleared 0, the CSR.ESTAT.MSGINT bit will be cleared 0 by the hardware and CSR.MSGIR.NULL position 1 in the next processor core internal clock cycle. Software developers are especially reminded that because of the "read clear" nature of the CSR.MSGIR register, it is recommended to read the CSR.ESTAT.MSGINT bits when checking for pending message interrupts.

6.3. Exceptions

6.3.1. Exception Entry

The entry for the TLB refill exception comes from `CSR.TLBRENTRY`.

The entry for the machine error exception comes from `CSR.MERRENTRY`.

Exceptions other than the above two exceptions are called general exceptions, and their entries are calculated by `address|offset`. Here `|` is a bitwise OR operation.

All general exception entries have the same base address from `CSR.EENTRY`.

The offset of the general exception entry is determined by both the mode of the interrupt offset and the exception number (ecode), which is equal to $2^{(\mathtt{CSR.ERG.V+2})}\times(\mathtt{ecode+64})$. See the ecode column in Table of exception encoding for general exceptions except interrupts; the ecode for interrupts is its int number plus `64`.

When `CSR.ECFG.VS=0`, all general exceptions have the same entry, and the software needs to determine the specific exception type by `Ecode` and `IS` fields in `CSR.ESTA`. When `CSR.ECFG.VS!=0`, different interrupt sources have different exception entries and the software does not need to confirm the exception type by `CSR.ESTA`.

Since the exception entry is an offset on the base address calculated by bitwise OR operation, when `CSR.ECFG.VS!=0`, during assigning the exception entry base address, the software needs to ensure that all possible offsets do not exceed the bound alignment space corresponding to the low bit of the entry base address.

6.3.2. Exception Priority

The exception priority follows two basic principles: first, the interrupt priority is higher than the exception; second, for the exception, the highest priority is detected in the fetching stage, followed by the priority detected in the decoding stage, and the priority detected in the execution stage.

For exceptions detected in the fetching stage: the highest priority is given to the fetch operation watchpoint exception, the second highest priority is given to the fetch operation address error exception, the second highest priority is given to TLB-related exceptions, and the lowest priority is given to the machine error exception.

The exceptions that can be detected in the decoding stage are mutually exclusive, so there is no need to consider the priority between them.

Only memory access instructions may trigger multiple exceptions at the same time during the execution stage, with the following priorities in descending order: **A**ddress a**L**ignment fault **E**xception (ALE) caused by unaligned addresses for memory access instructions requesting alignment addresses > **AD**dress error **E**xception (ADE) > **B**ound **C**heck **E**xception (BCE)[1] > TLB-related exceptions[2] > **A**ddress a**L**ignment fault **E**xception (ALE) caused by addresses that span two pages of different Cache attributes for memory access instructions allowing non-alignment addresses.

6.3.3. General Hardware Exception Handling of General Exceptions

There may be some differences in the handling of different general exceptions by the processor, and the general hardware exception handling of general exceptions is described here.

When a general exception is triggered, the processor does the following:

- Store PLV and IE in CSR.CRMD to PPLV and PIE in CSR.PRMD, then set PLV in CSR.CRMD to 0 and IE to 0;
- For implementations that support the Watch function, also store WE in CSR.CRMD to PWE in CSR.PRMD and then set WE in CSR.CRMD to 0;
- Record PC that triggered the exception by CSR.ERA;
- Jump to the exception entry to fetch instructions.

When the software executes the ERTN instruction returning from general exceptions, the processor does the following:

- Restore PPLV and PIE in CSR.PRMD to PLV and IE in CSR.CRMD;
- For implementations that support the Watch function, also restore PWE in CSR.PRMD to WE in CSR.CRMD;
- Jump to the address recorded by CSR.ERA to fetch instructions.

For the above hardware implementation, the software needs to save PPLV and PIE in CSR.PRMD if the interrupt needs to be enabled during the exception handling, and restore the saved contents to CSR.PRMD before the exception returns.

6.3.4. Hardware Exception Handling of TLB Refill Exception

When the TLB refill exception is triggered, the processor does the following:

- Store PLV and IE in CSR.CRMD to PPLV and PIE in CSR.TLBRPRMD, then set PLV in CSR.CRMD to 0, IE to 0, DA to 1 and PG to 0.
- For implementations that support the Watch function, also store WE in CSR.CRMD to PWE in CSR.TLBRPRMD, and then set WE in CSR.CRMD to 0;
- Record the [GRLEN-1:2] bits of the PC that triggered the exception instruction by ERA in CSR.TLBRERA, and set IsTLBR in CSR.TLBRERA to 1;
- Record the virtual memory access address that triggered the exception (or PC if triggered by fetching instructions) by CSR.TLBRBADV and the [PALEN-1:13] bits of address by VPPN in CSR.TLBREHI;
- Jump to the exception entry configured by CSR.TLBRENTRY to fetch instructions.

When software executes the ERTN instruction to return from TLB refill exception, the processor does the following:

- Restore PPLV and PIE in CSR.TLBRPRMD to PLV and IE in CSR.CRMD;
- For implementations that support the Watch function, restore PWE in CSR.TLBRPRMD to WE in CSR.CRMD;
- Set DA in CSR.CRMD to 0 and PG to 1;
- Set IsTLBR in CSR.TLBRERA to 0;
- Jump to the address recorded by CSR.TLBRERA to fetch instructions.

6.3.5. Hardware Exception Handling of Machine Error Exception

When the machine error exception is triggered, the processor does the following:

- Store PLV, IE, DA, PG, DATF and DATM in CSR.CRMD to PPLV, PIE, PDA, PPG, PDATF and PDATM in CSR.MERRCTL, then set PLV in CSR.CRMD to 0, IE to 0, DA to 1, PG to 0, DATF to 0, and DATM to 0;
- For implementations that support the Watch function, also store WE in CSR.CRMD to PWE in CSR.MERRCTL, and then set WE in CSR.CRMD to 0;
- Record PC that triggered the exception instruction by CSR.MERRERA;
- Set IsMERR in CSR.MERRCTL to 1;
- Record the specific error message by CSR.ERRINFO and CSR.MERRINFO1;
- Jump to the exception entry configured by CSR.MERRENTRY to fetch instructions.

When the software executes the ERTN instruction returning from the machine error exception, the processor does the following:

- Restore PPLV, PIE, PDA, PPG, PDATF and PDATM in CSR.MERRCTL;
- For implementations that support the Watch function, also restore PWE in CSR.MERRCTL to WE in CSR.CRMD;
- Set the IsMERR in CSR.TLBRERA to 0;
- Jump to the address recorded by CSR.MERRERA to fetch instructions.

6.4. Reset

A reset will reset all logic in the processor core and place the circuit in a determined state. The definition of the state of the processor after reset is given here.

The PC after the reset is 0x1C000000. Since the MMU must be in direct address translation mode after the reset, the physical address of the first instruction fetched after reset is also 0x1000000.

After the reset, the contents of the registers in the determined state are:

- PLV in CSR.CRMD is 0, IE is 0, DA is 1, PG is 0, DATF is 0, DATM is 0, and WE is 0;
- FPUen, VPUen, XVPUen and BTUen in CSR.PUCTL are all 0 values;
- All configurable bits in CSR.MISC are 0;
- VS and LIE in CSR.ECFG are 0;
- All bits of IS[1:0] in CSR.ESTA are 0;
- RDVA in CSR.RVACFG is 0;
- En in CSR.TCFG is 0;
- KLO in CSR.LLBCTL is 0;
- IsTLBR in CSR.TLBRERA is 0;
- IsMERR in CSR.MERRCTL is 0;
- PLV0-PLV3 in all implemented CSR.DMWs are 0;

- All configurable bits except `EvCode` in all implemented `CSR.PMCFG`s are `0`;
- All configurable bits in all implemented data breakpoint CSRs are `0`;
- All configurable bits in all implemented instruction breakpoint CSRs are `0`;
- `DST` in `CSR.DEBUG` is `0`.

In addition to what is specified above, the values of all other software-visible registers in the processor are undefinded after the reset. The software has to set their values before they can be used.

Whether TLB and Cache need to do a hardware reset during the reset is decided by the implementation. The software responsible for booting determines whether to do a software reset via the processor configuration information.

1. It is generated only when it is a memory access instruction of bound class.

2. The definition of TLB-related exceptions dictates that only one TLB-related exception will be generated by a single memory access instruction in any case.

Chapter 7. Control and Status Registers

7.1. Overview of Control and Status Registers

Table 14. Overview of Control and Status Registers

Address	Name	
`0x0`	**C**u**R**rent **M**o**D**e information	`CRMD`
`0x1`	**PR**e-exception **M**o**D**e information	`PRMD`
`0x2`	**E**xtended component **U**nit **EN**able	`EUEN`
`0x3`	**MISC**ellaneous controller	`MISC`
`0x4`	**E**xception **C**on**F**i**G**uration	`ECFG`
`0x5`	**E**xception **STAT**us	`ESTAT`
`0x6`	**E**xception **R**eturn **A**ddress	`ERA`
`0x7`	**BAD** virtual **A**ddress	`BADV`
`0x8`	**BAD I**nstruction	`BADI`
`0xC`	Exception **ENTRY** address	`EENTRY`
`0x10`	**TLB I**n**D**e**X**	`TLBIDX`
`0x11`	**TLB E**ntry **HI**gh-order bits	`TLBEHI`
`0x12`	**TLB E**ntry **LO**w-order bits **0**	`TLBELO0`
`0x13`	**TLB E**ntry **LO**w-order bits **1**	`TLBELO1`
`0x18`	**A**ddress **S**pace **ID**entifier	`ASID`
`0x19`	**P**age **G**lobal **D**irectory base address for **L**ower half address space	`PGDL`
`0x1A`	**P**age **G**lobal **D**irectory base address for **H**igher half address space	`PGDH`
`0x1B`	**P**age **G**lobal **D**irectory base address	`PGD`
`0x1C`	**P**age **W**alk **C**ontroller for **L**ower half address space	`PWCL`
`0x1D`	**P**age **W**alk **C**ontroller for **H**igher half address space	`PWCH`
`0x1E`	**STLB P**age **S**ize	`STLBPS`
`0x1F`	**R**educed **V**irtual **A**ddress **C**on**F**i**G**uration	`RVACFG`
`0x20`	**CPU ID**entity	`CPUID`
`0x21`	**P**rivileged **R**esource **C**on**F**i**G**uration **1**	`PRCFG1`
`0x22`	**P**rivileged **R**esource **C**on**F**i**G**uration **2**	`PRCFG2`
`0x23`	**P**rivileged **R**esource **C**on**F**i**G**uration **3**	`PRCFG3`
`0x30+n (0 ≤ n ≤ 15)`	Data **SAVA** register	`SAVEn`
`0x40`	**T**imer **ID**entity	`TID`
`0x41`	**T**imer **C**on**F**i**G**uration	`TCFG`

Address	Name	
`0x42`	**T**imer **VAL**ue	`TVAL`
`0x43`	**C**ou**NT**er **C**ompensation	`CNTC`
`0x44`	**T**imer **I**nterrupt **CL**ea**R**ing	`TICLR`
`0x60`	**LLB**it **C**on**T**ro**L**ler	`LLBCTL`
`0x80`	**IMP**lementation-specific **C**on**T**ro**L**ler **1**	`IMPCTL1`
`0x81`	**IMP**lementation-specific **C**on**T**ro**L**ler **2**	`IMPCTL2`
`0x88`	**TLB R**efill exception **ENTRY** address	`TLBRENTRY`
`0x89`	**TLB R**efill exception **BAD V**irtual address	`TLBRBADV`
`0x8A`	**TLB R**efill **E**xception **R**eturn **A**ddress	`TLBRERA`
`0x8B`	**TLB R**efill exception data **SAVE** register	`TLBRSAVE`
`0x8C`	**TLB R**efill exception **E**ntry **LO**w-order bits **0**	`TLBRELO0`
`0x8D`	**TLB R**efill exception **E**ntry **LO**w-order bits **1**	`TLBRELO1`
`0x8E`	**TLB R**efill exception **E**ntry **HI**gh-order bits	`TLBREHI`
`0x8F`	**TLB R**efill exception **PR**e-exception **M**o**D**e information	`TLBRPRMD`
`0x90`	**M**achine **ERR**or **C**on**T**ro**L**ler	`MERRCTL`
`0x91`	**M**achine **ERR**or **INFO**rmation **1**	`MERRINFO1`
`0x92`	**M**achine **ERR**or **INFO**rmation **2**	`MERRINFO2`
`0x93`	**M**achine **ERR**or exception **ENTRY** address	`MERRENTRY`
`0x94`	**M**achine **ERR**or **E**xception **R**eturn **A**ddress	`MERRERA`
`0x95`	**M**achine **ERR**or exception data **SAVE** register	`MERRSAVE`
`0x98`	**C**ache **TAG**s	`CTAG`
`0x180+n (0 ≤ n ≤ 3)`	**D**irect **M**apping configuration **W**indow **n**	`DMWn`
`0x200+2n (0 ≤ n ≤ 31)`	**P**erformance **M**onitor **C**on**F**i**G**uration **n**	`PMCFGn`
`0x201+2n (0 ≤ n ≤ 31)`	**P**erformance **M**onitor overall **C**ou**NT**er **n**	`PMCNTn`
`0x300`	**M**emory load/store **W**atch**P**oint overall **C**ontroller	`MWPC`
`0x301`	**M**emory load/store **W**atch**P**oint overall **S**tatus	`MWPS`
`0x310+8n (0 ≤ n ≤ 7)`	**M**emory load/store **W**atch**P**oint **n C**on**F**i**G**uration **1**	`MWPnCFG1`
`0x311+8n (0 ≤ n ≤ 7)`	**M**emory load/store **W**atch**P**oint **n C**on**F**i**G**uration **2**	`MWPnCFG2`
`0x312+8n (0 ≤ n ≤ 7)`	**M**emory load/store **W**atch**P**oint **n C**on**F**i**G**uration **3**	`MWPnCFG3`

Address	Name	
`0x313+8n (0 ≤ n ≤ 7)`	**M**emory load/store **W**atch**P**oint **n** **C**on**F**i**G**uration **4**	`MWPnCFG4`
`0x380`	**F**etch **W**atch**P**oint overall **C**ontroller	`FWPC`
`0x381`	**F**etch **W**atch**P**oint overall **S**tatus	`FWPS`
`0x390+8n (0 ≤ n ≤ 7)`	**F**etch **W**atch**P**oint **n** **C**on**F**i**G**uration **1**	`FWPnCFG1`
`0x391+8n (0 ≤ n ≤ 7)`	**F**etch **W**atch**P**oint **n** **C**on**F**i**G**uration **2**	`FWPnCFG2`
`0x392+8n (0 ≤ n ≤ 7)`	**F**etch **W**atch**P**oint **n** **C**on**F**i**G**uration **3**	`FWPnCFG3`
`0x393+8n (0 ≤ n ≤ 7)`	**F**etch **W**atch**P**oint **n** **C**on**F**i**G**uration **4**	`FWPnCFG4`
`0x500`	**D**e**B**u**G** register	`DBG`
`0x501`	**D**ebug **E**xception **R**eturn **A**ddress	`DERA`
`0x502`	**D**ebug data **SAVE** register	`DSAVE`

7.2. Characteristics of Accessing Control and Status Registers

7.2.1. Attributes of Reading and Writing

The definition of the “read/write” attribute for each field is described later in this manual in the control and status register field definition. The “read/write” attributes are defined primarily from the perspective of software and are divided into four types:

- RW - readable and writable. Software can write any value, except for illegal values that are explicitly stated in the definition and lead to uncertainty in the processor’s execution. Normally, software writes to these fields before it reads them, and what is read should be the value written. However, when the accessed field can be updated by hardware, or when an interrupt occurs between the two instructions executing the read and write operation, it is possible that the read value is not consistent with the written value.
- R - read-only. Software writes to these fields will not update their contents, and will have no side effects.
- R0 - always return `0` if read these fields. But at the same time software must ensure that either it avoids updating these fields by setting the CSR write mask bit, or it must write `0` when updating these fields. This requirement is to ensure software backward compatibility. For hardware implementations, fields marked with this attribute will prohibit software writing.
- W1 - write `1` is valid. Software writes `0` to these fields will not clear them to `0` and will have no side effects. Also, the read values of these fields have no real meaning and software should ignore these values.

7.2.2. Length of Control and Status Registers in LA32 and LA64

The length of all status control registers is either fixed 32 bits, or it depends on whether the implementation is LA32 or LA64. For the first type of registers, when they are accessed by CSR instructions in LA64, retrun values of reading these registers are symbolic expansion to 64 bits, and bits higher than 32 bits of values of writing to them are automatically ignored by hardware. For the second type, the definitions will clearly indicate the difference between LA32 and LA64.

7.2.3. Access Effects of Undefined and Unimplemented Control and Status Registers

When software uses CSR instructions to access CSR objects that are not defined in the architecture specification or that are implementable entries defined in the architecture specification but not implemented by the specific hardware, the return value of reading can be any value, but the write operation will not change the software-visible processor state.

Although software writes to these undefined or unimplemented status control registers do not change the software-visible processor state, software should not write to these registers if it wants to ensure backward compatibility.

7.3. Conflicts Caused by Control and Status Registers

Conflicts caused by the control and status register are maintained by the hardware, and the software does not need to add barrier-type instructions for avoiding conflict.

7.4. Basic Control and Status Registers

7.4.1. Current Mode Information (`CRMD`)

The information in this register is used to determine the the processor core’s privilege level, global interrupt enable bit, watchpoint enable bit, and address translation mode at that time.

Table 15. Definition of current mode information register

Bits	Name	Read/Write	Description
`1:0`	`PLV`	RW	Current privilege level. The legal value range is `0` to `3`, where `0` is the highest privilege level and `3` is the lowest privilege level. When an exception is triggered, the hardware sets the value of this field to `0` to ensure that it is at the highest privilege level after being caught. When the `ERTN` instruction is executed to return from the exception handler, if `CSR.MERRCTL.IsMERR=1`, the hardware restores the value of the `PPLV` field of `CSR.MERRCTL` to here; otherwise, if `CSR.TLBRERA.IsTLBR=1`, the hardware restores the value of the `PPLV` field of `CSR.TLBRPRMD` to here; otherwise, the hardware restores the value of the `PPLV` field of `CSR.TLBRPRMD` to here; otherwise, the hardware restores the value of the `PPLV` field of `CSR.TLBRPRMD` to here. Hardware restores the value of the `PPLV` field of `CSR.PRMD` to here.

Bits	Name	Read/Write	Description
2	IE	RW	Current global interrupt enable bit, which is active high. When an exception is triggered, the hardware sets the value of this field to 0, to ensure that the interrupt is masked when caught. This field needs to be explicitly set to 1 when the exception handler decides to re-open the interrupt response. When the ERTN instruction is executed to return from the exception handler, if CSR.MERRCTL.IsMERR=1, the hardware restores the value of the PIE field of CSR.MERRCTL to this field; Otherwise, if CSR.TLBRERA.IsTLBR=1, the hardware restores the value of the PIE field of CSR.TLBRPRMD here; Otherwise, the hardware restores the value of the PIE field of CSR.PRMD to here.
3	DA	RW	Direct address translation mode enable bit, which is active high. The hardware sets this field to 1 when a TLB refill exception or a machine error exception is triggered. If CSR.MERRCTL.IsMERR=1, the hardware restores the value of the PDA field of CSR.MERRCTL when the ERTN instruction is executed and returns from the exception handler; otherwise, if CSR.TLBRERA.IsTLBR=1, the hardware sets this field to 0. The legal combination of DA and PG bits is 0, 1 or 1, 0. The result is uncertain when the software is configured for other combinations.
4	PG	RW	Mapped address translation mode enable bit, which is active high. The hardware sets this field to 0 when a TLB refill exception or a machine error exception is triggered. When the ERTN instruction is executed to return from an exception handler, if CSR.MERRCTL.IsMERR=1, the hardware restores the value of the PPG field of CSR.MERRCTL to this; otherwise, if CSR.TLBRERA.IsTLBR=1, the hardware sets this field to 1. The legal combination of PG and DA bits is 0, 1 or 1, 0. The result is uncertain when the software is configured for other combinations.

Bits	Name	Read/Write	Description
6:5	DATF	RW	The type of memory access for fetch operations when in direct address translation mode. The hardware sets this field to 0 when a machine error exception is triggered. When the execution of the ERTN instruction returns from the exception handler and CSR.MERRCTL.IsMERR=1, the hardware restores the value of the PDATF field of CSR.MERRCTL to here. In the case of using software to handle TLB refill, when the software sets PG to 1, it needs to set the DATF field to 0b01 at the same time, which is the consistent cacheable type.
8:7	DATM	RW	The type of memory access for load and store operations when in direct address translation mode. The hardware sets this field to 0 when a machine error exception is triggered. When the execution of the ERTN instruction returns from the exception handler and CSR.MERRCTL.IsMERR=1, the hardware restores the value of the PDATM field of CSR.MERRCTL to here. In the case of using software to handle TLB refill, when the software sets PG to 1, it needs to set DATM to 0b01 at the same time, i.e., consistent cacheable type.
9	WE	RW	Instruction and data watchpoints enable bit, which is active high. The hardware sets the value of this field to 0 when an exception is triggered. When the ERTN instruction is executed to return from the exception handler. If CSR.MERRCTL.IsMERR=1, the hardware restores the value of the PWE field of CSR.MERRCTL to here; otherwise, if CSR.TLBRERA.IsTLBR=1, the hardware restores the value of the PWE field of CSR.TLBRPRMD to here; Otherwise, the hardware restores the value of the PWE field of CSR.PRMD here.
31:10	0	R0	Reserved field. Return 0 if read this field and the software does not allow to change its value.

7.4.2. Pre-exception Mode Information (PRMD)

When an exception is triggered, if the exception type is not TLB refill exception and machine error exception, the hardware will save the processor core's privilege level, global interrupt enable bit and watchpoint enable bit at that time to the pre-exception mode information register for restoring the processor core to the context when the exception returns.

Table 16. Definition of pre-exception mode information register

Bits	Name	Read/Write	Description
1:0	PPLV	RW	When an exception is triggered, the hardware records the old value of the PLV field in CSR.CRMD in this field if the exception type is not a TLB refill exception and a machine error exception. When the exception being processed is neither a TLB refill exception (CSR.TLBRERA.IsTLBR=0) nor a machine error exception (CSR.MERRCTL.IsMERR=0), the hardware restores the value of this field to the PLV field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
2	PIE	RW	When an exception is triggered, the hardware records the old value of the IE field in CSR.CRMD in this field if the exception type is not a TLB refill exception and a machine error exception. When the exception being processed is neither a TLB refill exception (CSR.TLBRERA.IsTLBR=0) nor a machine error exception (CSR.MERRCTL.IsMERR=0), the hardware restores the value of this field to the IE field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
3	PWE	RW	When an exception is triggered, the hardware records the old value of the WE field in CSR.CRMD in this field if the exception type is not a TLB refill exception and a machine error exception. When the exception being processed is neither a TLB refill exception (CSR.TLBRERA.IsTLBR=0) nor a machine error exception (CSR.MERRCTL.IsMERR=0), the hardware restores the value of this field to the WE field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
31:4	0	R0	Reserved field. Return 0 if read this field and the software does not allow to change its value.

7.4.3. Extended Component Unit Enable (EUEN)

In addition to the base integer instruction set and the privileged instruction set, the base floating-point instruction set, the binary translation extension instruction set, the 128-bit vector extension instruction set, and the 256-bit vector extension instruction set each have software-configurable enable bits. When these enable controls are disabled, execution of the corresponding instruction will trigger the corresponding instruction unavailable exception. Software uses this mechanism to determine the scope when saving the context. Hardware implementations can also use the control bits here to implement circuit power control.

Table 17. Definition of extended component unit enable register

Bits	Name	Read/Write	Description
0	FPE	RW	The base floating-point instruction enable bit. When this bit is 0, execution of the base floating-point instruction as described in Overview of Floating-Point Instructions will trigger a floating-point instruction disable exception (FPD).
1	SXE	RW	The 128-bit vector expansion instruction enable bit. When this bit is 0, execution of the 128-bit vector expansion instruction as described in Volume 2 will trigger the 128-bit vector expansion instruction disable exception (SXD).

Bits	Name	Read/Write	Description
2	ASXE	RW	The 256-bit vector expansion instruction enables the control bit. When this bit is 0, execution of the 256-bit vector expansion instruction as described in Volume 2 will trigger the 256-bit vector expansion instruction disable exception (ASXD).
3	BTE	RW	Binary translation expansion instruction enable bit. When this bit is 0, execution of the binary translation expansion instruction described in Volume 2 will trigger the binary translation expansion instruction disable exception (BTD).
31:4	0	R0	Reserved field. Return 0 if read this field, and software is not allowed to change its value.

7.4.4. Miscellaneous Controller (MISC)

This register contains a number of control bits for the operating behavior of the processor core at different privilege levels, including whether to enable 32-bit address mode, whether to allow partially privileged instructions at non-privileged levels, whether to enable address non-alignment check, and whether to enable page table write protection check.

Table 18. Definition of miscellaneous controller register

Bits	Name	Read/Write	Description
0	0	RO	Reserved field. Return 0 if read this field and software is not allowed to change its value.
1	VA32L1	RW	Whether to enable 32-bit address mode at the PLV1 privilege level. 0 - disable, 1 - enable. This bit can be read and written only in LA64, at the LA32 privilege level, this attribute is R0.
2	VA32L2	RW	Whether to turn on 32-bit address mode at the PLV2 privilege level. 0 - disable, 1 - enable. This bit is read/write only in LA64, and at the LA32 privilege level, this attribute is R0.
3	VA32L3	RW	Whether to enable 32-bit address mode at the PLV3 privilege level. 0 - disable, 1 - enable. This bit is read/write only in LA64, and at the LA32 privilege level, this attribute is R0.
4	0	R0	Reserved field. Return 0 if read this field and software is not allowed to change its value.
5	DRDTL1	RW	Whether to disable RDTIME-like instructions at the PLV1 privilege level. When this bit is 1, execution of an RDTIME-like instruction at the PLV1 privilege level will trigger an instruction privilege level error exception (IPE).
6	DRDTL2	RW	Whether to disable RDTIME-like instructions at the PLV2 privilege level. When this bit is 1, execution of RDTIME-like instructions at PLV2 privilege level will trigger instruction privilege level error exception (IPE).
7	DRDTL3	RW	Whether to disable RDTIME class instructions at the PLV3 privilege level. When this bit is 1, execution of RDTIME-like instructions at the PLV3 privilege level will trigger an instruction privilege level error exception (IPE).
8	0	R0	Reserved field. Return 0 if read this field and software is not allowed to change its value.

Bits	Name	Read/Write	Description
9	RPCNTL1	RW	Whether to allow software reads of the performance counter at the PLV1 privilege level. When this bit is 1, the PLV1 privilege level PCNT will not trigger an instruction privilege level error exception (IPE), if the CSRRD instruction is used to access any of the implemented performance counters at the PLV1 privilege level.
10	RPCNTL2	RW	Whether software reads of the performance counter are allowed at the PLV2 privilege level. When this bit is 1, the PLV2 privilege level When this bit is 1, accessing any implemented performance counter PCNT with CSRRD instruction at the PLV2 privilege level does not trigger instruction privilege level error exception (IPE).
11	RPCNTL3	RW	Whether software reads of the read performance counter are allowed at the PLV3 privilege level. When this bit is 1, the PLV3 privilege level When this bit is 1, accessing any implemented performance counter PCNT with the CSRRD instruction at the PLV3 privilege level does not trigger an instruction privilege level error exception (IPE).
12	ALCL0	RW	Whether to perform a non-alignment check for non-vector load/store instructions that are allowed to be non-aligned at PLV0 privilege level. 1 indicates that the check is performed, and an address alignment error exception is triggered if illegal. This bit is read/write only if the hardware implementation supports non-aligned addresses for these non-vector load/store instructions. Otherwise, the bit is a read-only constant 1.
13	ALCL1	RW	Whether to perform a non-alignment check for non-vector load/store instructions[1] that are allowed to be non-aligned at the PLV1 privilege level. 1 indicates that the check is performed and triggers an address alignment error exception if illegal. This bit is read/write only if the hardware implementation supports non-aligned addresses for these non-vector load/store instructions. Otherwise, the bit is a read-only constant 1.
14	ALCL2	RW	Whether to perform a non-alignment check for non-vector load/store instructions[1] that are allowed to be non-aligned at the PLV2 privilege level. 1 indicates that the check is performed and triggers an address alignment error exception if illegal. This bit is read/write only if the hardware implementation supports non-aligned addresses for these non-vector load/store instructions. Otherwise, the bit is a read-only constant 1.
15	ALCL3	RW	Whether to perform a non-alignment check for non-vector load/store instructions[1] that are allowed to be non-aligned at the PLV3 privilege level. 1 indicates that the check is performed and triggers an address alignment error exception if illegal. This bit is read/write only if the hardware implementation supports non-aligned addresses for these non-vector load/store instructions. Otherwise, the bit is a read-only constant 1.

Bits	Name	Read/Write	Description
16	DWPL0	RW	Whether to disable the check of the page table entry write protection during TLB virtual and real address translation at the PLV0 privilege level. When this bit is 1, the store instruction will not trigger a page modification exception even if it accesses a page table entry with D=0.
17	DWPL1	RW	Whether to disable the check of the page table entry write protection during TLB virtual and real address translation at the PLV1 privilege level. When this bit is 1, the store instruction will not trigger a page modification exception even if it accesses a page table entry with D=0.
18	DWPL2	RW	Whether to disable the check of the page table entry write protection during TLB virtual and real address translation at the PLV2 privilege level. When this bit is 1, the store instruction will not trigger a page modification exception even if it accesses a page table entry with D=0.
31:19	0	RO	Reserved field. Return 0 if read this field and software is not allowed to change its value.

7.4.5. Exception Configuration (ECFG)

This register is used to control the entry calculation method of exceptions and interrupts and the local enable bit of each interrupt.

Table 19. Definition of exception configuration register

Bits	Name	Read/Write	Description
12:0	LIE	RW	Local interrupt enable bits, which are high valid. These local interrupt enable bits correspond to the 13 interrupt sources recorded in the IS field in CSR.ESTAT. Each bit controls one interrupt source.
15:13	0	RO	Reserved field. Return 0 if read this field, and software is not allowed to change its value.
18:16	VS	KW	Configure the spacing of exceptions and interrupt entries. When VS=0, all exceptions and interrupts have the same entry base address. When VS!=0, the entry base address spacing between each exception and interrupt is 2VS instructions. Since the TLB refill exceptions and machine error exceptions have separate entry base addresses, the entry of both exceptions is not affected by the VS field.
31:19	0	RO	Reserved field. Return 0 if read this field, and software is not allowed to change its value.

7.4.6. Exception Status (ESTAT)

This register records the status information of the exceptions, including the first and second level encoding of the triggered exceptions, and the status of each interrupt.

Table 20. Definition of exception status register

Bits	Name	Read/Write	Description
1:0	IS[1:0]	RW	The status bits of the two software interrupts. Bit 0 and 1 correspond to SWI0 and SWI1 respectively. Software interrupt setting is also done by these two bits, writing 1 sets the interrupt, and writing 0 clears the interrupt.
12:2	IS[12:2]	R	The interrupt status bit. 1 indicates that the corresponding interrupt is set up. There is 1 inter-processor interrupt (IPI), 1 timer interrupt (TI), 1 performance counter overflow interrupt (PCOV), 8 hardware interrupts (HWI0-HWI7). In line-based interrupt mode, the hardware only records each interrupt source per clock cycle to these bits. The requirement that all interrupts must be level interrupts at this time is guaranteed by the interrupt source and is not maintained here.
15:13	0	R0	Reserved field. Return 0 if read this field, and software is not allowed to change its value.
21:16	Ecode	R	Exception encoding. When an exception is triggered: if it is a TLB refill exception or a machine error exception, this field remains unchanged; otherwise, the hardware writes the value defined in the Ecode column in the following table to this field according to the exception type.
30:22	EsubCode	R	Exception sub encoding. When an exception is triggered: if it is a TLB refill exception or a machine error exception, the field remains unchanged; otherwise, the hardware writes the value defined in the EsubCode column in the following table to this field according to the exception type.
31	0	R0	Reserved field. Return 0 if read this field and software is not allowed to change its value.

Table 21. Table of exception encoding

Ecode	EsubCode	Exception Code	Exception Type
0x0		INT	Only when CSR.ECFG.VS=0, it means it is an **INT**errupt.
0x1		PIL	**P**age **I**nvalid exception for **L**oad operation
0x2		PIS	**P**age **I**nvalid exception for **S**tore operation
0x3		PIF	**P**age **I**nvalid exception for **F**etch operation
0x4		PME	**P**age **M**odification **E**xception
0x5		PNR	**P**age **N**on-**R**eadable exception
0x6		PNX	**P**age **N**on-e**X**ecutable exception
0x7		PPI	**P**age **P**rivilege level **I**llegal exception
0x8	0	ADEF	**AD**dress error **E**xception for **F**etching instructions
	1	ADEM	**AD**dress error **E**xception for **M**emory access instructions
0x9		ALE	**A**ddress a**L**ignment fault **E**xception

Ecode	EsubCode	Exception Code	Exception Type
0xA		BCE	**B**ound **C**heck **E**xception
0xB		SYS	**SYS**tem call exception
0xC		BRK	**BR**ea**K**point exception
0xD		INE	**I**nstruction **N**on-defined **E**xception
0xE		IPE	**I**nstruction **P**rivilege error **E**xception
0xF		FPD	**F**loating-**P**oint instruction **D**isable exception
0x10		SXD	128-bit vector (**S**IMD instructions) e**X**pansion instruction **D**isable exception
0x11		ASXD	256-bit vector (**A**dvanced **S**IMD instructions) e **X**pansion instruction **D**isable exception
0x12	0	FPE	**F**loating-**P**oint error **E**xception
	1	VFPE	**V**ecctor **F**loating-**P**oint error **E**xception
0x13	0	WPEF	**W**atch**P**oint **E**xception for **F**etch watchpoint
	1	WPEM	**W**atch**P**oint **E**xception for **M**emory load/store watchpoint
0x14		BTD	**B**inary **T**ranslation expansion instruction **D**isable exception
0x15		BTE	**B**inary **T**ranslation related exceptions
0x16		GSPR	**G**uest **S**ensitive **P**rivileged **R**esource exception
0x17		HVC	**H**yper**V**isor **C**all exception
0x18	0	GCSC	**G**uest **CS**R **S**oftware **C**hange exception
	1	GCHC	**G**uest **CS**R **H**ardware **C**hange exception
0x1A-0x3E			Reserved Codes

7.4.7. Exception Return Address (ERA)

When an exception is triggered, if the exception type is neither a TLB refill exception nor a machine error exception, the PC of the instruction that triggered the exception will be recorded in this register.

Table 22. Definition of exception program counter register

Bits	Name	Read/Write	Description
GRLEN-1:0	PC	RW	When an exception is triggered: this field remains unchanged if the exception is a TLB refill exception or a machine error exception; otherwise, the hardware records the PC of the instruction that triggered the exception here. For LA64, in this case, if the privilege level that triggered the exception is in 32-bit address mode, then the higher 32 bits of the recorded PC are forced to 0.

7.4.8. Bad Virtual Address (BADV)

This register is used to record the bad address when a bad address exception is triggered. Such exceptions include:

- **AD**dress error **E**xception for **F**etching instructions (ADEF), at this time the PC of the instruction is recorded
- **AD**dress error **E**xception for **M**emory access instructions (ADEM)
- **A**ddress a**L**ignment fault **E**xception (ALE)
- **B**ound **C**heck **E**xception (BCE)
- Page **I**nvalid exception for **L**oad operation (PIL)
- Page **I**nvalid exception for **S**tore operation (PIS)
- Page **I**nvalid exception for **F**etch operation (PIF)
- Page **M**odification **E**xception (PME)
- Page **N**on-**R**eadable exception (PNR)
- Page **N**on-e**X**ecutable exception (PNX)
- Page **P**rivilege level **I**llegal exception (PPI)

Table 23. Definition of bad virtual address register

Bits	Name	Read/Write	Description
`GRLEN-1:0`	`VAddr`	RW	When a bad address exception exception is triggered, the hardware records the bad address here. For LA64, if the privilege level that triggered the exception is in 32-bit address mode, the high `32` bits of the recorded virtual address are forced to `0`.

7.4.9. Bad Instruction (BADI)

This register is used to record the instruction code of the instruction that triggers the synchronous-related exception. The so-called synchronous-related exceptions are all exceptions except the **INT**errupt (INT), the **G**uest **C**SR **H**ardware **C**hange exception (GCHC), and the **M**achine **ERR**or exception (MERR).

Table 24. Definition of bad instruction register

Bits	Name	Read/Write	Description
`31:0`	`Inst`	R	When a synchronous-related exception is triggered, the hardware records the instruction code that triggered the exception here.

7.4.10. Exception Entry Base Address (EENTRY)

This register is used to configure the entry base address for general exceptions and interrupts.

Table 25. Definition of exception entry base address register

Bits	Name	Read/Write	Description
`11:0`	`0`	R	Read-only constant `0`, writing to this field is ignored.
`GRLEN-1:12`	`VPN`	RW	The virtual page table number of the entry base address for general exceptions and interrupts.

7.4.11. Reduced Virtual Address Configuration (`RVACFG`)

This register is used to control the length of the address being reduced in the virtual address reduction mode.

Table 26. Definition of reduced virtual address configuration register

Bits	Name	Read/Write	Description
`3:0`	`RBits`	RW	The number of the high order bits of the address to be reduced in the virtual address reduction mode. It can be configured to a value between `0` and `8`. `0` is a special configuration value that means that the virtual address reduction mode is disabled. If the configured value is greater than `8`, the processor behavior is undefined.
`31:4`	`0`	RO	Reserved field. Return `0` if read this field and software is not allowed to change its value.

7.4.12. CPU Identity (`CPUID`)

This register contains the processor core number information.

Table 27. Definition of CPU identity register

Bits	Name	Read/Write	Description
`8:0`	`CoreID`	R	The number of the processor core. This information is used by the software to distinguish the individual processor cores in a multi-core system. When the system is integrated, the processor core number information for each processor core is set by the hardware according to the specific implementation. It is recommended that the processor core number be incremented from `0` in the system.
`31:9`	`0`	RO	Reserved field. Return `0` if read this field and software is not allowed to change its value.

7.4.13. Privileged Resource Configuration 1 (`PRCFG1`)

This register contains the privileged resources information.

Table 28. Definition of privileged resource configuration 1 register

Bits	Name	Read/Write	Description
`3:0`	`SAVENum`	R	The number of `SAVE` control and status registers.
`11:4`	`TimerBits`	R	The number of valid bits of the timer minus `1`.
`14:12`	`VSMax`	R	The maximum value that can be set for the exception and interrupt vector entry spacing (`CSR.ECTL.VS`).
`31:15`	`0`	RO	Reserved field. Return `0` if read this field and software is not allowed to change its value.

7.4.14. Privileged Resource Configuration 2 (PRCFG2)

This register contains the privileged resources information.

Table 29. Definition of privileged resource configuration 2 register

Bits	Name	Read/Write	Description
GRLEN-1:0	PSAVL	R	Indicates the page size that the TLB can support (Page Size). When bit i is 1, it indicates that a page size of 2^i bytes is supported.

7.4.15. Privileged Resource Configuration 3 (PRCFG3)

This register contains the privileged resources information.

Table 30. Definition of privileged resource configuration 3 register

Bits	Name	Read/Write	Description
3:0	TLBType	R	Indicates how the TLB is organized: 0: No TLB; 1: a fully associated **M**ultiple page size **TLB** (MTLB) 2: One fully associative **M**ultiple page size **TLB** (MTLB) + one group associative **S**ingular-Page-Size **TLB** (STLB); Other values: Reserved.
11:4	MTLBEntries	R	When TLBType=0, the field is read-only constant 0; When TLBType=1 or TLBType=2, the value of this field is the number of entries in the fully associative multipage size TLB minus 1.
19:12	STLBWays	R	When TLBType=0 or TLBType=1, the field is read-only constant at 0; When TLBType=2, the value of this field is the number of ways in the group associative singular-page-size TLB minus 1.
25:20	STLBSets	R	When TLBType=0 or TLBType=1, the field is read-only constant to 0; When TLBType=2, the value of this field is the power of the number of entries per way in the group associative singular-page-size TLB, i.e., $2^{STLBSets}$ entries per way.
31:26	0	R0	Reserved field. Return 0 if read this field and the software is not allowed to change its value.

7.4.16. Data Save Register (SAVE)

The data save registers are used to temporarily store data for the system software. Each data save register can store data from one general-purpose register.

The minimum number of data save registers is 1, and the maximum number is 16. The exact number of registers can be found in CSR.PRCFG1.SAVENum. Starting from SAVE0, the addresses of each SAVE register are 0x30, 0x31, … , 0x30+SAVENum-1.

All data save control and status registers have the same format, as shown in the table.

Table 31. Definition of data save register

Bits	Name	Read/Write	Description
GRLEN-1:0	Data	RW	Data for software to read and write only. The hardware does not modify the contents of this field except for the execution of CSR instructions.

7.4.17. LLBit Controller (LLBCTL)

This register is used for the access control operations performed on the LLBit.

Table 32. Definition of LLBit controller register

Bits	Name	Read/Write	Description
0	ROLLB	R	A read-only bit. Reading this bit will return the value of the current LLBit.
1	WCLLB	W1	A software writing 1 to this bit will clear the LLBit to 0. A software writing 0 to this bit will be ignored by hardware.
2	KLO	RW	Used to control the operation of the LLBit when the ERTN instruction is executed. When this bit is 1, the LLBit is not cleared to 0 when the ERTN instruction is executed. But the bit is automatically cleared to 0 by the hardware; it means that each time KLO is set to 1, it can only affect the execution of the ERTN instruction once.
31:3	0	R0	Reserved field. Return 0 if read this field, and software is not allowed to change its value.

7.4.18. Implementation-specific Controller 1 (IMPCTL1)

This register contains control information related to the microstructure characteristics at the time of the specific implementation. Its format and the specific meaning of each field are defined by the specific implementation.

7.4.19. Implementation-specific Controller 2 (IMPCTL2)

This register contains control information related to the microstructure characteristics at the time of the specific implementation. Its format and the specific meaning of each field are defined by the specific implementation.

7.4.20. Cache Tags (CTAG)

This register is used when the CACOP instruction accesses the Cache directly, to store the contents read from the CacheTag or the contents to be written to the CacheTag. The format and the meaning of each field are defined by the implementation.

7.5. Control and Status Registers Related to Mapped Address Translation

7.5.1. TLB Index (TLBIDX)

This register contains information such as the index associated with the TLB-related instruction when executing TLB-related instructions. The length of the Index field in the table depends on implementation, although LoongArch allows for an Index length of no more than 16 bits.

This register also contains the information related to the PS and P fields in the TLB table entry when executing TLB-related instructions.

Table 33. Definition of TLB index register

Bits	Name	Read/Write	Description
n-1:0	Index	RW	When executing the TLBRD and TLBWR instructions, the index of the access TLB table entry comes from here. When executing the TLBSRCH instruction, if it hits, the index of the hit entry is recorded here. For the correspondence between index values and TLB table entries, refer to the relevant section in TLB Maintenance Instructions.
15:n	0	R	Read-only constant 0, writing to this field is ignored.
23:16	0	RO	Reserved field. Return 0 if read this field and software is not allowed to change its value.
29:24	PS	RW	When executing the TLBRD instruction, the value read from the PS field of the TLB table entry is recorded here. When executing the TLBWR and TLBFILL instructions with CSR.TLBRERA.IsTLBR=0, the value written to the PS field of the TLB table entry comes from here.
30	0	RO	Reserved field. Return 0 if read this field and the software does not allow to change its value.
31	NE	RW	1 means the TLB table entry is empty (invalid TLB table entry), and 0 means the TLB table entry is non-empty (valid TLB table entry) When executing the TLBSRCH instruction, this bit is recorded as 0 if there is a hit entry, otherwise it is recorded as 1. When executing the TLBRD instruction, the E bit read from the TLB table entry is inverted and recorded here. When executing the TLBWR instruction, and when CSR.TI.BRFPC.IsTT.BR=0, the value written to the E bit of the TLB entry is written after it is inverted. If CSR.TLBRERA.IsTLBR=1, then the E bit of the TLB entry being written is always set to 1, regardless of the value of that bit.

7.5.2. TLB Entry High-order Bits (TLBEHI)

This register contains the information related to the virtual page number of the high-order bits of the TLB table entry during ececuting TLB-related instructions. Since the length of the VPPN field contained in the high-order bits of the TLB table entry is related to the range of valid virtual addresses supported by the implementation, the definition of the relevant register field is expressed separately.

Table 34. Definition of TLB entry high order bits register in LA64

Bits	Name	Read/Write	Description
12:0	0	R	Read-only constant 0, writing to this field is ignored.
VALEN-1:13	VPPN	RW	When executing the TLBRD instruction, the value of the VPPN field read from the TLB table entry is recorded here. If CSR.TLBRERA.IsTLBR=0, the VPPN value used to query TLB when executing TLBSRCH instruction and the value of VPPN field written to TLB table entry when executing TLBWR and TLBFILL instructions come from here. When the page invalid exception for load operation, page invalid exception for store operation, page invalid exception for fetch operation, page modification exception, page non-readable exception, page non-executable exception, and page privilege level ilegal exception are triggered, the [VALEN-1:13] bits of the virual address that triggered the exception are recorded here.
63:VALEN	Sign_Ext	R	Return a signed extension value of the highest bits of the VPPN field if read this field and writing to this field is ignored.

Table 35. Definition of TLB entry high order bits register in LA32

Bits	Name	Read/Write	Description
12:0	0	R	Read-only constant 0, writing to this field is ignored.
31:13	VPPN	RW	When executing the TLBRD instruction, the value of the VPPN field read from the TLB table entry is recorded here. If CSR.TLBRERA.IsTLBR=0, the VPPN value used to query TLB when executing TLBSRCH instruction and the value of VPPN field written to TLB table entry when executing TLBWR and TLBFILL instructions come from here. When the page invalid exception for load operation, page invalid exception for store operation, page invalid exception for fetch operation, page modification exception, page non-readable exception, page non-executable exception, and page privilege level ilegal exception are triggered, the [31:13] bits of the virual address that triggered the exception are recorded here.

7.5.3. TLB Entry Low-order Bits (TLBELO0, TLBELO1)

TLBELO0 and TLBELO1 registers contain the information related to the physical page number of the low-order bits of the TLB table entry during executing TLB-related instructions. Since TLB adopts a dual-page structure, the low-order bits of TLB table entry corresponds to the odd and even physical page table entries, where the even page information is in TLBELO0 and the odd page information is in TLBELO1. TLBELO0 and

TLBELO1 registers have exactly the same format definition, and the definition of each field is in the table.

When CSR.TLBRERA.IsTLBR=0, and when executing the TLBWR and TLBFILL instructions, and the written values of the G, PFN0, V0, PLV0, MAT0, D0, NR0, NX0, RPLV0, PFN1, V1, PLV1, MAT1, D1, NR1, NX1, and RPLV1 fields of the TLB table entry come from TLBELO0 and TLBELO1 fields, respectively.

When executing the TLBRD instruction, the above information read from the TLB table entry is written to the corresponding fields in the TLBELO0 and TLBELO1 registers one by one.

Table 36. Definition of TLB entry low order bits in LA64

Bits	Name	Read/Write	Description
0	V	RW	**V**alid bit (V) of the page table entry.
1	D	RW	**D**irty bit (D) of the page table entry.
3:2	PLV	RW	**P**rivilege **L**e**V**el of the page table entry (PLV).
5:4	MAT	RW	**M**emory **A**ccess **T**ype (MAT) of the page table entry.
6	G	RW	**G**lobal flag bit (G) of the page table entry. When executing the TLBFILL and TLBWR instructions, the G bit in TLBELO0 and TLBELO1 is 1 only if both bits are 1. The G bit of the page table entry filled into the TLB will be 1 only when the G bit in both TLBELO0 and TLBELO1 is 1. When executing the TLBRD instruction, when the G bit of the TLB table entry read is 1, the G bits in TLBELO0 and TLBELO1 are set to 1 at the same time.
11:7	0	R	Read-only constant 0, writing to this field is ignored.
PALEN-1:12	PPN	RW	**P**hysical **P**age **N**umber (PPN) of the page table.
60:PALEN	0	R	Read-only constant 0, writing to this field is ignored.
61	NR	RW	**N**on-**R**eadable bit (NR) of the page table entry.
62	NX	RW	**N**on-e**X**ecutable bit (NX) of the page table entry.
63	RPLV	RW	**R**estricted **P**rivilege **L**e**V**el enable (RPLV) of the page table. When RPLV=0, the page table entry can be accessed by any program whose privilege level is not lower than PLV; when RPLV=1, the page table entry can only be accessed by programs whose privilege level is equal to PLV.

Table 37. Definition of TLB entry low order bits in LA32

Bits	Name	Read/Write	Description
0	V	RW	**V**alid bit (V) of the page table entry.
1	D	RW	**D**irty bit (D) of the page table entry.
3:2	PLV	RW	**P**rivilege **L**e**V**el (PLV) of the page table entry.

Bits	Name	Read/Write	Description
5:4	MAT	RW	**M**emory **A**ccess **T**ype (MAT) of the page table entry.
6	G	G	**G**lobal flag bit (G) of the page table entry. When executing the TLBFILL and TLBWR instructions, the G bit in TLBELO0 and TLBELO1 is 1 only if both bits are 1. The G bit of the page table entry filled into the TLB will be 1 only when the G bit in both TLBELO0 and TLBELO1 is 1. When executing the TLBRD instruction, when the G bit of the TLB table entry read is 1, the G bits in TLBELO0 and TLBELO1 are set to 1 at the same time.
7	0	R	Read-only constant 0, writing to this field is ignored.
31:8	PPN	RW	**P**hysical **P**age **N**umber (PPN) of the page table.

7.5.4. Address Space Identifier (ASID)

This register contains the **A**ddress **S**pace **ID**entifier (ASID) information for access operations and TLB-related instructions. The length of the ASID may increase further as the architecture specification evolves, and this information is given directly to facilitate software to specify the length of the ASID.

Table 38. Definition of address space identifier register

Bits	Name	Read/Write	Description
9:0	ASID	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 the load/store instructions. When executing the TLBSRCH, TLBCLR and INVTLB instructions, it is used as the ASID key value information for querying the TLB. When executing the TLBWR or TLBFILL instructions, the value written to the ASID field of the TLB table entry is derived from this. The contents of the ASID field read from the TLB table entry when executing the TLBRD instruction are recorded here.
15:0	0	R	Read-only constant 0, writing to this field is ignored.
23:16	ASIDBITS	R	The length of the ASID field. It is directly equal to the value of this field.
31:24	0	R0	Reserved field. Return 0 if read this field and software is not allowed to change its value.

7.5.5. Page Global Directory Base Address for Lower Half Address Space (PGDL)

This register is used to configure the base address of the global directory for the lower half address space.

It is required that the base address of the global directory must be aligned to a `4KB` bound address.

This register also contains the information related to the `PS` and `P` fields in the TLB table entry when executing the TLB-related instructions.

Table 39. Definition of page global directory base address for lower half address space register

Bits	Name	Read/Write	Description
`11:0`	`0`	R	Read-only constant `0`, writing to this field is ignored.
`GRLEN-1:12`	`Base`	RW	The base address of the global directory in the lower half address space. By lower half address space, it means that the `[VALEN-1]` bit of the virtual address is equal to `0`.

7.5.6. Page Global Directory Base Address for Higher Half Address Space (PGDH)

This register is used to configure the base address of the global directory for the higher half address space. The base address of the global directory must be aligned to the `4KB` bound address, so the lowest `12` bits of this register are not configurable by software and are read-only constant `0`.

Table 40. Definition of page global directory base address for higher half address space register

Bits	Name	Read/Write	Description
`11:0`	`0`	R	Read-only constant `0`, writing to this field is ignored.
`GRLEN-1:12`	`Base`	RW	The base address of the global directory in the high half address space. By higher half address space, it means that the `[VALEN-1]` bit of the virtual address is equal to `1`.

7.5.7. Page Global Directory Base Address (PGD)

This register is a read-only register, whose content is the global directory base address information corresponding to the bad virtual address in the current context.

Table 41. Definition of page global directory base address register

Bits	Name	Read/Write	Description
`11:0`	`0`	R	Read-only constant `0`, writing to this field is ignored.
`GRLEN_1:12`	`Base`	R	If the highest bit of the bad virtual address in the current context is `0`, the return value of reading is equal to the `Base` field of `CSR.PGDL`; otherwise, the read return value is equal to the `Base` field of `CSR.PGDH`. When `CSR.TLBRERA.IsTLBR=0`, the bad virtual address information in the current context is located in `CSR.BADV`; otherwise, the bad virtual address information is located in `CSR.TLBRBADV`.

7.5.8. Page Walk Controller for Lower Half Address Space (PWCL)

The information in this register and the `CSR.PWCH` register together define the page table structure used in the operating system. This information will be used to instruct software or hardware to perform page table walking. See Multi-level Page Table Structure Supported by page walking for an illustration of the page table structure and walking process.

PWCL is implemented in LA32 only, for which the PWCL register must contain all the information needed to describe the page table structure, resulting in the last page table and the lowest two levels of the directory starting at no more than 32 bits, a restriction that still exists in LA64.

Table 42. Definition of page walk controller for lower half address space register

Bits	Name	Read/Write	Description
4:0	PTbase	RW	The start address of the last page table.
9:5	PTwidth	RW	The number of index bits of the last level page table.
14:10	Dirl_base	RW	The starting address of the lowest level directory.
19:15	Dirl_width	RW	The number of index bits of the lowest level directory. 0 means there is no such level.
24:20	Dir2_base	RW	The starting address of the next lower level directory.
29:25	Dir2_width	RW	The number of index bits of the next lowest level directory. 0 means there is no such level.
31:30	PTEWidth	RW	The length of each page table entry in the memory. 0 - 64 bit; 1 - 128 bit; 2 - 192 bit; 3 - 256 bit.

7.5.9. Page Walk Controller for Higher Half Address Space (PWCH)

This register and the information in the CSR.PWCL register together define the page table structure used in the operating system. This information will be used to instruct software or hardware to perform page table walking. See Multi-level Page Table Structure Supported by page walking for an illustration of the page table structure and walking process.

This register is only defined in LA64.

Table 43. Definition of page walk controller for higher half address space register

Bits	Name	Read/Write	Description
5:0	Dir3_base	RW	The starting address of the next higher level directory.
11:6	Dir3_width	RW	The number of index bits of the next higher level directory. 0 means there is no such level.
17:12	Dir4_base	RW	The starting address of the highest level directory.
23:18	Dir4_width	RW	The number of index bits of the highest level directory. 0 means there is no such level.
31:24	0	R0	Reserved field. Return 0 if read this field, and the software does not allow to change its value.

7.5.10. STLB Page Size (STLBPS)

This register is used to configure the size of the page in the STLB.

Table 44. Definition of STLB page size register

Bits	Name	Read/Write	Description
5:0	PS	RW	The STLB is a power of 2 of the page size. For example, if the page size is 16KB, then PS=0xE.

Bits	Name	Read/Write	Description
31:6	0	R0	Reserved field. Return 0 if read this field and software is not allowed to change its value.

7.5.11. TLB Refill Exception Entry Base Address (TLBRENTRY)

This register is used to configure the entry base address of the TLB refill exception. Since the processor core will enter direct address translation mode after the TLB refill exception is triggered, the entry base address filled here should be a physical address.

Table 45. Definition of TLB refill exception entry base address register in LA64

Bits	Name	Read/Write	Description
11:0	0	R	Read-only constant 0, writing to this field is ignored.
PALEN-1:12	PPN	RW	The [PALEN-1:12] bits of the entry base address of the TLB refill exception entry base address. The address filled in here by the system software should be the physical address.
63:PALEN	0	R	Read-only constant 0, writing to this field is ignored.

Table 46. Definition of TLB refill exception entry base address register in LA32

Bits	Name	Read/Write	Description
11:0	0	R	Read-only constant 0, writing to this field is ignored.
31:12	PPN	RW	The [31:12] bits of the entry base address of the TLB refill exception entry base address. The address filled in here by the system software should be the physical address.

7.5.12. TLB Refill Exception Bad Virtual Address (TLBRBADV)

This register is used to record the bad virtual address that triggered the TLB refill exception.

Table 47. Definition of TLB refill exception bad virtual address register

Bits	Name	Read/Write	Description
GRLEN-1:0	VAddr	RW	When the TLB refill exception is triggered, the hardware records the bad virtual address here. For LA64, in this case, if the privilege level that triggered the exception is in 32-bit address mode, then the high 32 bits of the recorded virtual address will be set to 0.

7.5.13. TLB Refill Exception Return Address (TLBRERA)

This register is used to record the PC of the instruction that triggered the TLB refill exception. In addition, this register contains flag bits to identify the current exception as a TLB refill exception.

Table 48. Definition of TLB refill exception program counter register

Bits	Name	Read/Write	Description
0	IsTLBR	RW	1 indicates that it is currently in the context of TLB refill exception processing. The hardware sets this bit to 1 when a TLB refill exception is triggered. When this bit is 1, execution of the ERTN instruction will clear it to 0 only if CSR.MERRCTL.IsMERR=0, otherwise it remains unchanged. Because the architecture defines a separate set of CSRs for TLB refill exceptions, when this bit is 1. • When ERTN returns, the information used to recover CSR.CRMD will come from CSR.TLBRPRMD; • ERTN return address will come from CSR.TLBRERA; • The table entries to be written by TLBWR and TLBFILL instructions will come from CSR.TLBREHI, CSR.TLBELO0 and CSR.TLBELO1; • TLBSRCH instruction queries information from CSR.TLBREHI; • The bad virtual address required for LDDIR and LDPTE instruction execution will come from CSR.TLBRBADV.
1	0	R	Read-only constant 0, writing to this field is ignored.
GRLEN-1:2	PC	RW	Record the [GRLEN-1:2] bits of the PC of the instruction that triggered the TLB refill exception. When the execution of ERTN instruction returns from the TLB refill exception handler (at this time, this register IsTLBR=1 and CSR.MERRCTL.IsMERR=0).

7.5.14. TLB Refill Exception Data Save Register (TLBRSAVE)

This register is used to store data temporarily for the system software. Each dava save register can hold the data of one general-purpose register.

The reason for the additional SAVE register for TLB refill exception processing is to address the case where a TLB refill exception is triggered during the processing of exceptions except the TLB refill exception.

Table 49. Definition of TLB refill exception data save register

Bits	Name	Read/Write	Description
GRLEN-1:0	Data	RW	Data for software to read and write only. The hardware does not modify the contents of this field except for the execution of CSR instructions.

7.5.15. TLB Refill Exception Entry Low-order Bits (TLBRELO0, TLBRELO1)

The TLBRELO0/TLBRELO1 registers are used to store the information related to the physical page number of the low-order bits of the TLB table entry during executing the TLB-related instructions (when the TLB refill exception context CSR.TLBRERA.IsTLBR=1). The format of TLBRELO0/TLBRELO1 registers and the meaning of each field are the same as TLBELO0/TLBELO1 registers.

However, the TLBRELO0/TLBRELO1 registers are not an exact copy of the TLBELO0/TLBELO1 registers in the case of CSR.TLBRERA.IsTLBR=1. This is reflected in two points:

- Regardless of the value of CSR.TLBRERA.IsTLBR, the TLBRD instruction updates only the TLBELO0 /TLBELO1 registers.
- Regardless of the value of CSR.TLBRERA.IsTLBR, the LDPTE instruction updates only the TLBRELO0 /TLBRELO1 registers.

Table 50. Definition of TLB refill exception entry low order bits register in LA64

Bits	Name	Read/Write	Description
0	V	RW	**V**alid bit (V) of the page table entry.
1	D	RW	**D**irty bit (D) of the page table entry.
3:2	PLV	RW	**P**rivilege **L**e**V**el (PLV) of the page table entry.
5:4	MAT	RW	**M**emory **A**ccess **T**ype (MAT) of the page table entry.
6	G	RW	**G**lobal flag bit (G) of the page table entry. When executing the TLBFILL and TLBWR instructions, the G bit of the page table entry filled into the TLB is 1 only when the G bit in both TLBELO0 and TLBELO1 is 1.
11:7	0	R	Read-only constant 0, writing to this field is ignored.
PALEN-1:12	PPN	RW	**P**hysical **P**age **N**umber (PPN) of the page table.
60:PALEN	0	R	Read-only constant 0, writing to this field is ignored.
61	NR	RW	**N**on-**R**eadable bit (NR) of the page table entry.
62	NX	RW	**N**on-e**X**ecutable bit (NX) of the page table entry.
63	RPLV	RW	**R**estricted **P**rivilege **L**e**V**el enable (RPLV) for the page table. When RPLV=0, the page table entry can be accessed by any program whose privilege level is not lower than PLV; when RPLV=1, the page table entry can only be accessed by programs whose privilege level is equal to PLV.

Table 51. Definition of tlb refill exception entry low order bits register in LA32

Bits	Name	Read/Write	Description
0	V	RW	**V**alid bit (V) of the page table entry.
1	D	RW	**D**irty bit (D) of the page table entry.
3:2	PLV	RW	**P**rivilege **L**e**V**el of the page table entry (PLV).
5:4	MAT	RW	**M**emory **A**ccess **T**ype (MAT) of the page table entry.

Bits	Name	Read/Write	Description
6	G	RW	**G**lobal flag bit (G) of the page table entry. When executing TLBFILL and TLBWR instructions, the G bit of the page table entry filled into the TLB is 1 only when the G bits in both TLBELO0 and TLBELO1 are 1.
11:7	0	R	Read-only constant 0, writing to this field is ignored.
31:12	PPN	RW	**P**hysical **P**age **N**umber (PPN) of the page table.

7.5.16. TLB Refill Exception Entry High-order Bits (TLBREHI)

When in the TLB refill exception context (CSR.TLBRERA.IsTLBR=1), the TLBREHI register stores the information related to the physical page number of the low-order bits of the TLB table entry during executing TLB-related instructions, etc. The format of the TLBREHI register and the meaning of each field are the same as the TLBEHI register.

However, the TLBREHI register is not an exact replica of the TLBEHI register in the case of CSR.TLBRERA.IsTLBR=1. This is reflected in:

- Regardless of the value of CSR.TLBRERA.IsTLBR equals, the execution of the TLBRD instruction updates only the TLBEHI register.

Table 52. Definition of TLB refill exception entry high order bits register in LA64

Bits	Name	Read/Write	Description
5:0	PS	RW	Page size specified by TLB refill exception. That is, when CSR.TLBRERA.IsTLBR=1, when executing TLBWR and TLBFILL instructions and the value of the PS field of the written TLB table entry comes from this.
12:0	0	R	The read-only constant is 0, and writes are ignored.
VALEN-1:13	VPPN	RW	When CSR.TLBRERA.IsTLBR=1, the value of VPPN used for querying TLB when executing TLBSRCH instruction, and the value of VPPN field of TLB table entry written when executing TLBWR and TLBFILL instructions come from here. When a TLB refill exception is triggered, the [VALEN-1:13] bits of the virtual address that triggered the exception are recorded here.
63:VALEN	Sign_Ext	R	The return value read from these bits is a signed extension of the highest bits of the VPPN field; writing to these bits is ignored.

Table 53. Definition of tlb refill exception entry high order bits register in LA32

Bits	Name	Read/Write	Description
12:0	0	R	Read-only is constant to 0, and writes are ignored.
31:13	VPPN	RW	With CSR.TLBRERA.ISTLBR=1, the VPPN value used to query the TLB when executing the TLBSRCH instruction, and the value of the VPPN field written to the TLB table entry when executing the TLBWR and TLBFILL instructions come from here. When a TLB refill exception is triggered, the [31:13] bits of the virtual address that triggered the exception are recorded here.

7.5.17. TLB Refill Exception Pre-exception Mode Information (TLBRPRMD)

When a TLB refill exception is triggered, the hardware saves the processor core's privilege level, Guest mode, global interrupt enable bit, and watchpoint enable bit into this register at that time, which is used to restore the processor core to the field when the exception returns.

Table 54. Definition of TLB refill exception pre-exception mode information register

Bits	Name	Read/Write	Description
1:0	PPLV	RW	When the TLB refill exception is triggered, the hardware records the old value of the PLV field in CSR.CRMD in this field. When CSR.TLBRERAIsTLBR=1, the hardware restores the value of this field to the PLV field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
2	PIE	RW	When the TLB refill exception is triggered, the hardware records the old value of the IE field in the CSR.CRMD in this field. When CSR.TLBRERAIsTLBR=1, the hardware restores the value of this field to the IE field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
3	0	R	If the virtualization extension is not implemented, this bit is read-only constant to 0 and writes are ignored.
4	PWE	RW	When the TLB refill exception is triggered, the hardware records the old value of the WE field in the CSR.CRMD in this field. When CSR.TLBRERAIsTLBR=1, the hardware restores the value of this field to the WE field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
31:5	0	R0	Reserved field. Return 0 if read this field, and software is not allowed to change its value.

7.5.18. Direct Mapping Configuration Window n (DMW0-DMW3)

This -group sender is involved in completing the direct mapping address translation mode. See Direct Mapped Address Translation Mode for more information about this address translation mode.

Table 55. Definition of direct mapping configuration window n register in LA64

Bits	Name	Read/Write	Description
0	PLV0	RW	1 indicates that the configuration of this window can be used for direct mapping address translation at the PLV0 privilege level.
1	PLV1	RW	1 indicates that the configuration of this window can be used for direct mapping address translation at the PLV1 privilege level.
2	PLV2	RW	1 indicates that the configuration of this window can be used for direct map address translation at the PLV2 privilege level.
3	PLV3	RW	1 indicates that the configuration of this window can be used for direct mapping address translation at the PLV3 privilege level.
5:4	MAT	RW	The virtual address falls under the memory access type of the access operation in this mapping window.
59:6	0	R0	Reserved field. Return 0 if read this field and software is not allowed to change its value.

Bits	Name	Read/Write	Description
63:60	VSEG	RW	The [63:60] bits of the virtual address of the direct mapping window.

Table 56. Definition of direct mapping configuration window n register in LA32

Bits	Name	Read/Write	Description
0	PLV0	RW	1 indicates that the configuration of this window can be used for direct mapping address translation at the PLV0 privilege level.
1	PLV1	RW	1 indicates that the configuration of this window can be used for direct mapping address translation at the PLV1 privilege level.
2	PLV2	RW	1 indicates that the configuration of this window can be used for direct map address translation at the PLV2 privilege level.
3	PLV3	RW	1 indicates that the configuration of this window can be used for direct mapping address translation at the PLV3 privilege level.
5:4	MAT	RW	The virtual address falls under the memory access type of the access operation in this mapping window.
24:6	0	R0	Reserved field. Return 0 if read this field and software is not allowed to change its value.
27:25	PSEG	RW	The [31:29] bits of the physical address of the direct mapping window.
28	0	R0	Reserved field. Return 0 if read this field and software is not allowed to change its value.
31:29	VSEG	RW	The [31:29] bits of the virtual address of the direct mapping window.

7.6. Control and Status Registers Related to Timers

7.6.1. Timer Identity (TID)

Each timer in the processor has a unique identifiable number, which is configured by the software in this register. Each timer also uniquely corresponds to a timer, and when the software reads the timer value using the RDTIME instruction, the timer ID number that is returned along with it is the corresponding timer number.

Table 57. Definition of timer identity register

Bits	Name	Read/Write	Description
31:0	TID	RW	Timer number. It can be configured via software. During a processor core reset, the hardware can reset it to the same value as the CoreID in CSR.CPUID.

7.6.2. Timer Configuration (TCFG)

This register is the interface to the software configuration timer. The number of valid bits of the timer is determined by the implementation, so the length of the TimeVal field in this register will change accordingly.

Table 58. Definition of timer configuration register

Bits	Name	Read/Write	Description
0	En	RW	Timer enable bit. Only when this bit is 1, the timer will perform countdown self decrement and set up the timing interrupt signal when it decrements to 0 value.
1	Periodic	RW	Timer cycle mode control bit. If this bit is 1, when the timer decreases to 0, the timer will set up the timer interrupt signal and reload the timer to the initial value configured in the TimeVal field, and then continue to decrement itself in the next clock cycle. If this bit is 0, the timer will stop counting until the software configures the timer again when the countdown reaches 0.
n-1:2	InitVal	RW	The initial value of the timer countdown self decrement count. This initial value must be an integer multiple of 4. The hardware will automatically fill in the lowest bit of the field value. Two bits of 0 are added before it is used.
GRLEN-1:n	0	R	Read-only constant 0, writing to this field is ignored.

7.6.3. Timer Value (TVAL)

The software can read this register to know the current count value of the timer. The number of valid bits of the timer is determined by the implementation, so the length of the TimeVal field in this register will also change.

Table 59. Definition of timer value register

Bits	Name	Read/Write	Description
n-1:0	TimeVal	R	The count value of the current timer.
GRLEN-1:n	0	R	Read-only constant 0, writing to this field is ignored.

7.6.4. Counter Compensation (CNTC)

This register can be configured by the software to correct the timer's readout value. The final readout value will be the original timer count value plus the timer compensation value. It is important to note that configuring this register does not directly change the timer's count value.

In LA32, this register is 32-bit and its value will be sign extended to 64 bits and then added to the original counter value.

Table 60. Definition of counter compensation register

Bits	Name	Read/Write	Description
GRLEN-1:0	Compensation	RW	Software-configurable counter compensation values.

7.6.5. Timer Interrupt Clearing (TICLR)

The software clears the timed interrupt signal set by the timer by writing 1 to bit 0 of this register.

Table 61. Definition of timer interrupt clearing register

Bits	Name	Read/Write	Description
0	CLR	W1	When 1 is written to this bit, the clock interrupt flag is cleared. The value read from this register is always 0.
31:1	0	R0	Reserved field. Return 0 if read this field and software is not allowed to change its value.

7.7. Control and Status Registers Related to RAS

7.7.1. Machine Error Controller (MERRCTL)

Since the timing of machine error exceptions cannot be predicted and controlled by the software, a separate set of CSRs is defined for machine error exceptions in order not to destroy any other site when triggering machine error exceptions, which is used by the system software to save and restore other sites. This set of independent CSRs except MERRERA and MERRSAVE, the rest are concentrated in MERRCTL register.

Table 62. Definition of machine error controller register

Bits	Name	Read/Write	Description
0	IsMERR	R	1 indicates that it is currently in the context of machine error exception processing. The hardware sets this bit to 1 when a machine error exception is triggered. When this bit is 1, execution of the ERTN instruction will clear it to 0. Because the architecture defines a separate set of CSRs for machine error exceptions, when this bit is 1, * when ERTN returns, information used to restore the CSRs will come from PPLV, PLV and so on of this field; * when ERTN returns, address information will come from CSR.MERRERA.
1	Repairabl e	RW	1 means that the hardware can automatically fix machine errors that occur, so the exception handler can return directly without any processing.
3:2	PPLV	RW	When a machine error exception is triggered, the hardware records the old value of the PLV field in CSR.CRMD in this field. When the IsMERR of this register is 1, the hardware returns from the exception handler by executing the ERTN instruction. The hardware restores the value of this field to the PLV field of CSR.CRMD.
4	PIE	R	When a machine error exception is triggered, the hardware records the old value of the IE field in CSR.CRMD in this field. When IsMERR of this register is 1, the hardware restores the value of this field to the IE field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.

Bits	Name	Read/Write	Description
5	0	RW	If the virtualization expansion is not implemented, this field is read-only constant 0 and writing to this field is ignored.
6	PWE	RW	When a machine error exception is triggered, the hardware records the old value of the WE field in CSR.CRMD in this field. When IsMERR of this register is 1, the hardware restores the value of this field to the WE field in CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
7	PDA	RW	When a machine error exception is triggered, the hardware records the old value of the DA field in the CSR.CRMD in this field. When IsMERR of this register is 1, the hardware restores the value of this field to the DA field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
8	PPG	RW	When a machine error exception is triggered, the hardware records the old value of the PG field in the CSR.CRMD in this field. When IsMERR of this register is 1, the hardware restores the value of this field to the PG field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
10:9	PDATF	RW	When a machine error exception is triggered, the hardware records the old value of the DATF field in the CSR.CRMD in this field. When IsMERR of this register is 1, the hardware restores the value of this field to the DATF field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
12:11	PDATM	RW	When a machine error exception is triggered, the hardware records the old value of the DATM field in the CSR.CRMD in this field. When IsMERR of this register is 1, the hardware restores the value of this field to the DATM field of CSR.CRMD when the ERTN instruction is executed to return from the exception handler.
15:13	0	R0	Reserved field. Return 0 if read this field and software must write 0, or mask out this field by csr mask write.
23:16	Cause	R	Machine error type code. Currently only the 0x1 value is defined for Cache checksum errors. The rest of the encoded values are reserved.
31:24	0	R0	Reserved field. Return 0 if read this field and software is not allowed to change its value.

7.7.2. Machine Error Information (MERRINFO1, MERRINFO2)

When a machine error exception is triggered, the hardware will store more information related to that error into these two registers for system software diagnostic purposes. The format and the meaning of each field are defined by the implementation.

7.7.3. Machine Error Exception Entry Base Address (MERRENTRY)

This register is used to configure the entry base address of the machine error exception. Since the processor core will enter the direct address translation mode after the machine error exception is triggered, the entry base address filled here should be the physical address.

Table 63. Definition of machine error exception entry base address register in LA64

Bits	Name	Read/Write	Description
`11:0`	`0`	R	Read-only constant `0`, writing to this field is ignored.
`PALEN-1:12`	`PPN`	RW	The `[PALEN-1:12]` bits of the entry base address of the machine error exception. The address filled in here by the system software should be the physical address.
`63:PALEN`	`0`	R	Read-only constant `0`, writing to this field is ignored.

Table 64. Definition of machine error exception entry base address register in LA32

Bits	Name	Read/Write	Description
`11:0`	`0`	R	Read-only constant `0`, writing to this field is ignored.
`31:12`	`PPN`	RW	The `[31:12]` bits of the entry base address of the machine error exception. The address entered here by the system software should be a physical address.

7.7.4. Machine Error Exception Return Address (MERRERA)

This register is used to record the `PC` of the instruction that triggered the machine error exception.

Table 65. Definition of machine error exception return address register

Bits	Name	Read/Write	Description
`GRLEN-1:0`	`PC`	RW	The `PC` of the instruction that triggered the machine error exception is recorded. The value stored here is used as the return address when the `ERTN` instruction is executed to return from the machine error exception handler (when `CSR.MERRCTL.IsMERR=1`).

7.7.5. Machine Error Exception Data Save Register (MERRSAVE)

This register is used to store data temporarily for the system software. Each dava save register can hold the data of one general-purpose register.

The reason for the additional `SAVE` register for the machine error exception handler is that the timing of the machine error exception cannot be predicted and controlled by the software, and it may occur during the processing of any other exception.

Table 66. Definition of machine error exception data save register

Bits	Name	Read/Write	Description
`GRLEN-1:0`	`DATA`	RW	Data for software to read and write only. The hardware will not modify the contents of this field except for the execution of CSR instructions.

7.8. Control and Status Registers Related to Performance Monitoring

LoongArch defines a hardware performance monitoring mechanism to support software performance analysis. The main body of this mechanism is a series of performance monitors. At least one performance monitor is implemented, and up to 32 monitors are implemented, the number is determined by the implementation. The software can determine how many performance monitors are available by reading the `CPUCFG.6.PMNUM[bit7:4]`.

Each performance monitor contains two CSRs: a **P**erformance **M**onitoring **C**on**F**i**G**uration register (`PMCFG`) and a **P**erformance **M**onitoring **C**ou**NT**er register (`PMCNT`).

All CSRs related to performance monitoring are alternately addressed starting at address `0x200`, with the `n`th performance monitoring configuration register at address `0x200+n`, and the `n`th performance monitoring counter at address `0x201+n`. The format of all performance monitoring configuration registers is the same, as described in Performance Monitor Configuration n (`PMCFG`); the format of all performance monitoring counters is the same, as described in Performance Monitor Overall Counter n (`PMCNT`).

7.8.1. Performance Monitor Configuration n (`PMCFG`)

Table 67. Definition of performance monitor configuration n register

Bits	Name	Read/Write	Description
`9:0`	`EvCode`	RW	The event number of the performance event being monitored. The definition of event numbers is divided into two parts, a part whose meaning is specified in the architecture specification and must be implemented by all processors compatible with this architecture, and a remaining part whose meaning is implementation specific and is defined by the processor's implementer.
`15:10`	`0`	R0	Reserved fields. Return `0` if read this field, and software is not allowed to change its value.
`16`	`PLV0`	RW	PLV0 privilege level enables counting for this performance monitor. `1` - count, `0` - stop.
`17`	`PLV1`	RW	PLV1 privilege level enables counting for this performance monitor. `1` - count, `0` - stop.
`18`	`PLV2`	RW	PLV2 privilege level enables counting for this performance monitor. `1` - count, `0` - stop.
`19`	`PLV3`	RW	Count enable for this performance monitor at the PLV3 privilege level. `1` - count, `0` - stop.
`20`	`PMIEn`	RW	Performance monitoring count overflow interrupt enable bit for this performance monitor. `1` - enable, `0` - disable.
`22:21`	`0`	R	If the virtualization expansion is not implemented, this field is read-only constant `0` and writing to this field is ignored.
`31:23`	`0`	R0	Reserved field. Return `0` if read this field and software is not allowed to change its value.

7.8.2. Performance Monitor Overall Counter n (`PMCNT`)

Table 68. Definition of performance monitor overall counter n register

Bits	Name	Read/Write	Description
GRLEN-1:0	Count	RW	The counter is incremented by 1 for each performance event monitored by the performance monitor. If the performance monitor has enabled the performance monitoring count overflow interrupt, and when the highest bit of Count is 1, the interrupt is triggered. This also means that the software can cancel the interrupt by clearing the highest bit of Count to 0.

7.9. Control and Status Registers Related to Watchpoints

LoongArch defines hardware watchpoint functions for fetch and load/store operations. After the software configures the watchpoints for fetch and load/store, the processor hardware will monitor the access addresses of the fetch and load/store operations and trigger a watchpoint exception when the watchpoint setting conditions are met.

The control and status registers associated with the watchpoints are used as interfaces for software to configure the watchpoints for fetch and load/store operations. Load/store watchpoints and fetch watchpoints each have a similar layout of control and status registers, a register for the overall configuration of all watchpoints, a register for the status of all watchpoints, and the four registers. The address of the overall configuration register of the load/store watchpoint is 0x300, the address of the overall status register of the load/store watchpoint is 0x301, and the addresses of the four configuration registers from 1 to 4 of the nth load/store watchpoint are 0x310+8n, 0x311+8n, 0x312+8n, and 0x313+8n, respectively. The address of the overall configuration register of the fetch instruction watchpoint is 0x380, the address of the overall status register of the fetch instruction watchpoint is 0x381, and the addresses of the four configuration registers 1-4 of the nth fetch instruction watchpoint are 0x390+8n, 0x391+8n, 0x392+8n, 0x393+8n in order.

The maximum number of load/store watchpoints and fetch instruction watchpoints is 14 each, and the actual number is determined by the implementation. The software can determine how many hardware watchpoints can be used by reading the values of CSR.MWPC.Num and CSR.FWPC.Num.

7.9.1. Memory Load/Store Watchpoint Overall Controller (MWPC)

This register contains configuration information to inform the software of the exact number of load/store watchpoints.

It is important to note that the global enable control signal for all watchpoints is in the WE bit of CSR.CRMD.

Table 69. Definition of memory load/store watchpoint overall controller register

Bits	Name	Read/Write	Description
5:0	Num	R	The number of load/store watchpoints.
19:16	0	R	If no virtualization extension is implemented, the field is read-only constant to 0 and writes are ignored.
31:20	0	R0	Reserved field. Reads return 0 and the software does not allow to change its value.

7.9.2. Memory Load/Store Watchpoint Overall Status (MWPS)

Table 70. Definition of memory load/store watchpoint overall status register

Bits	Name	Read/Write	Description
`n-1:0`	`Status`	RW1[2]	The hit status of the load/store watchpoint. It corresponds to the watchpoint one by one, and bit `i` corresponds to watchpoint `i`. When an address with a load/store operation hits a watchpoint, the corresponding bit is set to 1. The hardware does not clear the bits in this field except during a reset. The software can only clear them by writing 1, writing `0` is ignored.
`15:n`	`0`	R	Read-only constant `0`, writing to this field is ignored.
`16`	`Skip`	RW	The software notifies the hardware to ignore the next load/store watchpoint hit by setting this location to 1. By ignoring, it means that neither the corresponding bit in the Stauts field of this register is set to 1 nor the watchpoint exception is triggered. This function can avoid endlessly triggering the same watchpoint repeatedly without canceling it, thus simplifying the handling of watchpoint exceptions. When the Skip bit is 1, if the hardware encounters a loadjstore hit, it will ignore the hit and clear the Skip bit to 0. This means that each time the software sets the Skip bit to 1, the hardware will ignore at most one hit. This feature also causes the software to write 1 to this bit and then read out the value which may not be 1. This Skip bit corresponds to all load/store watchpoints. If the software modifies the configuration of the breakpoint and replaces it, do not set this bit, or even write `0` to clear it for safety reasons.
`31:17`	`0`	R	Read-only constant `0`, writing to this field is ignored.

7.9.3. Memory Load/Store Watchpoint n Configuration (`MWPnCFG1`-`MWPnCFG4`)

The information contained in the configuration 1 to 3 registers of each load/store watchpoint is used directly for the comparison judgment of the watchpoint check. Assuming that the address of the operation to be compared is maddr and the byte range is mbyten, the process of determining the hit of each watchpoint is as follows:

1. If `CSR.CRMD.WE=0`, the judgment is terminated, otherwise turn 2;
2. If the current is not in debug mode but the DMOnly bit of `MWPCFG3` is equal to 1, the judgment is terminated, otherwise turn to 3;
3. If the bit corresponding to the current privilege level in `PLV0`-`PLV3` of `MWPCFG3` is equal to `0`, the judgment is terminated, otherwise turn to 4;
4. If the operation is a load operation but the `LoadEn` bit in `MWPCFG3` is equal to `0`, or the operation is a store operation but the `StoreEn` bit in `MWPCFG3` is equal to `0`, the judgment is terminated, otherwise go to 5;
5. If the `LCL` bit in `MWPCFG3` is equal to 1, but the `CSR.ASID.ASID` is not equal to the `ASID` in `MWPCFG4`, the judgment is terminated, otherwise go to 6;
6. If `(maddr & (~MWPCFG2.Mask)) != (MWPCFG1.VAaddr & (~MWPCFG2.Mak))`, that is, the

address comparison is not equal, the judgment terminates, otherwise turn 7;

7. If `(~bytemask[7:0] & mbyten[7:0])` is equal to all `0` values, the judgment is terminated, otherwise the watchpoint is considered to be hit.

The concepts of `mbyten` and `bytemask`, which appear in the description of the judgment process above, are explained further below.

`mbyten` represents the bytes involved in the operation, which is an 8-bit bit vector whose value is related to the type of load/store operation and the low value of the address, as defined in the table:

Table 71. Definition of load/store watchpoint judgment process `mbyten`

Intsruction Name	`maddr[2:0]`							
`0`	1	2	3	4	5	6	7	

`LD[X].B[U], ST[X].B,`
 	
`LD{GT/LE}.B, ST{GT/LE}.B`
	`0x01`	`0x02`	`0x04`	`0x08`	`0x10`	`0x20`	`0x40`	`0x80`

`LD[X].H[U], ST[X].H`
 	
`LD{GT/LE}.H, ST{GT/LE}.H`
	`0x30`		`0x0C`		`0x30`		`0xC0`	

`LD[X].W[U], ST[X].W,`
 	
`LD{GT/LE}.W, ST{GT/LE}.W,`
 	
`LDPTR.W, STPTR.W,`
 	
`LL.W, SC.W,`
 	
`AM{SWAP/ADD/AND/OR/XOR/MAX/MIN}[.DB].W,`
 	
`AM{MAX/MIN}[_DBI].WU,`
 	
`FLD[X].S, FST[X]S,`
 	
`FLD{GT/LE}.S, FST{GT/LE}.S`
	`0x0F`				`0xF0`			

Intsruction Name	maddr[2:0]
LD[X].D, ST[X].D, LD{GT/LE}.D, ST[GT/LE].D, LDPTR.D, STPTR.D, LL.D, SC.D, AM{SWAP/ADD/AND/OR/XOR/MAX/MIN}[_DB].D, AM{MAX/MIN}[_DB].DU, FLD[X].D, FST[X].D, FLD{GT/LE}.D, FST{GT/LE}.D	0xFF

bytemask indicates which bytes do not participate in the comparison mask when watchpoint comparison, which is an 8-bit bit vector whose value is related to the low bit of VAddr in MWPCFG1 and Size in MWPCF`G3, as defined as shown.

Table 72. Definition of load store watchpoint bytemask

MWPCFG3.Size	MWPCFG1.Vaddr[2:0]							
0	1	2	3	4	5	6	7	
0b00	0x00							
0b01	0xF0				0x0F			
0b10	0xFC		0xF3		0xCF		0x3F	
0b11	0xFE	0xFD	0xFB	0xF7	0xEF	0xDF	0xBF	0x7F

Table 73. Definition of memory load/store watchpoint n configuration 1 register

Bits	Name	Read/Write	Description
GRLEN-1:0	VAddr	RW	The virtual address to be compared for this load/store watchpoint.

Table 74. Definition of memory load/store watchpoint n configuration 2 register

Bits	Name	Read/Write	Description
GRLEN-1:0	Mask	RW	Mask bit for address comparison for this load/store watchpoint. If bit i (0 ≤ i < GRLEN) is 1, it means that bit i of the address is not involved in the comparison.

Table 75. Definition of memory load/store watchpoint n configuration 3 register

Bits	Name	Read/Write	Description
0	DMOnly	RW	A bit of 1 indicates that the load/store watchpoint is only available in debug mode. Here “available” contains two meanings: first, the configuration register of the watchpoint can be modified by software in this mode, and second, the check hit of the watchpoint will trigger the watchpoint exception and mark the status of the watchpoint only in this mode. This bit can only be modified in debug mode (CSR.DBG.DM=1). This means that the (Host) software running in debug mode has the priority to use the watchpoint.
1	PLV0	RW	This watchpoint triggers the enable of the watchpoint exception at the PLV0 privilege level. 1 - enable, 0 - disable.
2	PLV1	RW	The watchpoint triggers the watchpoint exception enable at the PLV1 privilege level. 1 - enable, 0 - disable.
3	PLV2	RW	The watchpoint triggers the enable of the watchpoint exception at the PLV2 privilege level. 1 - enable, 0 - disable.
4	PLV3	RW	The watchpoint triggers the enablement of the watchpoint exception at the PLV3 privilege level. 1 - enable, 0 - disable.
6:5	0	R	If virtualization extensions are not implemented, the field is read-only constant at 0 and writes are ignored.
7	LCL	RW	1 indicates that the ASID comparison is performed during the watchpoint check
8	LoadEn	RW	1 indicates a watchpoint check for load operations, otherwise no check.
9	StoreEn	RW	1 means that a watchpoint check is performed for the store operation, otherwise, no check is performed.
11:10	Size	RW	Which bytes fall within the comparison range when the watchpoint check is performed.
31:12	0	R0	Reserved field. Return 0 if read this field, and the software does not allow to change its value.

Table 76. Definition of memory load/store watchpoint n configuration 4 register

Bits	Name	Read/Write	Description
9:0	ASID	RW	The ASID being compared
15:10	0	R	Read-only is always 0, writes are ignored.
23:16	0	R	If the virtualization extension is not implemented, the field is read-only constant to 0 and writes are ignored.
31:24	0	R	Read-only constant 0, writing to this field is ignored.

7.9.4. Fetch Watchpoint Overall Controller (FWPC)

This register contains configuration information to inform the software of the exact number of watchpoints to be fetched.

It is important to note that the global enable control signal for all watchpoints is in the WE bit of the CSR.CRMD.

Table 77. Definition of fetch watchpoint overall controller register

Bits	Name	Read/Write	Description
`5:0`	`Num`	R	the number of fetch watchpoints.
`19:16`	`0`	R	If the virtualization extension is not implemented, the field is read-only constant `0`, and writing to this field is ignored.
`31:20`	`0`	RO	Reserved field. Reads return `0` and the software does not allow to change its value.

7.9.5. Fetch Watchpoint Overall Status (`FWPS`)

Table 78. Definition of fetch watchpoint overall status register

Bits	Name	Read/Write	Description
`n-1:0`	`Status`	RW1[3]	The hit status of the surveillance point. It corresponds to the watchpoint one by one, with bit i corresponding to watchpoint `i`. When a PC with a fetch instruction hits a watchpoint, its corresponding bit is set to `1`, the hardware does not clear the bits in this field except during reset. The software can only clear them by writing `1`, writing `0` is ignored.
`15:n .`	`0`	R	Read-only constant `0`, writing to this field is ignored.
`16`	`Skip`	RW	The software notifies the hardware to ignore the next fetch point hit result by setting this location to `1`. By ignore, it means that neither the corresponding bit in the `Stauts` field of this register is set to 1 nor the watchpoint exception is triggered. This function can avoid endlessly triggering the same watchpoint repeatedly without canceling it, thus simplifying the handling of watchpoint exceptions. When the `Skip` bit is `1`, if the hardware encounters a hit on a fetch point, it will ignore the hit and clear the `Skip` bit to `0`. This means that each time the software sets the `Skip` bit to `1`, the hardware will ignore at most one hit on the point. This feature also causes the software to write 1 to this bit and then read out the value which may not be `1`. This `Skip` bit corresponds to all fetch watchpoints. If the software modifies the configuration of the breakpoint and replaces it, do not set this bit, or even write `0` to clear it for safety reasons.
`31:17`	`0`	R	Read-only constant `0`, writing to this field is ignored.

7.9.6. Fetch Watchpoint n Configuration (`FWPnCFG1`-`FWPnCFG3`)

The information contained in the configuration `1` to `3` registers of each fetch instruction watchpoint is used directly for comparison judgments of watchpoint checks. The process of judging the hit of each watchpoint is as follows:

1. If `CSR.CRMD.WE=0`, the judgment is terminated, otherwise turn `2`;
2. If the current is not in debug mode but the `DMOnly` bit of `FWPCFG3` is equal to `1`, the judgment is terminated, otherwise turn to `3`;
3. If the bit corresponding to the current privilege level in `PLV0`-`PLV3` of `FWPCFG3` is equal to `0`, judge and terminate, otherwise turn to `4`;
4. If the `LCL` bit in `FWPCFG3` is equal to `1`, but the `CSR.ASID.ASID` is not equal to the `ASID` in `FWPCFG4`, the judgment is terminated, otherwise turn `6`;
5. If `(pc & (~FWPCFG2.Mask)) != (FWPCFG1.VAddr & (~FWPCFG2.Mask))`, that is, the address comparison is not equal, the judgment is terminated, otherwise the watchpoint is considered hit.

Table 79. Definition of fetch watchpoint n configuration 1 register

Bits	Name	Read/Write	Description
`GRLEN-1:0`	`VAddr`	RW	the virtual address of the fatch watchpoint to be compared.

Table 80. Definition of fetch watchpoint n configuration 2 register

Bits	Name	Read/Write	Description
`GRLEN-1:0`	`Mask`	RW	the mask bit of the fetch watchpoint address comparison. If bit `i` (`0 ≤ i < GRLEN`) is `1`, it means that bit i of the address is not involved in the comparison.

Table 81. Definition of fetch watchpoint n configuration 3 register

Bits	Name	Read/Write	Description
`0`	`DMOnly`	RW	A bit of `1` indicates that the fetch point is only available in debug mode. Here "available" contains two meanings: First, the configuration register of the fetch watchpoint can be modified by software in this mode, and second, the check hit of the watchpoint will trigger a watchpoint exception and mark the status of the watchpoint only in this mode. This bit can only be modified in debug mode (`CSR.DBG.DM=1`). This means that the (Host) software running in debug mode has the priority to use the watchpoint.
`1`	`PLV0`	RW	This watchpoint triggers the enable of the watchpoint exception at the PLV0 privilege level. `1` - enable, `0` - disable.
`2`	`PLV1`	RW	The watchpoint triggers the watchpoint exception enable at PLV1 privilege level. `1` - enable, `0` - disable.
`3`	`PLV2`	RW	The watchpoint triggers the enable of the watchpoint exception at the PLV2 privilege level. `1` - enable, `0` - disable.
`4`	`PLV3`	RW	This watchpoint triggers the enablement of the watchpoint exception at the PLV3 privilege level. `1` - enable, `0` - disable.
`6:5`	`0`	R	If virtualization extensions are not implemented, the field is read-only constant to `0` and writes are ignored.
`7`	`LCL`	RW	`1` indicates that the comparison of ASIDs is performed during the watchpoint check.
`31:8`	`0`	R0	Reserved field. Return `0` if read this field and software is not allowed to change its value.

Table 82. Definition of fetch watchpoint n configuration 4 register

Bits	Name	Read/Write	Description
`9:0`	`ASID`	RW	The ASID being compared
`15:10`	`0`	R	Read-only constant `0`, writing to this field is ignored.
`23:16`	`0`	R	If the virtualization extension is not implemented, the field is read-only constant `0` and writing to this field is ignored.
`31:24`	`0`	R	Read-only constant `0`, writing to this field is ignored.

7.10. Control and Status Registers Related to Debugging

7.10.1. Debug Register (DBG)

Table 83. Definition of debug data save register

Bits	Name	Read/Write	Description
`0`	`DST`	R	`1` to indicate that it is currently in debug mode. The hardware sets this bit to `1` when a debug exception is triggered in non-debug mode. When this bit is `1`, the `ERTN` instruction is executed to clear this bit to `0`.
`7:1`	`DRev`	R	The version number of the debugging mechanism. `1` is the initial version.
`8`	`DEI`	R	`1` indicates that the debug exception type caught in debug mode is **DE**bug **I**nterrupt (DEI).
`9`	`DCL`	R	`1` indicates that the type of debug exception caught in debug mode is a **D**ebug **C**a**L**l exception (DCL).
`10`	`DFW`	R	`1` indicates that the type of debug exception caught in debug mode is the **D**ebug **F**etch **W**atchpoint exception (DFW).
`11`	`DMW`	R	`1` indicates that the debug exception type caught in debug mode is the **D**ebug load/store (**M**emory) **W**atchpoint exception (DMW).
`15:12`	`0`	R0	Read only as `0`.
`21:16`	`Ecode`	R	When a non-debug exception occurs in debug mode, the exception type code is recorded here. The meaning of the codes here is basically the same as the definitions in Table of exception encoding, with only three differences: • The TLB refill exception reuses the `0x7` exception code; • The debug call exception uses the `0xC` exception code; • The machine error exception uses the `0xE` exception code.
`31:22`	`0`	R0	Read only as `0`.

7.10.2. Debug Exception Return Address (DERA)

Table 84. Definition of debug exception program counter register

Bits	Name	Read/Write	Description
63:0	PC	RW	When a debug exception is triggered in non-debug mode, the hardware records the PC that triggered the exception here. When CSR.DBG.DM=1, the return address is fetched from here when the ERTN instruction is executed.

7.10.3. Debug Data Save Register (DSAVE)

This register is used to store data temporarily for the system software. Each dava save register can hold the data of one general-purpose register.

An additional SAVE register for debug exception handler is provided because debug exceptions can occur in any scenario and the handling of debug exceptions should be transparent to the software on the Host being debugged.

Table 85. Definition of debug register

Bits	Name	Read/Write	Description
63:0	Data	RW	Data for software to read and write only. The hardware does not modify the contents of this field except for the execution of CSR instructions.

1. The instructions affected by this control bit include LD[X].{H[U]/W[U]/D}, ST[X].{H/W/D}, LDPTR.{W/D}, STPTR.{W/D}, FLD[X].{S/D}, FST[X].{S/D}, LDPTE, LDDIR, IOCSRRD.{H/W/D} and IOCSRWR.{H/WD}.

2. Translator's note: This may be the fifth attribute not listed in Attributes of Reading and Writing.

3. Translator's note: This may be the fifth attribute not listed in Attributes of Reading and Writing.

Appendix A: Pseudocode Descriptions of the Function Definitions

A.1. Interpretation of Operators in Pseudocode

This section lists the meaning of the statement keywords and various operators involved in pseudocode, as well as the operator precedence relationships. In addition, the common conventions for different binary representations of numeric values in pseudocode are as follows:

- No prefix or `'d` or `##'d` prefix for decimal numbers, where the `##'d` prefix means that the decimal number is `##` bits wide;
- The prefix `'b` or `##'b` is used for binary numbers, where the prefix `##'b` indicates that the length of the binary number is `##` bits;
- The prefix `'h` or `##'h` indicates the hexadecimal number, where the prefix `##'h` indicates that the hexadecimal number is `##` bits wide, and the hexadecimal value of `A-F` uses uppercase letters.

Table 86. Interpretation of semantic keyword

Operators	Meaning

```
Return_Type
Function_Name(Variable, ...):
    Function_Body
return Return_Value
```
	Function Definition

```
if Condition1:
    Statement1
elif Condition2:
    Statement2
else:
    Statement3
```
	Conditional Statements

```
case Variable of:
    value1: Statement1
    value2: Statement2
    default: Default_Statement
```
	case conditional statement

```
Condition ? TRUE_Statement:
FALSE_Statement
```
	Conditional Judgment Statements

```
for Variable in Sequence:
    Statements
```
	for loop statement

Operators	Meaning
`range(N)`	A sequence of integers from `0` to `N-1` in steps of `1`
`range(Start, End, Step)`	Sequence of specified step values from the start value (inclusive) to the end value (exclusive)
`break`	Terminate the current loop
`signed(...)`	Signed integers
`unsigned(...)`	Unsigned integers
`fp16(...)`	Half-precision floating-point numbers
`fp32(...)`	Single-precision floating-point numbers
`fp64(...)`	Double-precision floating-point numbers
`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)`	`TRUE` if the variable is a signaling NaN number, `FALSE` otherwise
`isQNaN(Variable)`	`TRUE` if the variable is quiet NaN number, `FALSE` otherwise

Operators	Meaning
`SignalException(Exception)`	Trigger exception
`#`	Single line comment
`=`	Assignment

Table 87. Interpretation of bit string operators

Operators	Meaning
`[M:N]`	Bit `N` to bit `M` of the bit string
`{N{M}}`	Copy bit string `M` `N` times and splice them
`{N, M, …}`	Splice bit strings `N`, `M`, … in order

Table 88. Interpretation of arithmetic operators

Operators	Meaning
`+`	Add
`-`	Subtract
`*`	Multiply
`/`	Divide
`%`	Modulo
`**`	Power

Table 89. Interpretation of comparison operators

Operators	Meaning
`==`	equal to
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

Table 90. Interpretation of bit operators

Operators	Meaning
`&`	Bitwise AND
`|`	Bitwise OR
`^`	Bitwise XOR
`~`	Bitwise INVERSE

Operators	Meaning
`<<`	Logical Left Shift
`>>`	Logical Right Shift
`>>>`	Arithmetic Right Shift

Table 91. Interpretation of logical operators

Operators	Meaning
`and`	Logical AND
`or`	Logical OR
`not`	Logical NOT

Table 92. Operator priority

Operators	Meaning
`**`	Power
`-`	Inverse by place
`*, /, %`	Multiply, Divide, 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

A.2. Pseudocode Descriptions of Functional Functions

The pseudocode involved in the instruction descriptions in this manual is defined as follows.

A.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
```

A.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
```

A.2.3. Arithmetic Right Shift

```
bits(N) SRA(bits(N) x, integer sa):
    if sa == 0:
        result = x
    else:
        result = {{sa{x[N-1]}}, x[N-1:sa]}
    return result
```

A.2.4. Circular Right Shift

```
bits(N) ROTR(bits(N) x, integer sa):
    if sa == 0:
        result = x
    else:
        result = {x[sa-1:0], x[N-1:sa]}
    return result
```

A.2.5. Count the Number of Consecutive 1's Starting from High Order Bits

```
{bits(N)} CLO(bits(N) x):
    cnt = 0
    for i in range(N):
        if x[N-1-i] == 1'b0:
            return cnt
        else:
            cnt = cnt + 1
```

A.2.6. Count the Number of Consecutive `0`'s Starting from High Order Bits

```
{bits(N)} CLZ(bits(N) x):
    cnt = 0
    for i in range(N):
        if x[N-1-i] == 1'b1:
            return cnt
        else:
            cnt = cnt + 1
```

A.2.7. Count the Number of Consecutive 1's Starting from Low Order Bits

```
{bits(N)} CTO(bits(N) x):
    cnt = 0
    for i in range(N):
        if x[i] == 1'b0:
            return cnt
        else:
            cnt = cnt + 1
```

A.2.8. Count the Number of Consecutive 0's Starting from Low Order Bits

```
{bits(N)} CTZ(bits(N) x):
    cnt = 0
    for i in range(N):
        if x[i] == 1'b1:
            return cnt
        else:
            cnt = cnt + 1
```

A.2.9. Reverse the Order of the Bit String

```
{bits(N)} BITREV(bits(N) x):
    for i in range(N):
        res[i] = x[N-1-i]
    return res
```

A.2.10. CRC-32 Checksum Calculation

```
bits(32) CRC32(old_chksum, msg, width, poly):
    new_chksum = (old_chksum & 0xFFFFFFFF) ^ {{(64-width){1'b0}}, msg}
    for i in range(width):
        if (new_chksum & 1'b1):
            new_chksum = (new_chksum >> 1) ^ poly
        else:
            new_chksum = (new_chksum >> 1)
    return new_chksum
```

A.2.11. Single Precision Floating-point to Signed Word Integer

```
{bits(32)} FP32convertToSint32(bits(32) x, bits(1) I_en, bits(2) rm):
    case {I_en, rm} of:
        {1'b1, 2'd0}: return Sint32_convertToIntegerExactTiesToEven(x)
        {1'b1, 2'd1}: return Sint32_convertToIntegerExactTowardZero(x)
        {1'b1, 2'd2}: return
Sint32_convertToIntegerExactTowardPositive(x)
        {1'b1, 2'd3}: return
Sint32_convertToIntegerExactTowardNegative(x)
        {1'b0, 2'd0}: return Sint32_convertToIntegerTiesToEven(x)
        {1'b0, 2'd1}: return Sint32_convertToIntegerTowardZero(x)
        {1'b0, 2'd2}: return Sint32_convertToIntegerTowardPositive(x)
        {1'b0, 2'd3}: return Sint32_convertToIntegerTowardNegative(x)
```

A.2.12. Single Precision Floating-point to Signed Double Word Integer

```
{bits(64)} FP32convertToSint64(bits(32) x, bits(1) I_en, bits(2) rm):
    case {I_en, rm} of:
        {1'b1, 2'd0}: return Sint32_convertToIntegerExactTiesToEven(x)
        {1'b1, 2'd1}: return Sint32_convertToIntegerExactTowardZero(x)
        {1'b1, 2'd2}: return
Sint32_convertToIntegerExactTowardPositive(x)
        {1'b1, 2'd3}: return
Sint32_convertToIntegerExactTowardNegative(x)
        {1'b0, 2'd0}: return Sint32_convertToIntegerTiesToEven(x)
        {1'b0, 2'd1}: return Sint32_convertToIntegerTowardZero(x)
        {1'b0, 2'd2}: return Sint32_convertToIntegerTowardPositive(x)
        {1'b0, 2'd3}: return Sint32_convertToIntegerTowardNegative(x)
```

A.2.13. Double Precision Floating-point to Signed Word Integer

```
{bits(64)} FP64convertToSint32(bits(64) x, bits(1) I_en, bits(2) rm):
    case {I_en, rm} of:
        {1'b1, 2'd0}: return Sint64_convertToIntegerExactTiesToEven(x)
        {1'b1, 2'd1}: return Sint64_convertToIntegerExactTowardZero(x)
        {1'b1, 2'd2}: return
```

```
Sint64_convertToIntegerExactTowardPositive(x)
        {1'b1, 2'd3}: return
Sint64_convertToIntegerExactTowardNegative(x)
        {1'b0, 2'd0}: return Sint64_convertToIntegerTiesToEven(x)
        {1'b0, 2'd1}: return Sint64_convertToIntegerTowardZero(x)
        {1'b0, 2'd2}: return Sint64_convertToIntegerTowardPositive(x)
        {1'b0, 2'd3}: return Sint64_convertToIntegerTowardNegative(x)
```

A.2.14. Double Precision Floating-point to Signed Double Word Integer

```
{bits(64)} FP64convertToSint64(bits(64) x, bits(1) I_en, bits(2) rm):
    case {I_en, rm} of:
        {1'b1, 2'd0}: return Sint64_convertToIntegerExactTiesToEven(x)
        {1'b1, 2'd1}: return Sint64_convertToIntegerExactTowardZero(x)
        {1'b1, 2'd2}: return
Sint64_convertToIntegerExactTowardPositive(x)
        {1'b1, 2'd3}: return
Sint64_convertToIntegerExactTowardNegative(x)
        {1'b0, 2'd0}: return Sint64_convertToIntegerTiesToEven(x)
        {1'b0, 2'd1}: return Sint64_convertToIntegerTowardZero(x)
        {1'b0, 2'd2}: return Sint64_convertToIntegerTowardPositive(x)
        {1'b0, 2'd3}: return Sint64_convertToIntegerTowardNegative(x)
```

A.2.15. Round Single Precision Floating-point

```
{bits(32)} FP32_roundToInteger(bits(N) x, bits(1) I_en, bits(2) rm):
    if (I_en):
        return FP32_roundToIntegralExact(x)
    elif (rm == 0):
        return FP32_roundToIntegerTiesToEven(x)
    elif (rm == 1):
        return FP32_roundToIntegerTowardZero(x)
    elif (rm == 2):
        return FP32_roundToIntegerTowardPositive(x)
    elif (rm == 3):
        return FP32_roundToIntegerTowardNegative(x)
```

A.2.16. Round Double Precision Floating-point

```
{bits(64)} FP64_roundToInteger(bits(N) x, bits(1) I_en, bits(2) rm):
    if (I_en):
        return FP64_roundToIntegralExact(x)
    elif (rm=0):
        return FP64_roundToIntegerTi esToEven(x)
    elif (rm=1):
        return FP64_roundToIntegerTowardZero(x)
    elif (rm=2):
        return FP64_roundToIntegerTowardPositive(x)
    elif (rm=3):
        return FP64_roundToIntegerTowardNegative(x)
```

Appendix B: Table of Instruction Encoding

Table 93. Table of instruction encoding

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
CLO.W	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	rj					rd				
CLZ.W	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	rj					rd				
CTO.W	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	rj					rd				
CTZ.W	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	rj					rd				
CLO.D	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	rj					rd				
CLZ.D	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	rj					rd				
CTO.D	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	rj					rd				
CTZ.D	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	rj					rd				
REVB.2H	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	rj					rd				
REVB.4H	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	rj					rd				
REVB.2W	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	rj					rd				
REVB.D	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	rj					rd				
REVH.2W	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	rj					rd				
REVH.D	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	rj					rd				
BITREV.4B	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
BITREV.8B	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	rj					rd				
BITREV.W	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	rj					rd				
BITREV.D	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	rj					rd				
EXT.W.H	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	rj					rd				
EXT.W.B	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	rj					rd				
RDTIMEL.W	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	rj					rd				
RDTIMEH.W	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	rj					rd				
RDTIME.D	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	rj					rd				
CPUCFG	rd, rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	rj					rd				
ASRTLE.D	rj, rk	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	rk					rj					0	0	0	0	0
ASRTGT.D	rj, rk	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	rk					rj					0	0	0	0	0
ALSL.W	rd, rj, rk, sa2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	sa2		rk					rj					rd				
ALSL.WU	rd, rj, rk, sa2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	sa2		rk					rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
BYTEPICK.W	rd, rj, rk, sa2	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	sa2		rk					rj					rd				
BYTEPICK.D	rd, rj, rk, sa3	0	0	0	0	0	0	0	0	0	0	0	0	1	1	sa3			rk					rj					rd				
ADD.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	rk					rj					rd				
ADD.D	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	rk					rj					rd				
SUB.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	rk					rj					rd				
SUB.D	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	rk					rj					rd				
SLT	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	rk					rj					rd				
SLTU	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	rk					rj					rd				
MASKEQZ	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	rk					rj					rd				
MASKNEZ	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	rk					rj					rd				
NOR	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	rk					rj					rd				
AND	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	rk					rj					rd				
OR	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	rk					rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XOR	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1	rk					rj					rd				
ORN	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	rk					rj					rd				
ANDN	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	rk					rj					rd				
SLL.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	rk					rj					rd				
SRL.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	rk					rj					rd				
SRA.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	rk					rj					rd				
SLL.D	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	rk					rj					rd				
SRL.D	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	rk					rj					rd				
SRA.D	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	rk					rj					rd				
ROTR.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	rk					rj					rd				
ROTR.D	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1	rk					rj					rd				
MUL.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	rk					rj					rd				
MULH.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	rk					rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
MULH.WU	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	rk					rj					rd				
MUL.D	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	rk					rj					rd				
MULH.D	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	rk					rj					rd				
MULH.DU	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	rk					rj					rd				
MULW.D.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	rk					rj					rd				
MULW.D.WU	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	rk					rj					rd				
DIV.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	rk					rj					rd				
MOD.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	rk					rj					rd				
DIV.WU	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	rk					rj					rd				
MOD.WU	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	rk					rj					rd				
DIV.D	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	rk					rj					rd				
MOD.D	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	rk					rj					rd				
DIV.DU	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	rk					rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
MOD.DU	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	rk					rj					rd				
CRC.W.B.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	rk					rj					rd				
CRC.W.H.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	rk					rj					rd				
CRC.W.W.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	rk					rj					rd				
CRC.W.D.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	rk					rj					rd				
CRCC.W.B.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	rk					rj					rd				
CRCC.W.H.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	rk					rj					rd				
CRCC.W.W.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	rk					rj					rd				
CRCC.W.D.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	rk					rj					rd				
BREAK	code	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	code														
DBCL	code	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	code														
SYSCALL	code	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1	0	code														
ALSL.D	rd, rj, rk, sa2	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	sa2		rk					rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
SLLI.W	rd, rj, ui5	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	ui5					rj					rd				
SLLI.D	rd, rj, ui6	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	ui6						rj					rd				
SRLI.W	rd, rj, ui5	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	ui5					rj					rd				
SRLI.D	rd, rj, ui6	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	ui6						rj					rd				
SRAI.W	rd, rj, ui5	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	ui5					rj					rd				
SRAI.D	rd, rj, ui6	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	ui6						rj					rd				
ROTRI.W	rd, rj, ui5	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	1	ui5					rj					rd				
ROTRI.D	rd, rj, ui6	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	ui6						rj					rd				
BSTRINS.W	rd, rj, msbw, lsbw	0	0	0	0	0	0	0	0	0	1	1	msbw					0	lsbw					rj					rd				
BSTRPICK.W	rd, rj, msbw, lsbw	0	0	0	0	0	0	0	0	0	1	1	msbw					1	lsbw					rj					rd				
BSTRINS.D	rd, rj, msbd, lsbd	0	0	0	0	0	0	0	0	1	0	msbd						lsbd						rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
BSTRPICK.D	rd, rj, msbd, lsbd	0	0	0	0	0	0	0	0	1	1	msbd						lsbd						rj					rd				
FADD.S	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	fk					fj					fd				
FADD.D	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	fk					fj					fd				
FSUB.S	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	fk					fj					fd				
FSUB.D	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	fk					fj					fd				
FMUL.S	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	fk					fj					fd				
FMUL.D	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	fk					fj					fd				
FDIV.S	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	1	fk					fj					fd				
FDIV.D	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	0	fk					fj					fd				
FMAX.S	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	fk					fj					fd				
FMAX.D	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	fk					fj					fd				
FMIN.S	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	1	fk					fj					fd				
FMIN.D	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	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	09	08	07	06	05	04	03	02	01	00
FMAXA.S	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	1	fk					fj					fd				
FMAXA.D	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	1	0	fk					fj					fd				
FMINA.S	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	0	1	fk					fj					fd				
FMINA.D	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	0	fk					fj					fd				
FSCALEB.S	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	fk					fj					fd				
FSCALEB.D	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	fk					fj					fd				
FCOPYSIGN.S	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	1	fk					fj					fd				
FCOPYSIGN.D	fd, fj, fk	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	0	fk					fj					fd				
FABS.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	fj					fd				
FABS.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	fj					fd				
FNEG.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	fj					fd				
FNEG.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	1	0	fj					fd				
FLOGB.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	1	0	0	1	fj					fd				
FLOGB.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	1	0	1	0	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
FCLASS.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	1	1	0	1			fj					fd		
FCLASS.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	1	1	1	0			fj					fd		
FSQRT.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	1			fj					fd		
FSQRT.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	1	0			fj					fd		
FRECIP.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1			fj					fd		
FRECIP.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	1	1	0			fj					fd		
FRSQRT.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	1	0	0	1			fj					fd		
FRSQRT.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	1	0	1	0			fj					fd		
FRECIPE.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	1	1	0	1			fj					fd		
FRECIPE.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	1	1	1	0			fj					fd		
FRSQRTE.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	0	0	0	1			fj					fd		
FRSQRTE.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	0	0	1	0			fj					fd		
FMOV.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	0	1	0	1			fj					fd		
FMOV.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	0	1	1	0			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
MOVGR2FR.W	fd, rj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	1	0	0	1	rj					fd				
MOVGR2FR.D	fd, rj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	1	0	1	0	rj					fd				
MOVGR2FRH.W	fd, rj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	1	0	1	1	rj					fd				
MOVFR2GR.S	rd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	1	1	0	1	fj					rd				
MOVFR2GR.D	rd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	1	1	1	0	fj					rd				
MOVFRH2GR.S	rd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	1	1	1	1	fj					rd				
MOVGR2FCSR	fcsr, rj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	0	0	0	0	rj					fcsr				
MOVFCSR2GR	rd, fcsr	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	0	0	1	0	fcsr					rd				
MOVFR2CF	cd, fj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	0	1	0	0	fj					0	0	cd		
MOVCF2FR	fd, cj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	0	1	0	1	0	0	cj			fd				
MOVGR2CF	cd, rj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	0	1	1	0	rj					0	0	cd		

		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
MOVCF2GR	rd, cj	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	0	1	1	1	0	0	cj			rd				
FCVT.S.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	1	0	0	0	1	1	0	fj					fd				
FCVT.D.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	1	0	0	1	0	0	1	fj					fd				
FTINTRM.W.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	0	0	0	0	1	fj					fd				
FTINTRM.W.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	0	0	0	1	0	fj					fd				
FTINTRM.L.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	0	1	fj					fd				
FTINTRM.L.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	fj					fd				
FTINTRP.W.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	1	0	0	0	1	fj					fd				
FTINTRP.W.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	1	0	0	1	0	fj					fd				
FTINTRP.L.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	1	1	0	0	1	fj					fd				
FTINTRP.L.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	1	1	0	1	0	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
FTINTRZ.W.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	1	0	0	0	0	1	fj					fd				
FTINTRZ.W.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	1	0	0	0	1	0	fj					fd				
FTINTRZ.L.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	1	0	1	0	0	1	fj					fd				
FTINTRZ.L.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	1	0	1	0	1	0	fj					fd				
FTINTRNE.W.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	1	1	0	0	0	1	fj					fd				
FTINTRNE.W.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	1	1	0	0	1	0	fj					fd				
FTINTRNE.L.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	1	1	1	0	0	1	fj					fd				
FTINTRNE.L.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	1	1	1	0	1	0	fj					fd				
FTINT.W.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	1	0	0	0	0	0	1	fj					fd				
FTINT.W.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	1	0	0	0	0	1	0	fj					fd				
FTINT.L.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	1	0	0	1	0	0	1	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.L.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	1	0	0	1	0	1	0	fj					fd				
FFINT.S.W	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	1	0	0	0	1	0	0	fj					fd				
FFINT.S.L	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	1	0	0	0	1	1	0	fj					fd				
FFINT.D.W	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	1	0	0	1	0	0	0	fj					fd				
FFINT.D.L	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	1	0	0	1	0	1	0	fj					fd				
FRINT.S	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0	1	0	0	0	1	fj					fd				
FRINT.D	fd, fj	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0	1	0	0	1	0	fj					fd				
SLTI	rd, rj, si12	0	0	0	0	0	0	1	0	0	0	si12												rj					rd				
SLTUI	rd, rj, si12	0	0	0	0	0	0	1	0	0	1	si12												rj					rd				
ADDI.W	rd, rj, si12	0	0	0	0	0	0	1	0	1	0	si12												rj					rd				
ADDI.D	rd, rj, si12	0	0	0	0	0	0	1	0	1	1	si12												rj					rd				
LU52I.D	rd, rj, si12	0	0	0	0	0	0	1	1	0	0	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	09	08	07	06	05	04	03	02	01	00
ANDI	rd, rj, ui12	0	0	0	0	0	0	1	1	0	1	ui12												rj					rd				
ORI	rd, rj, ui12	0	0	0	0	0	0	1	1	1	0	ui12												rj					rd				
XORI	rd, rj, ui12	0	0	0	0	0	0	1	1	1	1	ui12												rj					rd				
CSRRD	rd, csr	0	0	0	0	0	1	0	0	csr														0	0	0	0	0	rd				
CSRWR	rd, csr	0	0	0	0	0	1	0	0	csr														0	0	0	0	1	rd				
CSRXCHG	rd, rj, csr	0	0	0	0	0	1	0	0	csr														rj!=0,1					rd				
CACOP	code, rj, si12	0	0	0	0	0	1	1	0	0	0	si12												rj					code				
LDDIR	rd, rj, level	0	0	0	0	0	1	1	0	0	1	0	0	0	0	level								rj					rd				
LDPTE	rj, seq	0	0	0	0	0	1	1	0	0	1	0	0	0	1	seq								rj					0	0	0	0	0
IOCSRRD.B	rd, rj	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	rj					rd				
IOCSRRD.H	rd, rj	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	rj					rd				
IOCSRRD.W	rd, rj	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	rj					rd				
IOCSRRD.D	rd, rj	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
IOCSRWR.B	rd, rj	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	rj					rd				
IOCSRWR.H	rd, rj	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	1	rj					rd				
IOCSRWR.W	rd, rj	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	1	1	0	rj					rd				
IOCSRWR.D	rd, rj	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	1	1	1	rj					rd				
TLBCLR		0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
TLBFLUSH		0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
TLBSRCH		0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
TLBRD		0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0
TLBWR		0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
TLBFILL		0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0
ERTN		0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
IDLE	level	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	1	level														
INVTLB	op, rj, rk	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	1	rk					rj					op				
FMADD.S	fd, fj, fk, fa	0	0	0	0	1	0	0	0	0	0	0	1	fa					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	09	08	07	06	05	04	03	02	01	00
FMADD.D	fd, fj, fk, fa	0	0	0	0	1	0	0	0	0	0	1	0	fa					fk					fj					fd				
FMSUB.S	fd, fj, fk, fa	0	0	0	0	1	0	0	0	0	1	0	1	fa					fk					fj					fd				
FMSUB.D	fd, fj, fk, fa	0	0	0	0	1	0	0	0	0	1	1	0	fa					fk					fj					fd				
FNMADD.S	fd, fj, fk, fa	0	0	0	0	1	0	0	0	1	0	0	1	fa					fk					fj					fd				
FNMADD.D	fd, fj, fk, fa	0	0	0	0	1	0	0	0	1	0	1	0	fa					fk					fj					fd				
FNMSUB.S	fd, fj, fk, fa	0	0	0	0	1	0	0	0	1	1	0	1	fa					fk					fj					fd				
FNMSUB.D	fd, fj, fk, fa	0	0	0	0	1	0	0	0	1	1	1	0	fa					fk					fj					fd				
FCMP.cond.S	cd, fj, fk	0	0	0	0	1	1	0	0	0	0	0	1	cond					fk					fj					0	0	cd		
FCMP.cond.D	cd, fj, fk	0	0	0	0	1	1	0	0	0	0	1	0	cond					fk					fj					0	0	cd		
FSEL	fd, fj, fk, ca	0	0	0	0	1	1	0	1	0	0	0	0	0	0	ca			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	09	08	07	06	05	04	03	02	01	00
ADDU16I.D	rd, rj, si16	0	0	0	1	0	0	si16																rj					rd				
LU12I.W	rd, si20	0	0	0	1	0	1	0	si20																				rd				
LU32I.D	rd, si20	0	0	0	1	0	1	1	si20																				rd				
PCADDI	rd, si20	0	0	0	1	1	0	0	si20																				rd				
PCALAU12I	rd, si20	0	0	0	1	1	0	1	si20																				rd				
PCADDU12I	rd, si20	0	0	0	1	1	1	0	si20																				rd				
PCADDU18I	rd, si20	0	0	0	1	1	1	1	si20																				rd				
LL.W	rd, rj, si14	0	0	1	0	0	0	0	0	si14														rj					rd				
SC.W	rd, rj, si14	0	0	1	0	0	0	0	1	si14														rj					rd				
LL.D	rd, rj, si14	0	0	1	0	0	0	1	0	si14														rj					rd				
SC.D	rd, rj, si14	0	0	1	0	0	0	1	1	si14														rj					rd				
LDPTR.W	rd, rj, si14	0	0	1	0	0	1	0	0	si14														rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
STPTR.W	rd, rj, si14	0	0	1	0	0	1	0	1	si14														rj					rd				
LDPTR.D	rd, rj, si14	0	0	1	0	0	1	1	0	si14														rj					rd				
STPTR.D	rd, rj, si14	0	0	1	0	0	1	1	1	si14														rj					rd				
LD.B	rd, rj, si12	0	0	1	0	1	0	0	0	0	0	si12												rj					rd				
LD.H	rd, rj, si12	0	0	1	0	1	0	0	0	0	1	si12												rj					rd				
LD.W	rd, rj, si12	0	0	1	0	1	0	0	0	1	0	si12												rj					rd				
LD.D	rd, rj, si12	0	0	1	0	1	0	0	0	1	1	si12												rj					rd				
ST.B	rd, rj, si12	0	0	1	0	1	0	0	1	0	0	si12												rj					rd				
ST.H	rd, rj, si12	0	0	1	0	1	0	0	1	0	1	si12												rj					rd				
ST.W	rd, rj, si12	0	0	1	0	1	0	0	1	1	0	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	09	08	07	06	05	04	03	02	01	00
ST.D	rd, rj, si12	0	0	1	0	1	0	0	1	1	1	si12												rj					rd				
LD.BU	rd, rj, si12	0	0	1	0	1	0	1	0	0	0	si12												rj					rd				
LD.HU	rd, rj, si12	0	0	1	0	1	0	1	0	0	1	si12												rj					rd				
LD.WU	rd, rj, si12	0	0	1	0	1	0	1	0	1	0	si12												rj					rd				
PRELD	hint, rj, si12	0	0	1	0	1	0	1	0	1	1	si12												rj					hint				
FLD.S	fd, rj, si12	0	0	1	0	1	0	1	1	0	0	si12												rj					fd				
FST.S	fd, rj, si12	0	0	1	0	1	0	1	1	0	1	si12												rj					fd				
FLD.D	fd, rj, si12	0	0	1	0	1	0	1	1	1	0	si12												rj					fd				
FST.D	fd, rj, si12	0	0	1	0	1	0	1	1	1	1	si12												rj					fd				
LDX.B	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	rk					rj					rd				
LDX.H	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	rk					rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
LDX.W	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	rk					rj					rd				
LDX.D	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	0	0	1	1	0	0	0	rk					rj					rd				
STX.B	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	rk					rj					rd				
STX.H	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	0	1	0	1	0	0	0	rk					rj					rd				
STX.W	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	0	1	1	0	0	0	0	rk					rj					rd				
STX.D	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	0	1	1	1	0	0	0	rk					rj					rd				
LDX.BU	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	rk					rj					rd				
LDX.HU	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	1	0	0	1	0	0	0	rk					rj					rd				
LDX.WU	rd, rj, rk	0	0	1	1	1	0	0	0	0	0	1	0	1	0	0	0	0	rk					rj					rd				
PRELDX	hint, rj, rk	0	0	1	1	1	0	0	0	0	0	1	0	1	1	0	0	0	rk					rj					hint				
FLDX.S	fd, rj, rk	0	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0	rk					rj					fd				
FLDX.D	fd, rj, rk	0	0	1	1	1	0	0	0	0	0	1	1	0	1	0	0	0	rk					rj					fd				
FSTX.S	fd, rj, rk	0	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	0	rk					rj					fd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
FSTX.D	fd, rj, rk	0	0	1	1	1	0	0	0	0	0	1	1	1	1	0	0	0			rk					rj					fd		
SC.Q	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	0	1	1	1	0			rk					rj					fd		
LLACQ.W	rd, rj	0	0	1	1	1	0	0	0	0	1	0	1	0	1	1	1	1	0	0	0	0	0			rj					fd		
SCREL.W	rd, rj	0	0	1	1	1	0	0	0	0	1	0	1	0	1	1	1	1	0	0	0	0	1			rj					fd		
LLACQ.D	rd, rj	0	0	1	1	1	0	0	0	0	1	0	1	0	1	1	1	1	0	0	0	1	0			rj					fd		
SCREL.D	rd, rj	0	0	1	1	1	0	0	0	0	1	0	1	0	1	1	1	1	0	0	0	1	1			rj					fd		
AMCAS.B	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	0	0	0	0			rk					rj					fd		
AMCAS.H	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	0	0	0	1			rk					rj					fd		
AMCAS.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	0	0	1	0			rk					rj					fd		
AMCAS.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	0	0	1	1			rk					rj					fd		
AMCAS_DB.B	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	0	1	0	0			rk					rj					fd		
AMCAS_DB.H	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	0	1	0	1			rk					rj					fd		
AMCAS_DB.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	0	1	1	0			rk					rj					fd		

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
AMCAS_DB.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	0	1	1	1	rk					rj					fd				
AMSWAP.B	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	1	0	0	0	rk					rj					fd				
AMSWAP.H	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	1	0	0	1	rk					rj					fd				
AMADD.B	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	1	0	1	0	rk					rj					fd				
AMADD.H	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	1	0	1	1	rk					rj					fd				
AMSWAP_DB.B	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	1	1	0	0	rk					rj					fd				
AMSWAP_DB.H	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	1	1	0	1	rk					rj					fd				
AMADD_DB.B	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1	0	rk					rj					fd				
AMADD_DB.H	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1	1	rk					rj					fd				
AMSWAP.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	rk					rj					rd				
AMSWAP.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	1	rk					rj					rd				
AMADD.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	0	0	1	0	rk					rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
AMADD.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	0	0	1	1	rk					rj					rd				
AMAND.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	0	1	0	0	rk					rj					rd				
AMAND.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	0	1	0	1	rk					rj					rd				
AMOR.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	0	1	1	0	rk					rj					rd				
AMOR.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	0	1	1	1	rk					rj					rd				
AMXOR.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	1	0	0	0	rk					rj					rd				
AMXOR.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	1	0	0	1	rk					rj					rd				
AMMAX.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	1	0	1	0	rk					rj					rd				
AMMAX.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	1	0	1	1	rk					rj					rd				
AMMIN.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	1	1	0	0	rk					rj					rd				
AMMIN.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	1	1	0	1	rk					rj					rd				
AMMAX.WU	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	1	1	1	0	rk					rj					rd				
AMMAX.DU	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	rk					rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
AMMIN.WU	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	rk					rj					rd				
AMMIN.DU	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	0	0	0	1	rk					rj					rd				
AMSWAP_DB.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	0	0	1	0	rk					rj					rd				
AMSWAP_DB.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	0	0	1	1	rk					rj					rd				
AMADD_DB.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	0	1	0	0	rk					rj					rd				
AMADD_DB.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	0	1	0	1	rk					rj					rd				
AMAND_DB.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	0	1	1	0	rk					rj					rd				
AMAND_DB.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	0	1	1	1	rk					rj					rd				
AMOR_DB.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	1	0	0	0	rk					rj					rd				
AMOR_DB.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	1	0	0	1	rk					rj					rd				
AMXOR_DB.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	1	0	1	0	rk					rj					rd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
AMXOR_DB.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	1	0	1	1			rk					rj					rd		
AMMAX_DB.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	1	1	0	0			rk					rj					rd		
AMMAX_DB.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	1	1	0	1			rk					rj					rd		
AMMIN_DB.W	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	1	1	1	0			rk					rj					rd		
AMMIN_DB.D	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	0	1	1	1	1	1			rk					rj					rd		
AMMAX_DB.WU	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0			rk					rj					rd		
AMMAX_DB.DU	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1			rk					rj					rd		
AMMIN_DB.WU	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	0			rk					rj					rd		
AMMIN_DB.DU	rd, rk, rj	0	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	1			rk					rj					rd		
DBAR	hint	0	0	1	1	1	0	0	0	0	1	1	1	0	0	1	0	0								hint							
IBAR	hint	0	0	1	1	1	0	0	0	0	1	1	1	0	0	1	0	1								hint							

		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
FLDGT.S	fd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	0	1	0	0	0	rk					rj					fd				
FLDGT.D	fd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	0	1	0	0	1	rk					rj					fd				
FLDLE.S	fd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	0	1	0	1	0	rk					rj					fd				
FLDLE.D	fd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	0	1	0	1	1	rk					rj					fd				
FSTGT.S	fd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	0	1	1	0	0	rk					rj					fd				
FSTGT.D	fd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	0	1	1	0	1	rk					rj					fd				
FSTLE.S	fd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	0	1	1	1	0	rk					rj					fd				
FSTLE.D	fd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	0	1	1	1	1	rk					rj					fd				
LDGT.B	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	rk					rj					rd				
LDGT.H	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	1	rk					rj					rd				
LDGT.W	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	0	0	1	0	rk					rj					rd				
LDGT.D	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	0	0	1	1	rk					rj					rd				
LDLE.B	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	0	1	0	0	rk					rj					rd				

		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
LDLE.H	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	0	1	0	1	rk					rj					rd				
LDLE.W	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	0	1	1	0	rk					rj					rd				
LDLE.D	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	0	1	1	1	rk					rj					rd				
STGT.B	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	1	0	0	0	rk					rj					rd				
STGT.H	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	1	0	0	1	rk					rj					rd				
STGT.W	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	1	0	1	0	rk					rj					rd				
STGT.D	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	1	0	1	1	rk					rj					rd				
STLE.B	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	1	1	0	0	rk					rj					rd				
STLE.H	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	1	1	0	1	rk					rj					rd				
STLE.W	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	1	1	1	0	rk					rj					rd				
STLE.D	rd, rj, rk	0	0	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	rk					rj					rd				
BEQZ	rj, offs	0	1	0	0	0	0	offs[15:0]																rj					offs[20:16]				
BNEZ	rj, offs	0	1	0	0	0	1	offs[15:0]																rj					offs[20:16]				

		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
BCEQZ	cj, offs	0	1	0	0	1	0	offs[15:0]																0	0	cj			offs[20:16]				
BCNEZ	cj, offs	0	1	0	0	1	0	offs[15:0]																0	1	cj			offs[20:16]				
JIRL	rd, rj, offs	0	1	0	0	1	1	offs[15:0]																rj					rd				
B	offs	0	1	0	1	0	0	offs[15:0]																offs[25:16]									
BL	offs	0	1	0	1	0	1	offs[15:0]																offs[25:16]									
BEQ	rj, rd, offs	0	1	0	1	1	0	offs[15:0]																rj					rd				
BNE	rj, rd, offs	0	1	0	1	1	1	offs[15:0]																rj					rd				
BLT	rj, rd, offs	0	1	1	0	0	0	offs[15:0]																rj					rd				
BGE	rj, rd, offs	0	1	1	0	0	1	offs[15:0]																rj					rd				
BLTU	rj, rd, offs	0	1	1	0	1	0	offs[15:0]																rj					rd				
BGEU	rj, rd, offs	0	1	1	0	1	1	offs[15:0]																rj					rd				