Xuantie-C920R2S1-User-Manual

Jun 07, 2024

杭州中天微系统有限公司 Hangzhou C-SKY MicroSystems Co., LTD

Address: Room 201, 2/F, Building 5, No.699 Wangshang Road , Hangzhou, Zhejiang, China

Website: www.xrvm.cn

杭州中天微系统有限公司 Hangzhou C-SKY MicroSystems Co., LTD

地址: 中国浙江省杭州市网商路 699 号 5 号楼 2 楼 201 室

网址: www.xrvm.cn

Version History

Version	Description	Date
01	Initial release.	2021.07.31
02	Added simplified power-off scenarios. Updated vector descriptions.	2021.09.17
03	Updated address encoding description of pmpaddr.	2021.10.25
04	Various updates to text and images.	2022.08.21
05	Added references for SYSMAP configuration.	2023.03.02
06	Added content related to multi-cluster subsystems.	2023.03.07
07	Added content related to RV debug.	2023.06.06
08	Added content related to vector module v1.0 and updated manual version to R2S0.	2023.08.01
09	Added content related to C920V2 and updated manual version to R2S0.	2023.09.01
10	Updated content related to Vector, added description for DVM, and updated manual version to R2S1.	2023.11.16
11	Modified the description of Secure Debug.	2024.03.08
12	Update PMU description.	2024.04.25
13	Added mseccfg register.	2024.05.09
14	Updated the outstanding capability of the master device interface	2024.05.11
15	Updated C920 programming model, instruction set version description	2024.05.30

Xuantie-C920R2S1-User-Manual

CHAPTER 1

Overview

1.1 Introduction

C920MP is a high-performance 64-bit multi-core CPU built on the RISC-V architecture. It is oriented to edge computing that requires high performance, such as edge servers, edge computing cards, advanced machine vision, advanced video surveillance, autonomous driving, mobile smart terminals, and 5G base stations. C920MP adopts a homogeneous multi-core architecture, supporting 1 to 4 configurable cores. Each C920 core runs on a microsystem architecture and has been optimized for high performance. Moreover high-performance technologies are introduced, such as a 3-way issue, 8-way execution superscalar architecture, and multi-channel data prefetching. In addition, C920 core performs real-time detection and shuts down internal idle function modules to reduce dynamic power consumption of CPU.

1.2 Features

1.2.1 Key Architectural Features of C920MP

- Homogeneous multi-core architecture and support for configuration of 1 to 4 cores;
- Support for independent power-off of each core and cluster power-off;
- Support for one AMBA 4.0 AXI/ACE Master interface and 128-bit bus width;
- Support for one configurable AXI 14.0 Low Latency Port (LLP) and 128-bit bus width;
- Support for one configurable AXI 14.0 Device Coherence Port (DCP) and 128-bit bus width;
- Two levels of caches provided: L1 cache running on the Harvard architecture and L2 shared cache;

- L1 cache size is configurable, and instruction and data cache support 32KB and 64KB separately, with a cache line size of 64 bytes;
- L1 cache supports for The Modified, Exclusive, Shared, Invalid (MESI) coherence protocol, and L2 cache supports for MESI coherence protocol;
- L2 cache supports for 16-way connection with configurable Error Correcting Code (ECC) mechanism;
- L2 cache size is configurable, supporting 256KB/512KB/1MB/2MB/4MB/8MB with a cache line size of 64 bytes;
- Support for Core Local Interrupt Controller (CLINT) and Platform-level Interrupt Controller (PLIC);
- Support for Timers;
- Support for RISC-V debugging framework and multi-core/multi-cluster debugging;

1.2.2 Key Features of C920 Core

- RISC-V 64GC[V] instruction architecture;
- Support for Little-endian mode;
- 9-stage to 12-stage deep pipelined architecture;
- Support for 3-way issue, 8-way execution superscalar architecture, fully transparent to software;
- In-order fetch, out-of-order issue, out-of-order completion, and in-order retirement;
- Two-level Translation Lookaside Buffer (TLB) memory management units for virtual/physical address translation and memory management;
- Instruction Cache (I Cache) and Data Cache (D Cache) sizes are configurable, supporting 32KB and 64KB, with a cache line size of 64B;
- I Cache can be configured with Parity Check, and D Cache can be configured with ECC or Parity Check;
- Support for instruction prefetch and auto-detection and dynamic startup of hardware;
- Low-power access technology for I Cache branch prediction;
- Low-power execution technology with short-loop cache;
- 64 KB two-level multi-way parallel branch predictor;
- Configurable branch target buffer with 1024/2048 entries;
- Support for 12-layer hardware return address stack;
- Indirect jump branch predictor with 256 entries;
- Non-blocking issue and speculative execution;
- Renaming technology based on physical registers;
- Support for 0-latency move instructions;
- Dual issue and full out-of-order execution for load/store instructions;
- Support for concurrent bus access for up to 8 read requests and 8 write requests;
- Support for write combining;
- Support for 8-way D Cache hardware prefetching and stride prefetching;
- Support for fixed configuration of floating-point execution unit, half-precision, single-precision and double-precision;

1.2.3 Key Features of Vector Computing Unit

- Compliance with RISC-V V extension;
- Support for computing capability up to 512GOPS (@int8)/256GFlops (@FP16) at the configuration of 4 cores and 2GHz;
- The Vector Execution Unit supports FP16/BF16/EP32/FP64 floating points and INT8/INT16/INT32/INT64 integer vector operations;
- Support for 128-bit vector register length VLEN;
- Support for dual vector execution units for computation and data store pipeline.
- Support for 128-bit vector data store access width VLEN;
- Support for segment load and store instruction;
- Support for performance-optimized unaligned memory access.

1.3 Configurable Options

Configurable options of C920MP are illustrated in the following Table 1.1 .

Table 1.1: **Configurable 0ptions of C920MP**

Configurable Unit	Configurable Options	Detailed Information
Number of C920 Core	1/2/3/4	C920MP provides configurable options for 1 to 4 C920 cores.
VECTOR_SIMD	Yes/No	Vector execution units are configurable.
Master Interface Protocol	AXI/ACE	Master Interface supports AXI or ACE protocol.
LLP	Yes/No	Optional for LLP
DCP	Yes/No	For peripherals to access on-chip cache, ensuring data consistency, and DCP can be connected with Direct Memory Access (DMA)
L1 I-Cache	32K/64K	Configured with the size of 32KB、64KB.
L1 D-Cache	32K/64K	Configured with the size of 32KB、64KB.
L1 ECC/Parity	Yes/No	Parity for L1 I-Cache ECC for L1 D-Cache
L2 Cache	Size: 256K/512K/1M/2M/4M/8M	Configured with the size of 356KB~8MB.
L2 ECC	Yes/No	ECC for L2 Tag/Data RAM.
MMU	SV39/SV48	Support for SV39 or SV48 mode, and SV48 configuration can also support SV39.
PMP Region	8/16/32/64	Available number of PMP region
ePMP	Yes/No	Enhanced PMP is configurable
Debug Resources	Min/Typical/Max	Debug Resource is configurable.
System Bus Access	Yes/No	Whether Debug supports System Bus Access or not
TEE	Yes/No	Optional TEE extension

Continued on next page

Table 1.1 – continued from previous page

Configurable Unit	Configurable Options	Detailed Information
ShareBus	Yes/No	Whether memory of C902MP supports ShareBus or not
System Bus Access	Yes/No	Support configuring a separate AXI bus interface for a debugger to bypass CPU and independently access to memory space.
pic_top (External Interrupt controller)		
Number of Interrupts	64-1024，step 32	Number of Interrupts
Number of Clusters	1-16	The number of clusters sharing the same pic_top
Number of Harts	1-256	The number of Harts sharing the same tic_top **Note:** There is no need to clarify the corresponding between Hart and Cluster.
TEE	Yes/No	Optional TEE extension
tdt_dmi_top (Debugging Conversion Bridge, JTAG to APB)		
Number of APB Ports	1-32	The same tdt_dmi_top can debug several clusters and one APB port corresponds to one cluster.
Sys APB Access	Yes/No	Sys APB Access permits CPU to access debug registers through Master Port and system bus

1.4 XuanTie Extended Architecture

C920 is compatible with XuanTie C-series extended architecture 1.0, which provides extensions in the following aspects:

- Operation instructions: C920 improves operation capabilities with integer, floating-point, and load/store instructions, well supplementing the RISC-V base instruction sets.
- Cache operations: C920 provides user-friendly cache maintenance operations to improve cache efficiency.
- Memory model: C920 manages address attributes efficiently to improve memory access efficiency.
- Control registers: C920 extends the features of control registers based on the standard RISC-V architecture.
- Multi-core synchronization instructions: C920 adopts multi-core synchronization instructions to improve efficiency of multicore consistency maintenance.

1.5 Version Compatibility

C920 is compatible with the RISC-V standard, for detailed information, please refer to Table 1.2 and Table 1.3 .

Table 1.2: **C920 Program Model and the Corresponding Versions**

Specification	C920V2
RISC-V Profile	RVA23
RISC-V Instruction Set Maunal Volume I: User-Level ISA	User-Level ISA (20191213 Ratified)
RISC-V Instruction Set Manual Volume II: Privileged Architecture	Version 20211203
RISC-V "V" Vector Extension	Version 1.0
RISC-V Bit-Manipulation ISA-extensions	Version 1.0.0-38-g865e7a7, 2021-06-28: Release candidate
RISC-V Cryptography Extensions Volume I Scalar & Entropy Source Instructions	Version v1.0.0, 2' nd December, 2021: Ratified 少部分（Zbkc, Zkt）支持
RISC-V External Debug Support	Version 0.13.2
RISC-V platform-level interrupt controller (PLIC)	Version 1.0
PMP Enhancements for memory access and execution prevention on Machine mode (Smepmp)	Version 1.0, 12/2021
RISC-V Code Size Reduction (Zc)	v1.0
RISC-V WAIT-on-Reservation-Set(Zawrs)	Version 1.0 11/2022
玄铁扩展指令集	支持

Table 1.3: **C920 Instruction Set and the Corresponding Versions**

Specification	Modules	C920V2
RISC-V Instruction Set Maunal Volume I: User-Level ISA	Width of an integer register in bits (XLEN)	RV64
	Control and Status Register (CSR) Instructions (Zicsr)	Version 2.0
	Instruction-Fetch Fence (Zifencei)	Version 2.0
	Standard Extensions for Half-Precision Floating-Point (Zfh, Zfhmin)	Version 1.0 (Zfh)
	Pause Hint (Zihintpause)	Version 2.0
	Standard Extension for Base Counters and Timers (Zicntr)	Version 2.0
	Standard Extension for Hardware Performance Counters (Zihpm)	支持
	RISC-V bfloat16 Specification	Version 1.0.0-rc1, 27 October 2023: Frozen
	Misc. basic Scalar FP (Zfa)	支持
	Non-Temporal Locality Hints (Zihintntl)	支持
	RISC-V Wait-on-Reservation-Set (Zawrs)	支持
	ZiCond	Version 1.0, 2023-02-22
RISC-V Instruction Set Manual Volume II: Privileged Architecture	Virtual Memory System	SV39 + SV48
	NAPOT Translation Contiguity (Svnapot)	Version 1.0
	Page-Based Memory Types (Svpbmt)	Version 1.0
	Fine-Grained Address-Translation Cache Invalidation (Svinval)	Version 1.0

Continued on next page

Table 1.3 – continued from previous page

Specification	Modules	C920V2
	"stimecmp / vstimecmp" Extension (Sstc)	Version 0.5.4-3f9ed34, 2021-10-13: frozen
	Count Overflow and Mode-Based Filtering Extension (Sscofpmf)	支持
	Base Cache Management Operation ISA Extensions (Zicbom, Zicboz, Zicbop)	Version 1.0.1-b34ea8a, 2022-05-13: Ratified
RISC-V "V" Vector Extension	Vector Extension for Half-Precision Floating-Point(Zvfh, Zfhmin)	Zvfh
RISC-V Bit-Manipulation ISA-extensions	Bit-manipulation (Zba, Zbb, Zbc, Zbs)	Zba, Zbb, Zbc, Zbs
RISC-V Cryptography Extensions Volume I Scalar & Entropy Source Instructions	RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions	Zbkc, Zkt
RISC-V Code Size Reduction (Zc)	Zca, Zcf, Zcb, Zcmb, Zcmp, Zcmpe, Zcmt	Zca, Zcd, Zcb

1.6 Naming Conventions

The standard symbols and operators in this document is shown in Fig. 1.1 .

Symbol	Function
+	Addition
−	Subtraction
*	Multiplication
/	Division
>	Greater than
<	Less than
=	Equal to
⩾	Greater than or equal to
⩽	Less than or equal to
≠	Not equal
&	And
|	Or
⊕	Exclusive-OR
NOT	Negation
:	Connection
⟹	Transmission
⟺	Exchange
±	Error
0b0011	Binary number
0x0F	Hexadecimal number
rd	Integer Destination Register
rs1	Integer Source Register 1
rs2	Integer Source Register 2
rs3	Integer Source Register 3
fd	Floating Point Destination Register
fs1	Floating Point Source Register 1
fs2	Floating Point Source Register 2
fs3	Floating Point Source Register 3
vd	Vector Destination Register
vs1	Vector Source Register 1
vs2	Vector Source Register 2
vs3	Vector Source Register 3

Fig. 1.1: **Symbol List**

1.6.1 Terms

- **Logic 1:** The level value corresponding to the Boolean logic value TRUE.
- **Logic 0:** The level value corresponding to the Boolean logic value FALSE.
- **Set:** The action of setting one or more bits to the level value corresponding to logic 1.
- **Clear:** The action of setting one or more bits to the level value corresponding to logic 0.
- **Reserved bit:** A bit reserved for feature extension. The value of a reserved bit is 0 unless otherwise specified.
- **Signal:** An electrical value used to transfer information based on its state or state transition.
- **Pin:** An external electrical and physical connection. One pin can connect to multiple signals.
- **Enable:** The action of switching a discrete signal to a valid state:
  - Switch a valid low-level signal from a high level to a low level.
  - Switch a valid high-level signal from a low level to a high level.
- **Disable:** The action of switching the state of an enabled signal:
  - Switch a valid low-level signal from a low level to a high level.
  - Switch a valid high-level signal from a high level to a low level.
- **LSB:** The least significant bit. **MSB**: The most significant bit.
- **Signal**, **bit field**, and **control bit**: represented by a general rule.
- **Identifier followed by a value range**: Indicates a group of signals from the most significant bit to the least significant bit.

  For example, "addr[4:0]" indicates a group of address buses, where addr[4] indicates the most significant bit, and addr[0] indicates the least significant bit.
- **Single identifier**: Indicates a single signal.

  For example, "pad_cpu_rst_b" indicates a single signal.

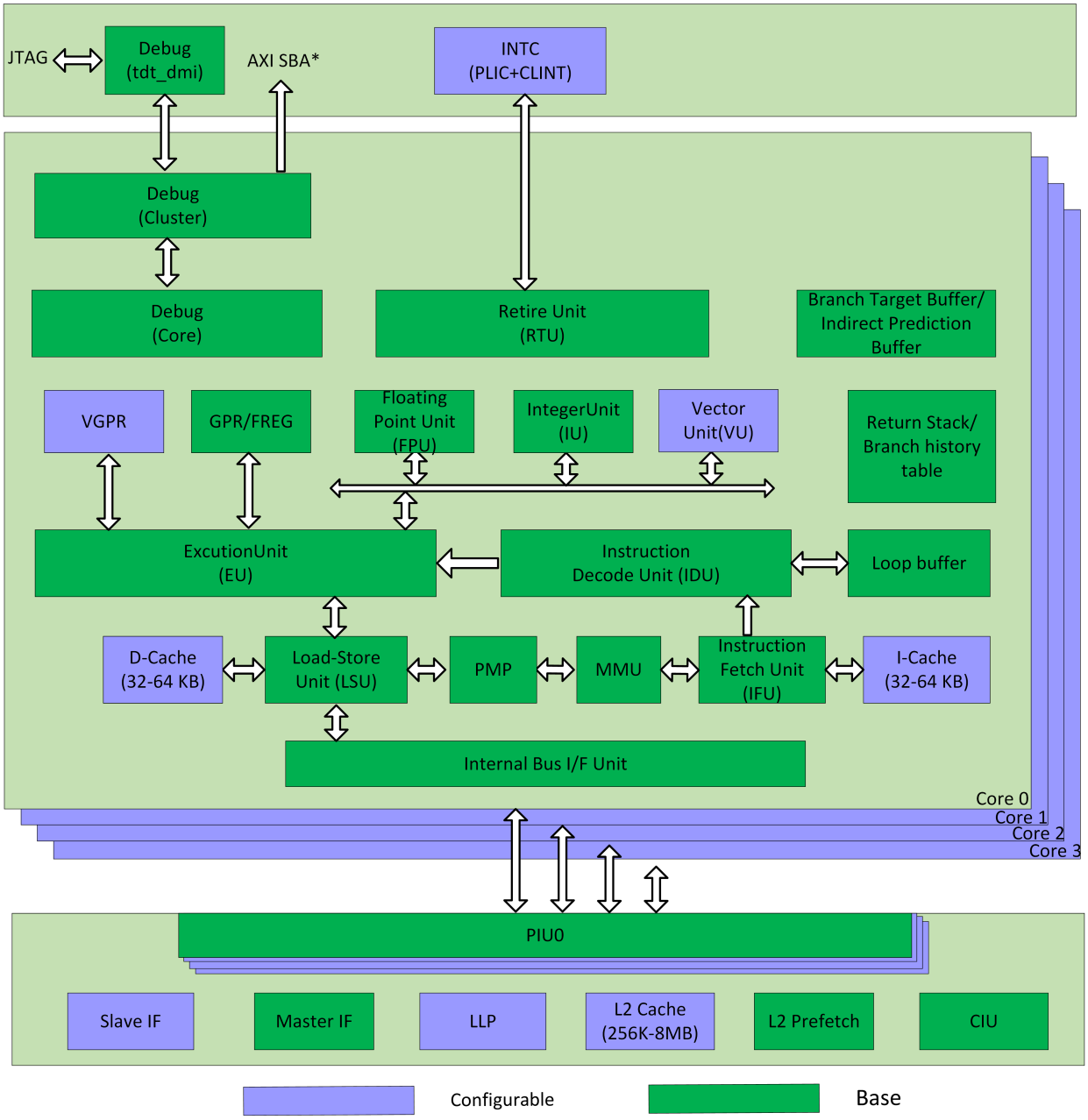
  In some cases, an identifier followed by a number is used to express a specific meaning. For example, "addr15" indicates the 16th bit of a group of buses.

CHAPTER 2

C920MP Overview

2.1 Structure Diagram

The structure diagram of C920MP is shown in Fig. 2.1 .



* ： This feature is configurable.

Fig. 2.1: **C920MP Microarchitecture**

2.2 In-core Subsystems

C920 mainly consists of the following in-core subsystems: Instruction Fetch Unit (IFU), Instruction Decoding Unit (IDU), Integer Unit (IU), Floating-point Unit (FPU), Load/Store Unit (LSU), Retirement Unit (RTU), Virtual Memory Management unit (MMU), and Physical Memory Protection (PMP) unit.

2.2.1 IFU

Instruction Fetch Unit (IFU) enables to fetch up to eight instructions at a time and process them in parallel. It improves access efficiency with a variety of technologies, such as I cache branch prediction, instruction registers, loop acceleration buffers, and direct/indirect branch prediction. IFU features low power consumption, high branch prediction accuracy, and high prefetch efficiency.

2.2.2 IDU

Instruction Decode Unit (IDU) enables to decode three instructions and detect data correlation at a time. IDU detects data correlation between instructions by physical register renaming technology, and perform out-of-order instruction dispatch to the next-level pipeline for execution. IDU supports out-of-order scheduling and distribution of instructions. It mitigates performance loss due to data correlation through speculative issuing.

2.2.3 Execution Unit

Execution units include IU, FPU and Vector Execution Unit (VU).

IU consists of Arithmetic Logic Unit (ALU), Multiplication Unit (MULT), Division Unit (DIV), and Jump Unit (BJU). ALU performs 64-bit integer operation. MULT supports 16*16, 32*32, and 64*64 integer multiplication. DIV adopts the base 16 SRT algorithm, and the cycle time varies with the operation numbers. BJU can complete branch prediction error handling within a single cycle.

FPUs consist of Floating-point Arithmetic Logic Unit (FALU), Floating-point Fused Multiply-add Unit (FMAU), and Floating-point Divide and Square Unit (FDSU). FPU supports half-precision, single-precision and double-precision operations. FALU is applied to operations such as addition, subtraction, comparison, conversion, register data transmission, sign-injection, and classification. FMAU performs common multiplication, fused multiply-add and other operations. FDSU performs floating-point division and square root, and other operations.

Vector Execution Unit is the extension of FPU. And FPU is extended to vector floating-point unit based on the scalar floating-point calculation. Vector Floating Point Units consist of Vector Floating Point Arithmetic Logic Unit (VFALU), Vector Floating Point Multiply-add Unit (VFMAU), and Vector Floating Point Divide-Square Unit (VFDSU), and support vector floating point operations in different bit widths.

In addition, Vector Integer Unit has been added. Vector integer units include Vector Addition Unit (VALU), Vector Shift Unit (VSHIFT), Vector Multiplication Unit (VMUL), Vector Division Unit (VDIVU), Vector Permutation Unit (VPERM), Vector Reduction unit (VREDU), and vector Logic Operation Unit (VMISC).

2.2.4 LSU

Load Store Unit (LSU) supports dual issue for scalar store/load instructions, single issue for vector store/load instructions, and full out-of-order execution for all the store/load instructions. LSU also supports non-blocking access to caches, and byte, halfword, word, doubleword, and quadword store/load instructions, and sign bit/zero extension for byte and halfword load instructions. Store/load instructions can be executed in a pipeline so that only one data entry is accessed per cycle. LSU supports 8-way hardware prefetch, transferring data to L1 D-Cache in advance. If D-Cache is absent, LSU supports parallel bus access.

2.2.5 RTU

RTU consists of a re-order buffer and a physical register stack. And the re-order buffer controls out-of-order recycling and in-order retirement of instructions. The physical register stack controls out-of-order recycling and transfer of results. RTU improves the instruction retirement efficiency through parallel recycling and fast retirement of instructions. Moreover, RTU supports parallel retirement of up to three instructions per clock cycle and implements precise exceptions.

2.2.6 MMU

MMU complies with RISC-V SV39/SV48 standard for converting 39/48-bit virtual addresses to 40-bit physical addresses. C920 MMU extends software refill methods and address attributes, based on the hardware refill criteria defined in SV39/SV48.

For detailed information, please refer to *Memory Model* .

2.2.7 PMP

PMP complies with RISC-V standard, supports 8/16/32/64 entries, but does not support the NA4 mode. The minimum granularity supported by the PMP unit is 4 KB.

For detailed information, please refer to *Memory Model* .

2.3 Multi-core Subsystems

C920 multicore subsystem contains Data Coherence Interface Unit (CIU), L2 cache, Master Device Interface Unit, configurable AXI 4.0 Device Coherence Port (DCP) and Low Latency Port (LLP).

2.3.1 CIU

CIU ensures data coherence between L1 D-Caches based on MESI protocol. Two listening buffers are configured to handle multiple listening requests in parallel and fully utilize the listening bandwidth. CIU adopts an efficient data bypassing mechanism. When a listening request hits L1 D-Cache under listening, the data is directly bypassed to the

request initiation core. In addition, CIU supports broadcasting of invalid Translation Lookaside Buffer (TLB)/I-Cache requests, which reduces the software maintaining costs of data coherence between TLB/I-Cache and D-Cache.

2.3.2 L2 cache

L2 cache is tightly coupled to the CIU for synchronous access with L1 D-Caches. L2 cache adopts a block-based pipelining architecture and can handle two access requests in parallel within one cycle. It supports a maximum access bandwidth of 1024 bits. The operating frequency of L2 cache is the same as that of C920. TAG RAM and DATA RAM access latency can be configured by software.

2.3.3 Master Device Interface

The master device interface supports ACE/AXI4.0 protocol and address access by keyword priority, and can operate under different system clock to CPU clock ratios (1:1, 1:2, 1:3, 1:4, 1:5, 1:6, 1:7, 1:8).

2.3.4 DCP

DCP supports AXI4.0 protocol, which supports for peripheral access to on-chip D-Cache. The hardware achieves data consistency and is applied to connecting to external Direct Memory Access (DMA).

2.3.5 LLP

Low Latency Port (LLP) supports the AXI4.0 protocol and serves as a dedicated port for accessing system peripherals. Due to its separate data pathway, the LLP is not subject to bandwidth constraints imposed by the master port.

2.4 Multi-cluster Subsystem

C920 multi-cluster subsystem includes: Interrupt Controller (PIC), timers, and a customized multi-cluster, multicore single-port debugging framework.

2.4.1 PIC

PIC includes Platform Level Interrupt Controller (PLIC) and Core Localized Interrupt Controller (CLINT). PLIC supports sampling and distribution of up to 1023 external interrupt sources, level and pulse interrupts, and 32 levels of interrupt priority. CLINT supports for handling software interrupts and timer interrupts. The PIC in C920 adopts an external design to handle external and local interrupts across multiple clusters.

For detailed information, please refer to *Interrupt Controller* .

2.4.2 Timer

Multi-cluster multi-core system provides one shared 64-bit system timer. Each core has its own private timer comparison value register. Values of the system timer are collected and compared with those in the private timer comparison value register to generate timer signals.

For detailed information, please refer to *Interrupt Controller* .

2.4.3 Debugging System

C920 adopts a multi-cluster, multi-core, single-port debugging framework that accesses each cluster' s debug unit (DM) through a shared JTAG interface, to control cores in and out of debug mode and access processor resources. JTAG interface and DM support RISC-V debug V0.13.2 protocol.

For detailed information, please refer to *Debug* chapter and *Multicore Debug Controller Reference Manual*.

2.5 Interface Overview

In terms of features, C920 is mainly classified into clock reset signal, bus system, interrupt system, debug system, low power system, DFT system, and CPU running monitoring signal. The key interfaces of C920 are illustrated in Fig. 2.2 .

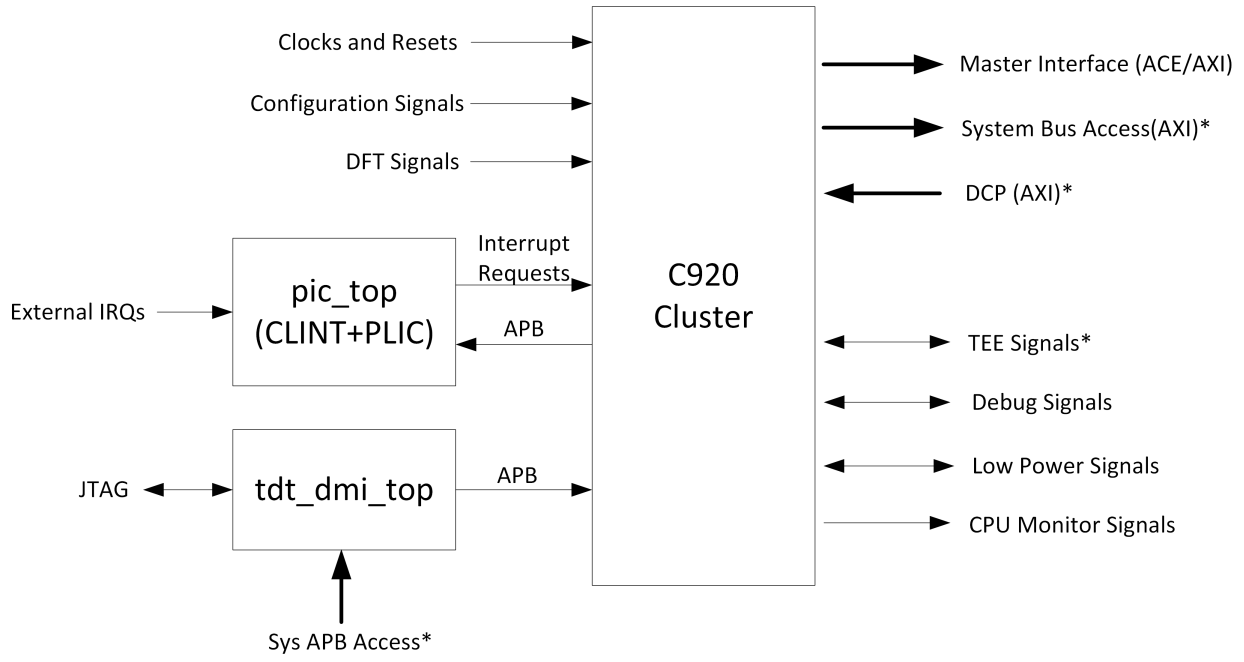


Fig. 2.2: **C920MP Interfaces Overview**

CHAPTER 3

Instruction Sets

This chapter mainly describes the instruction sets implemented in C920, which are divided into two main pats: RV base instruction sets and XuanTie extended instruction sets.

3.1 RV Base Instruction Sets

3.1.1 Integer Instruction Set (RV64I)

The integer instruction set can be categorized by features as follows:

- Add/Subtract instructions
- Logical operation instructions
- Shift instructions
- Compare instructions
- Data transfer instructions
- Branch jump instructions
- Memory access instructions
- CSR operation instructions
- Low power instructions
- Exception-return instructions
- Special function instructions

Table 3.1: **Integer Instructions(RV64I) List**

Instruction	Description	Execution Latency
Add/Subtract Instructions		
ADD	A signed add instruction	1
ADDW	A signed add instruction on the lower 32 bits	1
ADDI	A signed add immediate instruction	1
ADDIW	A signed add immediate instruction on the lower 32 bits	1
SUB	A signed subtract instruction	1
SUBW	A signed subtract instruction on the lower 32 bits	1
Logic Operation Instructions		
AND	A bitwise AND instruction.	1
ANDI	An immediate bitwise AND instruction	1
OR	A bitwise OR instruction	1
ORI	An immediate bitwise OR instruction	1
XOR	A bitwise XOR instruction.	1
XORI	An immediate bitwise XOR instruction	1
Shift Instructions		
SLL	A logical left shift instruction	1
SLLW	A word logical left shift instruction on the lower 32 bits	1
SLLI	An immediate logical left shift instruction	1
SLLIW	An immediate logical left shift instruction on the lower 32 bits	1
SRL	A logical right shift instruction	1
SRLW	A logical right shift instruction on the lower 32 bits	1
SRLI	An immediate logical right shift instruction	1
SRLIW	An immediate logical right shift instruction on the lower 32 bits	1
SRA	An arithmetic right shift instruction	1
SRAW	An arithmetic right shift instruction on the lower 32 bits	1
SRAI	An immediate arithmetic right shift instruction	1
SRAIW	An immediate arithmetic right shift instruction on the lower 32 bits	1
Compare Instructions		
SLT	A signed set-if-less-than instruction	1
SLTU	An unsigned set-if-less-than instruction	1
SLTI	A signed set-if -less-than-immediate instruction	1
SLTIU	An unsigned set-if -less-than-immediate instruction	1
Data Transfer Instructions		
LUI	A load upper immediate instruction	1
AUIPC	An add upper immediate to PC instruction	1
Branch Jump Instructions		
BEQ	A branch-if-equal instruction	1
BNE	A branch-if-not-equal instruction	1
BLT	A signed branch-if-less-than instruction	1
BGE	A signed branch-if-greater- than-or-equal instruction	1
BLTU	An unsigned branch-if-less-than instruction	1

Continued on next page

Table 3.1 – continued from previous page

Instruction	Description	Execution Latency
BGEU	An unsigned branch-if-greater- than-or-equal instruction	1
JAL	An instruction for directly jumping to a subroutine	1
JALR	An jump and link register instruction	1
Memory Access Instructions		
LB	A sign-extended byte byte-load instruction	WEAK ORDER LOAD: >=3 STORE: 1 STRONG ORDER Aperiodic
LBU	An unsign-extended byte-load instruction	Same as above
LH	A sign-extended halfword-load instruction	Same as above
LHU	An unsign-extended halfword-load instruction	Same as above
LW	A sign-extended word- load instruction	Same as above
LWU	An unsign-extended word-load instruction	Same as above
LD	A doubleword-load instruction	Same as above
SB	A byte-store instruction	Same as above
SH	A halfword-store instruction	Same as above
SW	A word-store instruction	Same as above
SD	A doubleword-store instruction	Same as above
Control Register Operation Instructions		
CSRRW	CSR read/write	Blocked Aperiodic
CSRRS	CSR read/set	Same as above
CSRRC	CSR read/clear	Same as above
CSRRWI	CSR read/write immediate	Same as above
CSRRSI	CSR read/set immediate	Same as above
CSRRCI	CSR read/clear immediate	Same as above
Low Power Instructions		
WFI	An instruction for entering the low-power mode	Aperiodic
Exception-return Instructions		
MRET	An exception return instruction in machine mode (M-mode)	Blocked Aperiodic
SRET	An exception return instruction in supervisor mode (S-mode)	Same as above
Special Function Instructions		
FENCE	A memory synchronization instruction	Aperiodic
FENCE.I	An instruction stream synchronization instruction	Blocked Aperiodic
SFENCE.VMA	A virtual memory synchronization instruction	Same as above
ECALL	An environment exception instruction	1
EBREAK	A breakpoint instruction	1

For specific instruction descriptions and definitions, please refer to *Appendix A-1 I Instructions* .

3.1.2 Multiplication and Division Instructions (RV64M) Set

Table 3.2: **Integer Multiplication and Division (RV64M) Instruction**

Instruction	Description	Execution latency
MUL	A signed multiply instruction	4
MULW	A signed multiply instruction on the lower 32 bits	4
MULH	A signed multiply instruction that extracts upper bits	4
MULHS	A signed-unsigned multiply instruction that extracts upper bits	4
MULHU	An unsigned multiply instruction that extracts upper bits	4
DIV	A signed divide instruction.	3-20
DIVW	A signed divide instruction on the lower 32 bits	3-12
DIVU	An unsigned divide instruction.	3-20
DIVUW	An unsigned divide instruction on the lower 32 bits	3-12
REM	A signed remainder instruction	3-20
REMW	A signed remainder instruction on the lower 32 bits	3-12
REMU	An unsigned remainder instruction.	3-20
REMUW	An unsigned remainder instruction on the lower 32 bits	3-12

For specific instruction descriptions and definitions, please refer to *Appendix A-2 M instructions* .

3.1.3 Atomic Instruction Set (RV64A)

Table 3.3: **Atomic Instruction (RV64A) List**

Instruction	Description	Execution Latency
LR.W	A word load-reserved instruction.	This instruction is split into multiple atomic instructions for execution. This instruction may be split into blocking execution, with unpredictable instruction delays
LR.D	A doubleword load-reserved instruction.	
SC.W	A word store-conditional instruction.	
SC.D	A doubleword store-conditional instruction.	
AMOSWAP.W	An atomic swap instruction on the lower 32 bits.	
AMOSWAP.D	An atomic swap instruction.	
AMOADD.W	An atomic add instruction that on the lower 32 bits.	
AMOADD.D	An atomic add instruction.	
AMOXOR.W	An atomic bitwise XOR instruction on the lower 32 bits.	
AMOXOR.D	An atomic bitwise XOR instruction.	
AMOAND.W	An atomic bitwise AND instruction on the lower 32 bits.	
AMOAND.D	An atomic bitwise AND instruction.	

Continued on next page

Table 3.3 – continued from previous page

Instruction	Description	Execution Latency
AMOOR.W	An atomic bitwise OR instruction that on the lower 32 bits.	
AMOOR.D	An atomic bitwise OR instruction	
AMOMIN.W	An atomic signed MIN instruction on the lower 32 bits.	
AMOMIN.D	An atomic signed MIN instruction	
AMOMAX.W	An atomic signed MAX instruction on the lower 32 bits.	
AMOMAX.D	An atomic signed MAX instruction.	
AMOMINU.W	An atomic unsigned MIN instruction on the lower 32 bits.	
AMOMINU.D	An atomic unsigned MIN instruction.	
AMOMAXU.W	An atomic unsigned MAX instruction on the lower 32 bits.	
AMOMAXU.D	An atomic unsigned MAX instruction.	

For specific instruction descriptions and definitions, please refer to *Appendix A-3 A Instructions* .

3.1.4 Single-precision Floating-point Instruction Set (RV64F)

Single-precision floating-point instruction set can be categorized by features as follows:

- Operation instructions
- Sign injection instructions
- Data transfer instructions
- Compare instructions
- Data type conversion instructions
- Memory store instructions
- Floating-point classification instructions

Table 3.4: **RV64F Instruction Set**

Instruction	Description	Execution Latency
Operation Instructions		
FADD.S	A single-precision floating-point add instruction.	3
FSUB.S	A single-precision floating-point subtract instruction.	3
FMUL.S	A single-precision floating-point multiply instruction	3
FMADD.S	A single-precision floating-point multiply-add instruction.	4
FMSUB.S	A single-precision floating-point multiply-subtract instruction.	4
FNMADD.S	A single-precision floating-point negate-(multiply-add) instruction.	4

Continued on next page

Table 3.4 – continued from previous page

Instruction	Description	Execution Latency
FNMSUB.S	A single-precision floating-point negate- (multiply-subtract) instruction.	4
FDIV.S	A single-precision floating-point divide instruction.	8-14
FSQRT.S	A single-precision floating-point square-root instruction.	8-14
Sign Injection Instructions		
FSGNJ.S	A single-precision floating-point sign-injection instruction.	3
FSGNJN.S	A single-precision floating-point sign negate injection instruction.	3
FSGNJX.S	A single-precision floating-point sign XOR injection instruction.	3
Data Transfer Instructions		
FMV.X.W	A single-precision floating-point read /move instruction.	1+1 in split execution
FMV.W.X	A single-precision floating-point write /move instruction.	3, in sequential execution
Compare Instructions		
FMIN.S	A single-precision floating-point MIN instruction.	3
FMAX.S	A single-precision floating-point MAX instruction.	3
FEQ.S	A single-precision floating-point compare equal instruction.	1+1 in split execution
FLT.S	A single-precision floating-point compare less than instruction.	1+1 in split execution
FLE.S	A single-precision floating-point compare less than or equal to instruction.	1+1 in split execution
Data Type Conversion Instructions		
FCVT.W.S	An instruction that converts a single-precision floating-point number into a signed integer.	3+1 in split execution
FCVT.WU.S	An instruction that converts a single-precision floating-point number into an unsigned integer.	3+1 in split execution
FCVT.S.W	An instruction that converts a signed integer into a single-precision floating-point number.	3, in sequential execution
FCVT.S.WU	An instruction that converts an unsigned integer into a single-precision floating-point number.	3, in sequential execution
FCVT.L.S	An instruction that converts a single-precision floating-point number into a signed long integer.	3+1 in split execution
FCVT.LU.S	An instruction that converts a single-precision floating-point number into an unsigned long integer.	3+1 in split execution
FCVT.S.L	An instruction that converts a signed long integer into a single-precision floating-point number.	3, in sequential execution
FCVT.S.LU	An instruction that converts an unsigned long integer into a single-precision floating-point number.	3, in sequential execution
Memory Store Instructions		
FLW	A single-precision floating-point load instruction.	WEAK ORDER LOAD: >=3 STORE: 1 STRONG ORDER Aperiodic

Continued on next page

Table 3.4 – continued from previous page

Instruction	Description	Execution Latency
FSW	A single-precision floating-point store instruction.	Same as above
Floating-point Classification Instructions		
FCLASS.S	A single-precision floating-point classification instruction.	1+1

For specific instruction descriptions and definitions, please refer to *Appendix A-4 F instructions* .

3.1.5 Double-Precision Floating-Point Instruction Set

Double-precision floating-point instruction set can be categorized by features as follows:

- Operation instructions
- Sign-injection instructions
- Data transfer instructions
- Compare instructions
- Data type conversion instruction
- Memory store instructions

Table 3.5: **Double-Precision Floating-Point Instruction Set List**

Instruction	Description	Execution latency
Operation Instructions		
FADD.D	A double-precision floating-point add instruction.	3
FSUB.D	A double-precision floating-point subtract instruction.	3
FMUL.D	A double-precision floating-point multiply instruction.	4
FMADD.D	A double-precision floating-point multiply-add instruction.	5
FMSUB.D	A double-precision floating-point multiply-subtract instruction.	5
FNMADD.D	A double-precision floating-point negate- (multiply-add) instruction.	5
FNMSUB.D	A double-precision floating-point negate- (multiply-subtract) instruction.	5
FDIV.D	A double-precision floating-point divide instruction.	8-22
FSQRT.D	A double-precision floating-point square-root instruction.	8-22
Sign Injection Instructions		
FSGNJ.D	A double-precision floating-point sign-injection instruction	3
FSGNJN.D	A double-precision floating-point negate sign-injection instruction	3
FSGNJX.D	A double-precision floating-point XOR sign-injection instruction	3
Data Transfer Instructions		
FMV.X.D	A double-precision floating-point read transfer instruction	1+1 in split execution
FMV.D.X	A double-precision floating-point write transfer instruction	1+1 in split execution
Compare Instructions		
FMIN.D	A double-precision floating-point MIN instruction	3
FMAX.D	A double-precision floating-point MAX instruction.	3

Continued on next page

Table 3.5 – continued from previous page

Instruction	Description	Execution latency
FEQ.D	A double-precision floating-point compare equal instruction.	1+1 in split execution
FLT.D	A double-precision floating-point compare less than instruction.	1+1 in split execution
FLE.D	A double-precision floating-point compare less than or equal to instruction.	1+1 in split execution
Data Type Conversion Instructions		
FCVT.S.D	An instruction that converts a double-precision floating-point number into a single-precision floating-point number.	3
FCVT.D.S	An instruction that converts a double-precision floating-point number into a half-precision floating-point number.	3
FCVT.W.D	An instruction that converts a double-precision floating-point number into a signed integer.	3+1 in split execution
FCVT.WU.D	An instruction that converts a double-precision floating-point number into an unsigned integer.	3+1 in split execution
FCVT.D.W	An instruction that converts a signed integer into a double-precision floating-point number	3, in sequential execution
FCVT.D.WU	The instruction that converts an unsigned integer into a double-precision floating-point number.	3, in sequential execution
FCVT.L.D	An instruction that converts a double-precision floating-point number into a signed long integer.	3+1 in split execution
FCVT.LU.D	An instruction that converts a double-precision floating-point number into an unsigned long integer.	3+1 in split execution
FCVT.D.L	An instruction that converts a signed long integer into a double-precision floating-point number.	3, in sequential execution
FCVT.D.LU	An instruction that converts an unsigned long integer into a double-precision floating-point number.	3, in sequential execution
Memory Store Instructions		
FLD	A double-precision floating-point load instruction	WEAK ORDER LOAD: >=3 STORE: 1 STRONG ORDER Aperiodic
FSD	A double-precision floating-point store instruction.	Same as above
Floating-point Classification Instructions		
FCLASS.D	A double-precision floating-point classification instruction	1+1

For specific instruction descriptions and definitions, please refer to *Appendix A-5 D Instructions* .

3.1.6 Compressed Instruction Set (RV64C)

Compressed Instruction Set can be categorized by features as follows:

- Add/Subtract instructions

- Logical operation instructions
- Shift instructions
- Data transfer instructions
- Branch jump instructions
- Immediate offset access instructions

Table 3.6: **Compressed Instruction (RV64C) List**

Instruction	Description	Execution Latency
Add/Subtract Instructions		
C.ADD	A signed add instruction	1
C.ADDW	A signed add instruction on the lower 32 bits	1
C.ADDI	A signed add immediate instruction	1
C.ADDIW	A signed add immediate instruction on the lower 32 bits	1
C.SUB	A compressed signed subtract instruction	1
C.SUBW	A signed subtract instruction on the lower 32 bits	1
C.ADDI16SP	An instruction that adds an immediate scaled by 16 to the stack pointer	1
C.ADDI4SPN	An instruction that adds an immediate scaled by 4 to the stack pointer	1
Logic Operation Instructions		
C.AND	A bitwise AND instruction	1
C.ANDI	An immediate bitwise AND instruction	1
C.OR	A bitwise OR instruction	1
C.XOR	A bitwise XOR instruction	1
Shift Instructions		
C.SLLI	An immediate logical left shift instruction	1
C.SRLI	An immediate logical right shift instruction	1
C.SRAI	An immediate arithmetic right shift instruction	1
Data Transfer Instructions		
C.MV	A data transfer instruction	1
C.LI	Load lower immediate	1
C.LUI	Load upper immediate	1
Branch Jump Instructions		
C.BEQZ	A branch-if-equal-to-zero instruction.	1
C.BNEZ	A branch- if-not-equal-to-zero instruction.	1
C.J	An unconditional jump instruction	1
C.JR	A jump to register instruction	1
C.JALR	A jump And link register instruction	1
Immediate Offset Access Instructions		
C.LW	A word load instruction	WEAK ORDER LOAD: >=3 STORE: 1 STRONG ORDER Aperiodic

Continued on next page

Table 3.6 – continued from previous page

Instruction	Description	Execution Latency
C.SW	A word store instruction	Same as above
C.LWSP	A load Word from stack pointer instruction	Same as above
C.SWSP	A word stack store instruction	Same as above
C.LD	A doubleword load instruction	Same as above
C.SD	A doubleword store instruction	Same as above
C.LDSP	A doubleword stack load instruction	Same as above
C.SDSP	A doubleword stack store instruction	Same as above
C.FLD	A double-precision load instruction	Same as above
C.FSD	A double-precision store instruction	Same as above
C.FLDSP	A double-precision stack store instruction	Same as above
C.FSDSP	A double-precision stack load instruction	Same as above
Special Instructions		
C.NOP	A no-operation instruction	1
C.EBREAK	A breakpoint instruction	1

For specific instruction descriptions and definitions, please refer to *Appendix A-6 C Instructions*

3.1.7 Vector Instruction Set (RV64V)

For specific information of vector instruction set, please refer to *RISC-V "V" Vector Extension, Version 1.0* .

URL: https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf

3.2 XuanTie Extended Instruction Set

C920 provides some extended custom instructions based on RV64GC instruction set. The half-precision floating-point instructions of C920 extended instruction set can be directly applied. Moreover, all C920 extended instruction sets need to enable the Extended Instruction Set Enable bit (THEADISAEE) in the Machine Mode Extended Status Register (MXSTATUS), to operated normally; Otherwise, illegal instruction exceptions will be generated.

3.2.1 Arithmetic Operation Instructions

Table 3.7: **Arithmetic Operation Instructions Set**

Instruction	Description	Execution Latency
Add/Subtract Instructions		
ADDSL	Register shift and add instruction	1
MULA	A multiply-add instruction	Non-additive number correlation: 4
MULS	A multiply-subtract instruction	Non-additive number correlation: 4
MULAW	A multiply-add instruction on the lower 32 bits	Additive number correlation: 1

Continued on next page

Table 3.7 – continued from previous page

Instruction	Description	Execution Latency
MULSW	A multiply-subtract instruction on the lower 32 bits.	Additive number correlation: 1
MULAH	A multiply-add instruction on the lower 16 bits	Additive number correlation: 1
MULSH	A multiply-subtract instruction on the lower 16 bits.	Additive number correlation: 1
Shift Instructions		
SRRI	A cyclic right shift instruction.	1
SRRIW	A cyclic right shift instruction on the lower 32 bits.	1
Move Instructions		
MVEQZ	A moving instruction when the register value is 0	1
MVNEZ	A moving instruction when the register value is not 0	1

For specific instruction descriptions and definitions, please refer to *Appendix B-3 Arithmetic Operation Instructions* .

3.2.2 Bit Operation Instructions

Table 3.8: **Bit Operation Instructions Set**

Instruction	Description	Execution Latency
Bit Operation Instructions		
TST	An instruction for testing bits with the value of 0.	1
TSTNBZ	An instruction for testing bytes with the value of 0.	1
REV	A byte reverse instruction	1
REVW	A byte reverse instruction on the lower 32 bits.	1
FF0	An instruction for fast finding the first bit with the value of 0.	1
FF1	An instruction for fast finding the first bit with the value of 1.	1
EXT	A signed extension instruction for extracting consecutive bits of a register.	1
EXTU	A zero extension instruction for extracting consecutive bits of a register.	1

For specific instruction descriptions and definitions, please refer to *Appendix B-4 Bitwise Operation Instruction* .

3.2.3 Memory Access Instructions

Table 3.9: **Memory Access Instructions Set**

Store Instruction	Description	Execution latency
FLRD	A doubleword load instruction for shifting floating-point registers.	WEAK ORDER >=3 STRONG ORDER Aperiodic
FLRW	A word load instruction for shifting floating-point registers.	-
FLURD	A doubleword load instruction for shifting the lower 32 bits in floating-point registers.	-

Continued on next page

Table 3.9 – continued from previous page

Store Instruction	Description	Execution latency
FLURW	A word load instruction for shifting the lower 32 bits in floating-point registers.	-
LRB	A byte load instruction for shifting registers and extending signed bits.	-
LRH	A halfword load instruction for shifting registers and extending signed bits	-
LRW	A halfword load instruction for shifting registers and extending signed bits	-
LRD	A doubleword load instruction for shifting registers.	-
LRBU	A byte load instruction for shifting registers and extending zero bits.	-
LRHU	A halfword load instruction for shifting registers and extending zero bits.	-
LRWU	A word load instruction for shifting registers and extending zero bits.	-
LURB	A byte load instruction for shifting registers and extending signed bits.	-
LURH	A halfword load instruction for shifting registers and extending signed bits.	-
LURW	A word load instruction for shifting the lower 32 bits in registers and extending signed bits.	-
LURD	A doubleword load instruction for shifting the lower 32 bits in registers.	-
LURBU	A byte load instruction for shifting the lower 32 bits in registers and extending zero bits.	-
LURHU	A halfword load instruction for shifting the lower 32 bits in registers and extending zero bits.	-
LURWU	A word load instruction for shifting the lower 32 bits in registers and extending zero bits.	-
LBIA	A base-address auto-increment instruction for loading bytes and extending signed bits.	This instruction is split into the load and ALU instructions for execution.
LBIB	A byte load instruction for auto-incrementing the base address and extending signed bits.	-
LHIA	A base-address auto-increment instruction for loading halfwords and extending signed bits.	-
LHIB	A halfword load instruction for auto-incrementing the base address and extending signed bits.	-
LWIA	A base-address auto-increment instruction for loading words and extending signed bits.	-

Continued on next page

Table 3.9 – continued from previous page

Store Instruction	Description	Execution latency
LWIB	The word load instruction for auto-incrementing the base address and extending signed bits.	-
LDIA	A base-address auto-increment instruction for loading doublewords and extending signed bits.	-
LDIB	A doubleword load instruction for auto-incrementing the base address and extending signed bits.	-
LBUIA	A base-address auto-increment instruction for loading bytes and extending zero bits.	-
LBUIB	A byte load instruction for auto-incrementing the base address and extending zero bits.	-
LHUIA	An address auto-increment instruction for loading halfwords and extending zero bits.	-
LHUIB	A halfword load instruction for auto-incrementing the base address and extending zero bits	-
LWUIA	An address auto-increment instruction for loading words and extending zero bits.	-
LWUIB	A word load instruction for auto-incrementing the base address and extending zero bits.	-
LDD	A double-register load instruction.	This instruction is split into two load instructions for execution.
LWD	A double-register word load instruction for extending signed bits.	-
LWUD	A double-register word load instruction for extending zero bits.	-
FSRD	A doubleword store instruction for shifting floating-point registers.	WEAK ORDER LOAD: >=3 STORE: 1 STRONG ORDER Aperiodic
FSRW	A word store instruction for shifting floating-point registers.	-
FSURD	A doubleword store instruction for shifting the lower 32 bits in floating-point registers.	-
FSURW	A word store instruction for shifting the lower 32 bits in floating-point registers.	-
SRB	A byte store instruction for shifting registers.	-
SRW	A word store instruction for shifting registers.	-
SRD	A doubleword store instruction for shifting registers.	-
SURB	A byte store instruction for shifting the lower 32 bits in registers.	-
SURH	A halfword store instruction for shifting the lower 32 bits in registers.	-
SURW	A word store instruction for shifting the lower 32 bits in registers.	-

Continued on next page

Table 3.9 – continued from previous page

Store Instruction	Description	Execution latency
SURD	A doubleword store instruction for shifting the lower 32 bits in floating-point registers	-
SBIA	A base-address auto-increment instruction for storing bytes	This instruction is split into the store and ALU instructions for execution.
SBIB	A byte store instruction for auto-incrementing the base address.	-
SHIA	A base-address auto-increment instruction for storing halfwords.	-
SHIB	A halfword store instruction for auto-incrementing the base address.	-
SWIA	A base-address auto-increment instruction for storing words.	-
SWIB	A word store instruction for auto-incrementing the base address.	-
SDIA	A base-address auto-increment instruction for storing doublewords	-
SDIB	A doubleword store instruction for auto-incrementing the base address.	-
SDD	A double-register store instruction.	This instruction is split into two store instructions for execution.
SWD	An instruction for storing the lower 32 bits in double registers	-

For specific instruction descriptions and definitions, please refer to *Appendix B-5 Store Instructions* .

3.2.4 Cache Instructions

Table 3.10: **Cache Instructions List**

Instruction	Description	Execution Latency(LMUL=1)
DCACHE.CALL	An instruction that clears all dirty page table entries in the D-Cache.	Blocked Aperiodic
DCACHE.CIALL	An instruction that clears all dirty page table entries in the D-Cache and invalidates the entries.	
DCACHE.CIPA	An instruction that clears dirty page table entries that match the specified physical addresses in the D-Cache and invalidates the entries. (This instruction also acts on the L2 cache.)	
DCACHE.CISW	An instruction that clears dirty page table entries in the D-Cache based on the specified way/set and invalidates the entries.	
DCACHE.CIVA	An instruction that clears dirty page table entries that match the specified virtual addresses in the D-Cache and invalidates the entries. (This instruction also acts on the L2 cache.)	

Continued on next page

Table 3.10 – continued from previous page

Instruction	Description	Execution Latency(LMUL=1)
DCACHE.CPA	An instruction that clears dirty page table entries that match the specified physical addresses in the D-Cache. (This instruction also acts on the L2 cache.)	
DCACHE.CPAL1	An instruction that clears dirty page table entries that match the specified physical addresses in the L1 D-Cache.	
DCACHE.CSW	An instruction that clears dirty page table entries in the D-Cache based on the specified way/set.	
DCACHE.CVA	An instruction that clears dirty page table entries that match the specified virtual addresses in the D-Cache. (This instruction also acts on the L2 cache.)	
DCACHE.CVAL1	An instruction that clears dirty page table entries that match the specified virtual addresses in the L1 D-Cache.	
DCACHE.IPA	An instruction that invalidates page table entries that match the specified physical addresses in the D-Cache. (This instruction also acts on the L2 cache.)	
DCACHE.ISW	An instruction that invalidates page table entries in the D-Cache based on the specified way/set.	
DCACHE.IVA	An instruction that invalidates page table entries that match the specified virtual addresses in the D-Cache. (This instruction also acts on the L2 cache.)	
DCACHE.IALL	An instruction that invalidates all page table entries in the D-Cache	
ICACHE.IALL	An instruction that invalidates all page table entries in the I-Cache	Aperiodic
ICACHE.IALLS	An instruction that invalidates all page table entries in the I-Cache through broadcasting	
ICACHE.IPA	An instruction that invalidates page table entries that match the specified physical addresses in the I-Cache.	
ICACHE.IVA	An instruction that invalidates page table entries that match the specified virtual addresses in the I-Cache.	

For specific instruction descriptions and definitions, please refer to *Appendix B-1 Cache Instructions* .

3.2.5 Multi-core Synchronization Instructions

Table 3.11: **Multi-core Synchronization Instructions**

Multi-core Synchronization Instructions	Description
SYNC	A synchronization instruction
SYNC.S	A synchronization broadcast instruction

Continued on next page

Table 3.11 – continued from previous page

Multi-core Synchronization Instructions	Description
SYNC.I	An instruction for synchronizing the clearing operation.
SYNC.IS	A broadcast instruction for synchronizing the clearing operation.

For specific instruction descriptions and definitions, please refer to *Appendix B-2 Multi-core Synchronization Instructions* .

3.2.6 Half-precision Floating-point Instructions

Table 3.12: **Half-precision Floating-point Instructions Set**

Instruction	Description	Execution latency
Operation Instructions		
FADD.H	A half-precision floating-point add instruction.	3
FSUB.H	A half-precision floating-point subtract instruction.	3
FMUL.H	A half-precision floating-point multiply instruction.	3
FMADD.H	A half-precision floating-point multiply-add instruction.	4
FMSUB.H	A half-precision floating-point multiply-subtract instruction.	4
FNMADD.H	A half-precision floating-point negate- (multiply-add) instruction.	4
FNMSUB.H	A half-precision floating-point negate- (multiply-subtract) instruction.	4
FDIV.H	A half-precision floating-point divide instruction.	8-11
FSQRT.H	A half-precision floating-point square-root instruction.	8-11
Sign Injection Instructions		
FSGNJ.H	A half-precision floating-point sign-injection instruction	3
FSGNJN.H	A half-precision floating-point negate sign-injection instruction	3
FSGNJX.H	A half-precision floating-point XOR sign-injection instruction	3
Data Transfer Instructions		
FMV.X.H	A half-precision floating-point read transfer instruction	1+1 in split execution
FMV.H.X	A half-precision floating-point write transfer instruction	3
Compare Instructions		
FMIN.H	A half-precision floating-point MIN instruction	3
FMAX.H	A half-precision floating-point MAX instruction.	3
FEQ.H	A half-precision floating-point compare equal instruction.	1+1 in split execution
FLT.H	A half-precision floating-point compare less than instruction.	1+1 in split execution
FLE.H	A half-precision floating-point compare less than or equal to instruction.	1+1 in split execution
Data Type Conversion Instructions		
FCVT.S.H	An instruction that converts a half-precision floating-point number into a single-precision floating-point number.	3
FCVT.H.S	An instruction that converts a single-precision floating-point number into a half-precision floating-point number.	3

Continued on next page

Table 3.12 – continued from previous page

Instruction	Description	Execution latency
FCVT.D.H	An instruction that converts a half-precision floating-point number into a double-precision floating-point number.	3
FCVT.H.D	An instruction that converts a double-precision floating-point number into a half-precision floating-point number.	3
FCVT.W.H	An instruction that converts a half-precision floating-point number into a signed integer.	3+1 in split execution
FCVT.WU.H	An instruction that converts a half-precision floating-point number into an unsigned integer.	3+1 in split execution
FCVT.H.W	An instruction that converts a signed integer into a half-precision floating-point number	3, in sequential execution
FCVT.H.WU	The instruction that converts an unsigned integer into a half-precision floating-point number.	3, in sequential execution
FCVT.L.H	An instruction that converts a half-precision floating-point number into a signed long integer.	3+1 in split execution
FCVT.LU.H	An instruction that converts a half-precision floating-point number into an unsigned long integer.	3+1 in split execution
FCVT.H.L	An instruction that converts a signed long integer into a half-precision floating-point number.	3, in sequential execution
FCVT.H.LU	An instruction that converts an unsigned long integer into a half-precision floating-point number.	3, in sequential execution
Memory Store Instructions		
FLH	A half-precision floating-point load instruction	WEAK ORDER LOAD: >=3 STORE: 1 STRONG ORDER Aperiodic
FSH	A half-precision floating-point store instruction.	Same as above
Floating-point Classification Instructions		
FCLASS.H	A single-precision floating-point classification instruction	1+1

For specific instruction descriptions and definitions, please refer to *Appendix B-6 Half-precision Floating-point Instructions* .

CHAPTER 4

CPU Mode and Register

4.1 CPU Mode

C920 supports three RISC-V **privilege modes**: Machine Mode (M-mode), Supervisor Mode (S-mode), and User Mode (U-mode). C920 runs programs in M-mode after reset. The three modes correspond to different operation privileges and differ in the following aspects:

1. Register access
2. Use of privileged instructions
3. Memory access

The U-mode provides the lowest privileges

User programs are only allowed to access only the registers specific to the U-mode, which prevents user programs from accessing privileged information. The operating system manages and serves user programs by coordinating their behaviors.

The S-mode provides higher privileges than the U-mode but lower privileges than the M-mode

Programs running in S-mode are not allowed to access control registers specific to the M-mode and are limited by physical memory protection (PMP). The page-based virtual memory acts as the core of the S-mode.

The M-mode has the highest privileges

Programs running in M-mode have full access to memory, I/O resources, and underlying features required for starting and configuring the system. By default, CPU switches to the M-mode to respond to exceptions and interrupts that occur in any mode unless the exceptions and interrupts are delegated.

Most instructions can run in all the three modes. However, some privileged instructions with major impacts on systems can run only in S-mode or M-mode. For specific information, please refer to *Appendix A Standard Instructions* and *Appendix B Xuantie Extended Instructions* to check execution permission of instructions.

Processor' s operating mode changes in response to an exception. (The privilege mode in which an exception occurs is different from that in which the CPU responds to the exception.) CPU switches to a higher privilege mode to respond to the exception, and switches back to the lower privilege mode after the exception is handled.

4.2 Register View

The register view of C920 is shown in Fig. 4.1 :

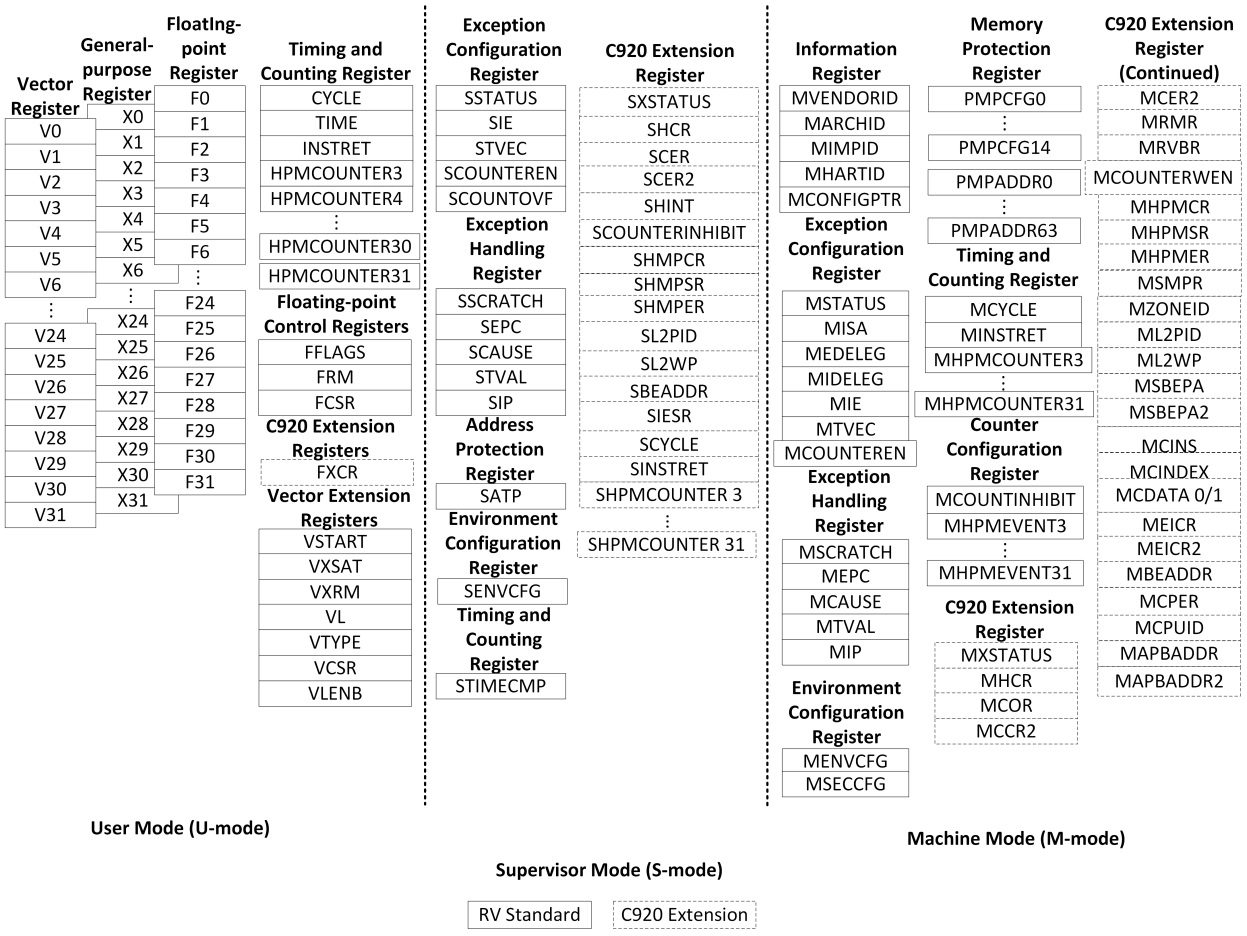


Fig. 4.1: **Register View**

4.3 General-purpose Registers

C920 provides thirty-two 64-bit general-purpose registers, sharing the same features and definations as those defined in RISC-V. For specific information, please check Table 4.1.

Table 4.1: **General-purpose Registers**

Register	ABI Name	Description
x0	zero	A hardwired zero register
x1	ra	A return address register
x2	sp	A stack pointer register
x3	gp	A global pointer register
x4	tp	A thread pointer register
x5	t0	A temporary/standby link register
x6-7	t1-2	Temporary registers
x8	s0/fp	A reserved/frame pointer register
x9	s1	A reserved register
x10-11	a0-1	Function argument/Return value registers
x12-17	a2-7	Function argument registers
x18-27	s2-11	Reserved register
x28-31	t3-6	Temporary registers

The general-purpose registers are designed to sore instruction operands, instruction execution results, and address information.

4.4 Floating-point Registers

In addition to standard RV64F instructions, C920 also supports floating-point half-precision computing and provides 32 independent 64-bit floating-point registers. These registers are accessible in U-mode, S-mode, and M-mode.

Table 4.2: **Floating-point Registers**

Register	ABI Name	Description
f0-7	ft0-7	Floating-point temporary registers.
f8-9	fs0-1	Floating-point reserved registers.
f10-11	fa0-1	Floating-point argument/return value registers.
f12-17	fa2-7	Floating-point argument registers.
f18-27	fs2-11	Floating-point reserved registers.
f28-31	ft8-11	Floating-point temporary registers.

Unlike general-purpose register x0, floating-point register f0 is not hardwired to 0, and its bit values are variable like other floating-point registers. A single-precision floating-point number occupies only the lower 32 bits of a 64-bit floating-point register, and the upper 32 bits must be set to 1; Otherwise, the number will be considered as nonnumeric. A half-precision floating-point number occupies only the lower 16 bits of a 64-bit floating-point register, and the upper 48 bits must be set to 1; Otherwise, the number will be considered as nonnumeric.

Increasing independent floating-point registers could expand the register capacity and bandwidth, so as to improve CPU performance. Besides, it is necessary to add floating-point load and store instructions at the same time, as well as instructions for transferring data between floating-point and general-purpose registers.

4.4.1 Transfer Data between Floating-point and General-purpose Registers

Data can be transferred between floating-point and general-purpose registers by transferring instructions through floating-point registers. And the transferring instructions include:

- FMV.X.H/FMV.H.X: half-precision transfer instruction of floating-point registers.
- FMV.X.W/FMV.W.X: single-precision transfer instruction of floating-point registers.
- FMV.X.D/FMV.D.X: double-precision transfer instruction of floating-point registers.

When half-precision/single-precision/double-precision data is transferred from a general-purpose register to a floating-point register, the data format remains unchanged. Therefore, a program can directly use these registers without converting their types.

For specific information, please refer to *Appendix A-4 F instructions* .

4.4.2 Maintain the Consistency of Register Precision

Floating-point registers can store half-precision, single-precision, double-precision and integer data. For example, the type of data stored in floating-point register f1 depends on the last write operation, which may be any of the four data types.

The FPU does not detect data formats based on hardware, and the hardware' s parsing of the data format in the floating-point register depends only on the floating-point instruction itself, regardless of the data format of the last write operation to this register. It is entirely up to the compiler or program itself to ensure the consistency of the data precision in registers.

4.5 Vector Register

C920 contains 32 independent vector architecture registers with 128-bit width, which are accessible in normal U-mode, S-mode, and M-mode. The vector registers enable data exchange with general-purpose registers/floating-point registers through vector transfer instructions.

4.5.1 Transfer Data Between Vector Registers and General-Purpose Registers

The data transfer between vector registers and general-purpose registers can be achieved by transferring instructions through vector integer register. And the transferring instructions include:

- VMV.V.X: Vector move from integer to vector instruction
- VMV.S.X: Vector move from integer to vector scalar element instruction

4.5.2 Transfer Data between Vector Registers and Floating-point Registers

The data transfer between vector registers and floating-point registers can be achieved by transferring instructions through vector floating-point register. And the transferring instructions include:

- VFMV.V.F: Vector move from floating-point to vector instruction
- VFMV.F.S: Vector move from scalar element in vector to floating-point instruction
- VFMV.S.F: Floating-Point move to scalar element in vector instruction

4.6 System Control Registers

4.6.1 Standard Control Registers

This section describes RISC-V standard control registers implemented in C920, categorized according to M-mode, S-mode, and Umode.

The RISC-V standard M-mode control registers implemented in C920 are described in Table 4.3.

Table 4.3: **RISC-V Standard M-mode Control Registers**

Register	Read/Write Permission	ID	Description
M-mode Information Registers			
mvendorid	Read-only in M-mode	0xF11	A vendor ID register.
marchid	Read-only in M-mode	0xF12	An architecture ID register.
mimpid	Read-only in M-mode	0xF13	An M-mode implementation ID register.
mhartid	Read-only in M-mode	0xF14	An M-mode hart ID register.
mconfigptr	Read-only in M-mode	0xF15	An M-mode configuration data structure pointer.
M-mode Exception Configuration Registers			
mstatus	Read/Write in M-mode	0x300	An M-mode CPU status register.
misa	Read/Write in M-mode	0x301	An M-mode CPU instruction set attribute register.
medeleg	Read/Write in M-mode	0x302	An M-mode exception delegation register.
mideleg	Read/Write in M-mode	0x303	An M-mode interrupt delegation register.
mie	Read/Write in M-mode	0x304	An M-mode interrupt enable register.
mtvec	Read/Write in M-mode	0x305	An M-mode vector base address register.
mcounteren	Read/Write in M-mode	0x306	An M-mode counter enable register.
M-mode Exception Handling Registers			
mscratch	Read/Write in M-mode	0x340	An M-mode temporary data backup register upon exceptions.
mepc	Read/Write in M-mode	0x341	An M-mode exception reserve program counter.
mcause	Read/Write in M-mode	0x342	An M-mode exception event cause register.
mtval	Read/Write in M-mode	0x343	An M-mode exception event vector register.
mip	Read/Write in M-mode	0x344	An M-mode interrupt pending state register.
M-mode Environment Configuration Registers			
menvcfg	Read/Write in M-mode	0x30A	An M-mode environment configuration register.
mseccfg	Read/Write in M-mode	0x747	An M-mode security configuration register.
M-mode Memory Protection Registers			

Continued on next page

Table 4.3 – continued from previous page

Register	Read/Write Permission	ID	Description
pmpcfg0	Read/Write in M-mode	0x3A0	Physical memory protection configuration register 0.
pmpcfg2	Read/Write in M-mode	0x3A2	Physical memory protection configuration register 2.
……			
pmpcfg14	Read/Write in M-mode	0x3AE	Physical memory protection configuration register 14.
pmpaddr0	Read/Write in M-mode	0x3B0	Physical memory protection base address register 0.
……			
pmpaddr63	Read/Write in M-mode	0x3EF	Physical memory protection base address register 63.
M-mode Timier and Counter Registers			
mcycle	Read/Write in M-mode	0xB00	An M-mode cycle counter.
minstret	Read/Write in M-mode	0xB02	An M-mode retired instruction counter.
mhpmcounter3	Read/Write in M-mode	0xB03	Machine-mode counter 3.
……			
mhpmcounter31	Read/Write in M-mode	0xB1F	M-mode counter 31.
M-mode Counter Configuration Registers			
mcountinhibit	Read/Write in M-mode	0x320	M-mode count disable register.
mhpmevent3	Read/Write in M-mode	0x323	M-mode performance monitor event select register 3.
……			
mhpmevent31	Read/Write in M-mode	0x33F	M-mode performance monitor event select register 31.
Debug/Trace Registers (Shared with Debug Mode)			
tselect	Read/Write in M-mode	0x7A0	Debug/trace trigger select register.
tdata1	Read/Write in M-mode	0x7A1	Debug/trace trigger data register 1.
tdata2	Read/Write in M-mode	0x7A2	Debug/trace trigger data register 2.
tdata3	Read/Write in M-mode	0x7A3	Debug/trace trigger data register 3.
tinfo	Read-only in M-mode	0x7A4	Debug/trace trigger information register.
tcontrol	Read/Write in M-mode	0x7A5	Debug/trace trigger control register.
mcontext	Read/Write in M-mode	0x7A8	M-mode content register.
Debug Mode Registers/ Trace Registers			
dcsr	Read/Write in debug mode	0x7B0	Debug mode control and status register.
dpc	Read/Write in debug mode	0x7B1	Debug mode program Counter.
dscratch0	Read/Write in debug mode	0x7B2	Debug mode temporary data backup register 0
dscratch1	Read/Write in debug mode	0x7B3	Debug mode temporary data backup register 1

The RISC-V standard S-mode control registers implemented in C920 are illustrated in Table 4.4.

Table 4.4: **RISC-V Standard S-mode Control Registers**

Register	Read/Write Permission	ID	Description
S-mode CPU Control and Status Extension Registers			
sstatus	Read/Write in S-mode	0x100	An S-mode CPU status register.
sie	Read/Write in S-mode	0x104	An S-mode interrupt enable control register.
stvec	Read/Write in S-mode	0x105	An S-mode vector base address register.
scounteren	Read/Write in S-mode	0x106	An S-mode counter enable control register.
scountovf	Read-only in S-mode	0xDA0	An S-mode counter interrupt overflow register.
S-mode Environment Configuration Register			
senvcfg	Read/Write in S-mode	0x10a	An S-mode environment configuration register.
S-mode Exception Handling registers			
sscratch	Read/Write in S-mode	0x140	An S-mode temporary data backup register upon exceptions.
sepc	Read/Write in S-mode	0x141	An S-mode exception reserved program counter.
scause	Read/Write in S-mode	0x142	An S-mode exception event cause register.
stval	Read/Write in S-mode	0x143	An S-mode exception event vector register.
sip	Read/Write in S-mode	0x144	An S-mode interrupt pending state register.
S-mode Address Protection Registers			
satp	Read/Write in S-mode	0x180	An S-mode virtual address translation and protection register.
S-mode Address Debug Registers			
scontext	Read/Write in S-mode	0x5A8	An S-mode scenario register.
S-mode Timer and Counter Registers			
stimecmp	Read/Write in S-mode	0x14D	An S-mode timer interrupt comparison value register.

The RISC-V standard U-mode control registers implemented in C920 are described in *RISC-V Standard U-mode Control Registers*.

Table 4.5: **RISC-V Standard U-mode Control Registers**

Register	Read/Write Permission	ID	Description
U-mode Floating-point Control Registers			
fflags	Read/Write in U-mode	0x001	A floating-point accrued exception status register.
frm	Read/Write in U-mode	0x002	A floating-point dynamic rounding mode control register.
fcsr	Read/Write in U-mode	0x003	A floating-point control status register.
U-mode Timer and Counter Registers			
cycle	Read-only in U-mode	0xC00	An U-mode cycle counter.
time	Read-only in U-mode	0xC01	An U-mode timer.
instret	Read-only in U-mode	0xC02	An U-mode retired instruction counter.
hpmcounter3	Read-only in U-mode	0xC03	U-mode counter 3
……			
hpmcounter31	Read-only in U-mode	0xC1F	U-mode counter 31.
Vector Extension Registers			
vstart	Read/Write in U-mode	0x008	A Vector start position register.
vxsat	Read/Write in U-mode	0x009	A fixed-point overflow flag register.
vxrm	Read/Write in U-mode	0x00A	A fixed-point rounding mode register.

Continued on next page

Table 4.5 – continued from previous page

Register	Read/Write Permission	ID	Description
vcsr	Read/Write in U-mode	0x00F	A vector control and status register.
vl	Read-only in U-mode	0xC20	A vector length register.
vtype	Read-only in U-mode	0xC21	A vector data type register.
vlenb	Read-only in U-mode	0xC22	A vector byte size register.

4.6.2 Extended Control Registers

This section describes extended control registers implemented in C920, categorized according to M-mode, S-mode, and U-mode.

The extended M-mode control registers of C920 are described in Table 4.6.

Table 4.6: **Extended M-mode Control Registers of C920**

Register	Read/Write Permission	ID	Description
M-mode CPU Control and Status Extension Registers			
mxstatus	Read/Write in M-mode	0x7C0	An extended M-mode status register.
mhcr	Read/Write in M-mode	0x7C1	An M-mode hardware configuration register.
mcor	Read/Write in M-mode	0x7C2	An M-mode hardware operation register.
mccr2	Read/Write in M-mode	0x7C3	An M-mode L2 cache control register.
mcer2	Read/Write in M-mode	0x7C4	An M-mode L2 cache ECC register.
mhint	Read/Write in M-mode	0x7C5	An M-mode implicit operation register.
mrmr	Read/Write in M-mode	0x7C6	An M-mode reset register.
mrvbr	Read/Write in M-mode	0x7C7	An M-mode reset vector base address register.
mcer	Read/Write in M-mode	0x7C8	An M-mode L1Cache ECC register.
mcounterwen	Read/Write in M-mode	0x7C9	An M-mode counter write enable register.
M-mode Extension Registers Group 2			
mhpmcr	Read/Write in M-mode	0x7F0	A performance monitor control address register.
mhpmsr	Read/Write in M-mode	0x7F1	A performance monitor start trigger register.
mhpmer	Read/Write in M-mode	0x7F2	A performance monitor stop trigger register.
msmpr	Read/Write in M-mode	0x7F3	A Snoop monitor enable register.
mzoneid	Read/Write in M-mode	0x7F5	A zone id register.
ml2pid	Read/Write in M-mode	0x7F6	A performance monitor refill ID register.
ml2wp	Read/Write in M-mode	0x7F7	A L2 fine-grained configuration register.
msbepa	Read/Write in M-mode	0x7FB	An M-mode L1Cache ECC single bit error physical address register.
msbepa2	Read/Write in M-mode	0x7FC	An M-mode L2Cache ECC single bit error physical address register.
M-mode Cache Access Extension Registers			
mcins	Read/Write in M-mode	0x7D2	An M-mode cache instruction register.
mcindex	Read/Write in M-mode	0x7D3	An M-mode cache access index register.
mcdata0	Read/Write in M-mode	0x7D4	An M-mode cache data register 0.

Continued on next page

Table 4.6 – continued from previous page

Register	Read/Write Permission	ID	Description
mcdata1	Read/Write in M-mode	0x7D5	An M-mode cache data register 1.
meicr	Read/Write in M-mode	0x7D6	An L1 Cache hardware error injection register.
meicr2	Read/Write in M-mode	0x7D7	An L2 Cache hardware error injection register.
mbeaddr	Read/Write in M-mode	0x7D8	An L1 LD BUS ERROR address register.
mcper	Read/Write in M-mode	0x7D9	A TEE cache permission control register.
M-mode CPU Model Extension Registers			
mcpuid	Read-only in M-mode	0xFC0	An M-mode processor ID register.
mapbaddr	Read-only in M-mode	0xFC1	An on-chip bus base address register.
mapbaddr2	Read-only in M-mode	0xFC3	An on-chip system interconnect base address register.
M-mode Debug Extension Control Register			
mhaltcause	Read-only in M-mode	0x7FE0	An HALT cause register.
mdbginfo	Read-only in M-mode	0x7FE1	A Debug information register.
mpcfifo	Read-only in M-mode	0x7FE2	A pcfifo register.
mdbginfo2	Read-only in M-mode	0x7FE3	Debug information register 2.

Note:

"mrmr" register has been removed from C920 (R1S4 and above). The software can still access this register, with the result that read as zero and write invalid without triggering an exception.

For specific definations and features of registers, please refer to *Appendix C-1 RISC-V Standard Machine Mode Control and Status Registers*.

The extended S-mode control registers of C920 are described in Table 4.7.

Table 4.7: **Extended S-mode Control Registers of C920**

Register	Read/Write Permission	ID	Description
S-mode Processor Control and Status Extension Registers			
sxtatus	Read/Write in S-mode	0x5C0	An S-mode extension status register.
shcr	Read/Write in S-mode	0x5C1	An S-mode hardware control register.
scer2	Read-only in S-mode	0x5C2	An S-mode L2Cache ECC register.
scer	Read-only in S-mode	0x5C3	An S-mode L1Cache ECC register.
shint	Read/Write in S-mode	0x5C6	An S-mode implicit register.
shint2	Read/Write in S-mode	0x5C7	S-mode implicit register 2.
shpminhibit	Read/Write in S-mode	0x5C8	An S-mode count inhibit register.
shpmcr	Read/Write in S-mode	0x5C9	An S-mode performance monitor control register.
shpmsr	Read/Write in S-mode	0x5CA	An S-mode performance monitor control start trigger register.
shpmer	Read/Write in S-mode	0x5CB	A S-mode performance monitor start and stop trigger register.
sl2pid	Read/Write in S-mode	0x5CC	A fine-grained refill register.
sl2wp	Read/Write in S-mode	0x5CD	An L2 fine-grained configuration register.
sbeaddr	Read/Write in S-mode	0x5D0	An S-mode BUS error register.

Continued on next page

Table 4.7 – continued from previous page

Register	Read/Write Permission	ID	Description
ssbepa	Read/Write in S-mode	0x5D1	An S-mode L1 Cache ECC single-bit error physical address register.
ssbepa2	Read/Write in S-mode	0x5D1	An S-mode L2 Cache ECC single-bit error physical address register.
scycle	Read/Write in S-mode	0x5E0	An S-mode cycle counter
sinstret	Read/Write in S-mode	0x5E2	An S-mode instruction retirement counter.
shpmcounter3	Read/Write in S-mode	0x5F3	S-mode counter 3
……			
shpmcounter31	Read/Write in S-mode	0x5FF	S-mode counter 31.

For specific definations and features of registers, please refer to *Appendix C-2 RISC-V Standard S-mode Control Register*.

The extended U-mode control registers of C920 are described in Table 4.8 .

Table 4.8: **Extended U-mode control registers of C920**

Register	Read/Write Permission	ID	Description
Extended U-mode Floating-point Control Registers			
fxcr	Read/Write in U-mode	0x800	An extended U-mode floating-point control register.

For specific definations and features of registers, please refer to *Appendix C-3 RISC-V Standard U-mode Control Register*.

4.7 Data Format

4.7.1 Integer Data Format

The values within a register does not inherently have a big-endian or little-endian distinction; rather, it is distinguished as being either signed or unsigned. And the format is consistently arranged from right to left, representing the least significant bit to the most significant bit (MSB), as is shown in Fig. 4.2 .

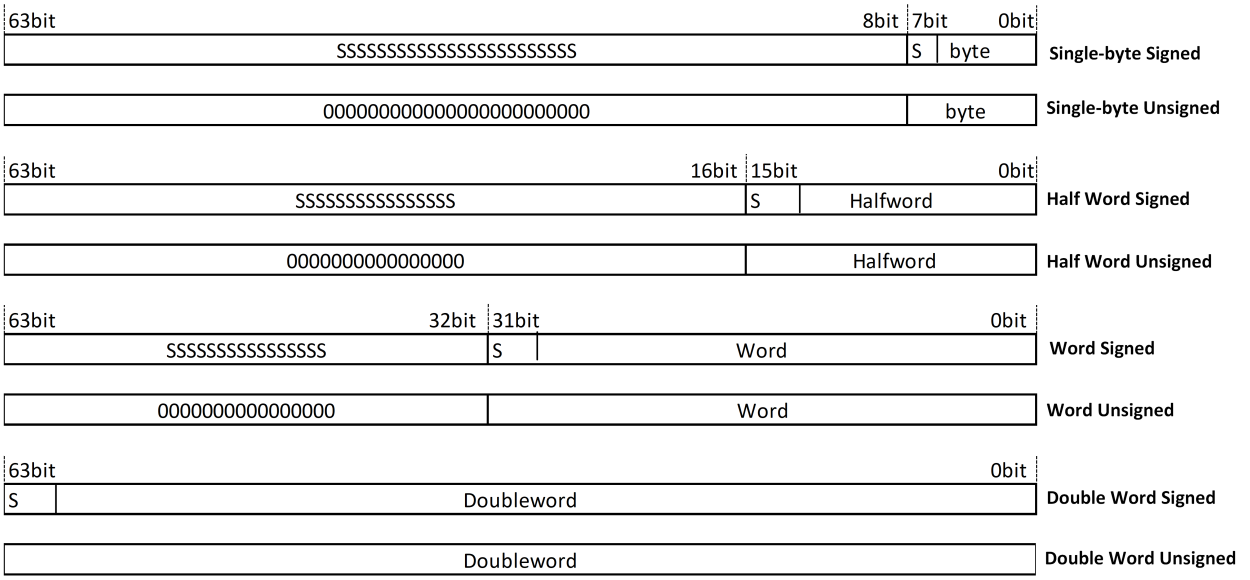


Fig. 4.2: **Integer Data Structure in Registers**

4.7.2 Floating-point Data Format

Floating-point Units (FPU) of C920 comply with RISC-V standard and the ANSI/IEEE 754-2008 floating-point standard, support for half-precision, single-precision and double-precision floating-point operations. And the data format is shown in Fig. 4.3. Single-precision data occupies only the lower 32 bits of a 64-bit floating-point register, and the upper 32 bits must be set to 1; Otherwise, the data will be considered as nonnumeric. While half-precision data occupies only the lower 16 bits of a 64-bit floating-point register, and the upper 48 bits must be set to 1; Otherwise, the data will be considered nonnumeric.

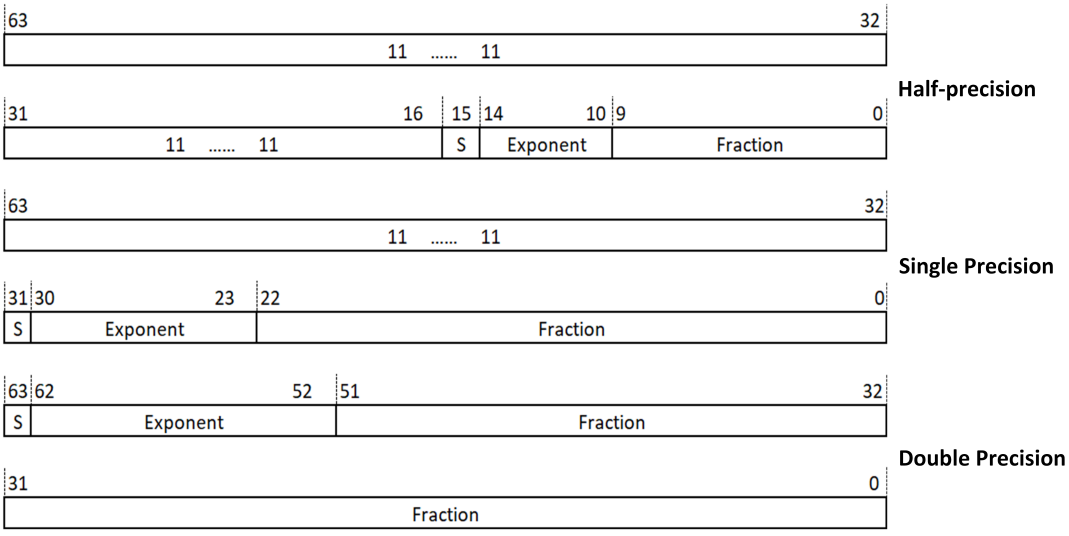


Fig. 4.3: **Floating-point Data Structure in Registers**

4.7.3 Vector Data Format

The vector register has a width of 128 bits, and the number of elements contained in the vector register is determined by the current vector element width. C920 supports vector element widths of 8 bits, 16 bits, 32 bits, and 64 bits. The data arrangement of the vector register with different vector element widths is shown in Fig. 4.4.

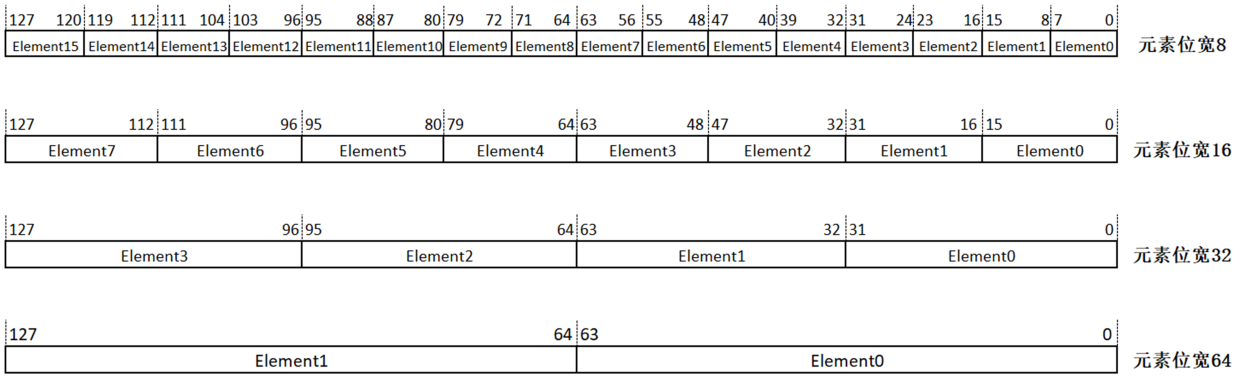


Fig. 4.4: **Organization Structure of Vector Data in Registers**

4.8 Big-endian and Little-endian

The concepts of big-endian and little-endian are proposed with respect to the data store format of memories. In the big-endian mode, the most significant byte of an address is stored to the lower bits in physical memory. While in the little-endian mode, the most significant byte of an address is stored to the upper bits in physical memory. The data format is shown in Fig. 4.5.

C920 supports only the little-endian mode and binary integers with standard complements. The length of each instruction operand can be explicitly encoded in programs (load/store instructions) or implicitly indicated in instruction operations (index operation and byte extraction). In general, an instruction receives a 64-bit operand and generates a 64-bit result.

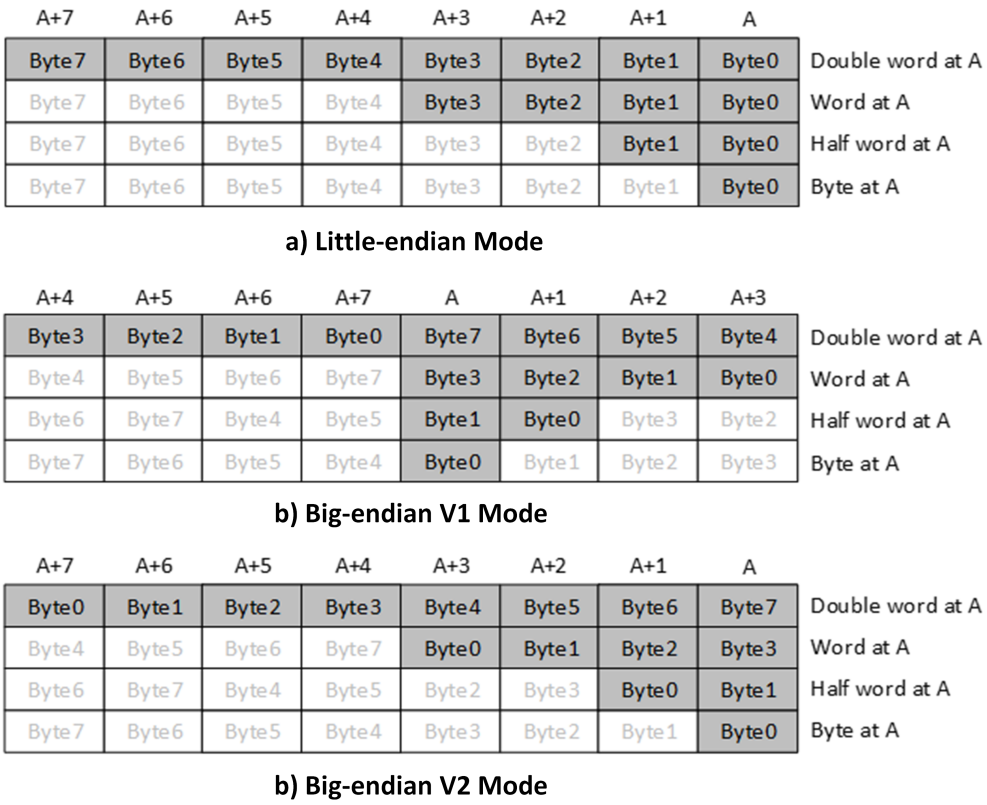


Fig. 4.5: **Data Structure in Memory**

CHAPTER 5

Exception and Interrupt

5.1 Overview

Exception handling (including instruction exception and external exception) is an essential feature of CPU. CPU is enabled to respond to these exceptions once they occur, including hardware errors, instruction execution errors and user program request services.

The key of exception handling is to save the operating status of CPU when an exception occurs and resume the status when CPU exits exception handling. Exceptions can be identified in all stages of the instruction pipeline. CPU hardware ensures that subsequent instructions do not change CPU status. Exceptions are handled at the boundary of an instruction. To be specific, CPU responds to the exceptions when the instruction retires, and saves the address of the to-be-executed instruction when CPU exits exception handling. CPU does not handle the exceptions until the instruction retires, even if exceptions are identified before an instruction retires. To ensure proper functioning of programs, CPU needs to avoid repeatedly running the executed instructions after exception handling is completed.

In machine mode (M-mode), CPU responds to an exception in the following procedure: (The term "exception" generally refers to instruction exceptions and external interrupts)

Step 1: Save PC to the mepc register.

Step 2: Update the mcause and mtval registers based on the exception type.

Step 3: Save the interrupt-enable (MIE) bit of the mstatus register to the MPIE bit, clear MIE bit, and prohibit responses to interrupts.

Step 4: Save the privilege mode before the exception occurs to MPP bit in the mstatus register, and switch to M-mode.

Step 5: Obtain the entry address of exception program based on the base address and mode in the mtvec register, and begin the execution of the first instructionrun of the exception program in sequence.

C920 conforms to the exception vector table defined in RISC-V, as shown in Table 5.1.

Table 5.1: **Exception and Interrupt Vector Assignment**

Interrupt Flag	Exception Vector ID	Description
1	0	Unavailable
1	1	A software interrupt in supervisor mode (S-mode)
1	2	Reserved
1	3	A software interrupt in M-mode
1	4	Unavailable
1	5	A timer interrupt in S-mode
1	6	Reserved
1	7	A timer interrupt in M-mode
1	8	Unavailable
1	9	An external interrupt in S-mode
1	10	Reserved
1	11	An external interrupt in M-mode
1	13	A performance detection overflow interrupt (i.e., configuring a performance detection unit)
1	16	L1 Data cache ECC interrupt (if configuring ECC)
1	Others	Reserved
0	0	Unavailable
0	1	A fetch instruction access error exception
0	2	An illegal instruction exception
0	3	A debug breakpoint exception
0	4	A load instruction unaligned access exception
0	5	A load instruction access error exception
0	6	A store/atomic instruction unaligned access exception
0	7	A store/atomic instruction access error exception
0	8	A user-mode (U-mode) environment call exception
0	9	An S-mode environment call exception
0	10	Reserved
0	11	An M-mode environment call exception
0	12	An instruction fetch page error exception
0	13	A load instruction page error exception
0	14	Reserved
0	15	A store/atomic instruction page error exception
0	>= 16	Reserved

C920 supports the delegation of exceptions and interrupts. When an exception or interrupt occurs in S-mode, CPU needs to switch to M-mode for handling, which causes performance loss of CPU. Delegation enables CPU to respond to exceptions and interrupts in S-mode. While exceptions that occur in M-mode are not delegated, but still handled

in M-mode. Interrupts that occur in M-mode can be delegated to the S-mode for handling, except the external interrupts, software interrupts, and timer interrupts that occur in M-mode. In M-mode, CPU does not respond to delegated interrupts.

In S-mode and U-mode, CPU can respond to all eligible interrupts and exceptions. CPU responds to undelegated exceptions and interrupts in M-mode, and updates the M-mode exception handling registers. CPU responds to delegated exceptions and interrupts in S-mode, and updates the S-mode exception handling registers.

5.2 Exception

5.2.1 Exception Handling

In M-mode, CPU responds to exceptions in the following specific procedure: (The term "exception" specifically refers to illegal instructionS and access error.)

Step 1: Save the exception PC to mepc register.

Step 2: Set the interrupt flag in the mcause register to 0, write the exception ID to the mcause register, and update the mtval register based on the rules defined in Table 5.2.

Table 5.2: **Updates of Mtval upon Exception Occurrence**

Exception Vector ID	Exception	Mtval Update
1	A fetch instruction access error exception	Virtual address accessed by the fetch instruction
2	An illegal instruction exception	Instruction code
3	A debug breakpoint exception	0
4	A load instruction unaligned access exception	Virtual address accessed by the load instruction
5	A load instruction access error exception	0
6	An store/atomic instruction unaligned access exception	Virtual address accessed by the store/atomic instruction
7	An Store/atomic instruction access error exception	0
8	An U-mode environment call exception	0
9	An S-mode environment call exception	0
11	An M-mode environment call exception	0
12	A fetch instruction page error exception	Virtual address accessed by the fetch instruction
13	A load instruction page access exception	Virtual address accessed by the load instruction
15	An store/atomic instruction page error exception	Virtual address accessed by the store/atomic instruction

Step 3: Save MIE bit of the mstatus register to the MPIE field, clear the MIE field, and prohibit responses to interrupts.

Step 4: Save the privilege mode before the exception occurs to MPP field of the mstatus register, and switch to the M-mode.

Step 5: PC fetches the instruction from the address specified by mtvec.Base and executes it. And the instruction is usually a jump instruction for jumping to the top-level handler function. This function analyzes the mcause to obtain the exception code and calls the corresponding handler function of that code.

5.2.2 Exception Return

An exception return can be achieved by executing mret instruction. At this point, CPU performs the following operations:

- Restore the mepc register to PC. (The mepc register stores PC where the exception occurs. Adjusting the mepc register enables to skip the exception instruction; Otherwise, the exception instruction will be executed again.)
- Restore mstatus.MIE from mstatus.MPIE.
- Restore the privilege mode from mstatus.MPP before the exception occurred.

5.2.3 Imprecise Exceptions

In rare cases, CPU may encounter "imprecise exceptions" . An imprecise exception means that the mepc register does not point to the instruction triggering the exception when the exception occurs. For example, the bus returns an error after the CPU executes a load instruction. Beacuse pipelines feature fast instruction retirement, and the load instruction may have retired when the bus returns the error. And the mepc register points to the subsequent instruction instead of the load instruction.

However, imprecise exceptions rarely occur in practical systems. Once it does occur, it signifies that the system has encountered a fatal error.

5.3 Interrupt

5.3.1 Interrupt Priorities

When multiple interrupt requests occur simultaneously, the priority is determined in the following order (in descending order):

- L1 ECC interrupt
- M-mode external interrupt
- M-mode software interrupt
- M-mode timer interrupt
- S-mode external interrupt
- S-mode software interrupt
- S-mode timer interrupt

- Performance Monitoring Unit (PMU) overflow interrupt
- L1 ECC interrupt (delegated)
- S-mode external interrupt (delegated)
- S-mode software interrupt (delegated)
- S-mode timer interrupt (delegated)
- PMU overflow interrupt (delegated)

5.3.2 Interrupt Response

In M-mode, the CPU responds to an interrupt in the following specific procedure:

Step 1: Execute the current instruction and save the PC of the next instruction to the mepc register.

Step 2: Set the interrupt flag of the mcause register to 1, write the interrupt ID into the mcause register, and update the mtval register to 0.

Step 3: Save the MIE bit of the mstatus register to the MPIE field, clear the MIE field, and prohibit responses to interrupts.

Step 4: Save the privilege mode before the exception occurs to the MPP field of mstatus, and switch to M-mode.

Step 5 (mtvec.Mode=0, direct interrupt): PC fetches the instruction from the address specified by mtvec.Base and executes it. And the instruction is usually a jump instruction for jumping to the top-level handler function. This function analyzes mcause to obtain the the vector ID and calls the corresponding handler function for that ID.

Step 5 (mtvec.Mode=1, vector interrupt): PC fetches and executes the instruction from mtvec.Base + 4 * vector ID and executes it. Typically, the fetched instruction is a jump instruction that jumps to handler function of the corresponding interrupt.

5.3.3 Interrupt Return

An interrupt return is accomplished by executing the mret instruction. In this case, CPU performs the following operations:

- Restore the mepc register to PC. (The mepc register stores the PC of the next instruction, so no adjustment is needed)
- Restore mstatus.MIE from mstatus.MPIE.
- Restore the privilege mode from mstatus.MPP before the interrupt occurred.

CHAPTER 6

Memory Model

6.1 Overview

6.1.1 Memory Attributes

C920 supports two memory types: Memory and Device, which are distinguished by Strongly Ordered (SO) bit. Memory supports speculative execution and out-of-order execution. It is further classified into cacheable memory and non-cacheable memory, based on the cacheable (C) attribute. And it is noted that SO region can not store instructions.

Table 6.1 describes the page attributes corresponding to different memory type.

Table 6.1: **Classification of Memory Type**

Memory Type	SO	C
Cacheable memory	0	1
Non-cacheable memory	0	0
Bufferable device	1	0
Non-bufferable device	1	0

The page attribute of an address is determined by sysmap.h, an extension configuration file of C920, which is open to users, and users can define the page attributes of different address segments according to their needs.

sysmap.h supports page attribute settings of 8 address spaces. The upper limit (exclusive) of the i-th (i = 0 to 7) address space is defined by the macro SYSMAP_BASE_ADDRi, and the lower limit (inclusive) is defined by SYSMAP_BASE_ADDR(i - 1), which is:

SYSMAP_BASE_ADDR(i - 1) <= Address of the i-th space address < SYSMAP_BASE_ADDRi

The lower limit of the i-th address space is 0x0. The memory address beyond the 8 address spaces of sysmap.h file are cacheable by default. Every boundary between the address spaces is aligned on a 4KB boundary. Therefore, the macro SYSMAP_BASE_ADDRi defines the upper 28 bits of an address.

The address attribute within the i-th (i = 0 to 7) address sapce is defined by the macro SYSMAP_FLAGi (i=0~7), the related address attribute layout is depicted in Fig. 6.1.

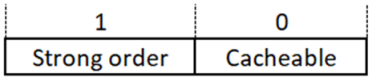


Fig. 6.1: **Address Attribute Format in sysmap.h File**

6.1.2 Memory Ordering Model

C920MP adopts a weak memory ordering model, which is defined as follows:

- Ordering of access to the same address is maintained among multiple cores, including read after read (RAR), write after write (WAW), write after read (WAR), add read after write (RAW).
- Weak ordering of access to different addresses is allowed among multiple cores, including RAR, WAW, WAR, add RAW.
- Atomic other-multi-copy is ensured: When one core obtains the write data of another core, it is required that the other cores can also obtain the write data at the same time. While when one core can obtain the write data of its own core, it is not required that other cores be able to obtain the write data at this time

Weak memory ordering causes inconsistency between the actual read/write order among multiple cores and the access order defined by the program. Therefore, C920 provides extended SYNC instructions to enforce memory access ordering in software.

SYNC instructions define the execution order of all instructions, ensuring that all instructions before the SYNC instruction must be completed before the SYNC instruction is executed. In addition, the SYNC instruction can additionally synchronize the instruction memory, which means the SYNC instruction clears the pipeline and re-fetches instructions after instructions preceding a SYNC instruction are executed. For detailed instructions, please refer to Table 6.2.

Table 6.2: **SYNC Instructions Description**

Mnemonic	Instruction Description	Scope
SYNC.IS	Synchronize data and instruction memory	Shareable
SYNC.I	Synchronize data and instruction memory	Non-shareable
SYNC.S	Synchronize data memory	Shareable
SYNC	Synchronize data memory	Non-shareable

6.1.3 SYSMAP Configuration Reference

- Definition of attributes of address space 0: 40' h0 <= addr0[39:0] <40' h100_0000, flg0 = 2' b01. This address space segment includes SRAM, configured with cacheable attribute, and the attribute definition is as follows:

```
`define CT_SYSMAP_BASE_ADDR0    28'h1000

`define CT_SYSMAP_FLG0          2'b01
```

- Definition of attributes of address space 1: 40' h100_0000 <= addr1[39:0] < 40' h200_0000, flg1 = 2' b10. This address space segment mainly corresponds to the write addresses of special function (i.e., the print function), configured with the strong_order attribute, ensuring that the operations are guaranteed to be issued from the core to the bus. The attribute definition is as follows:

```
`define CT_SYSMAP_BASE_ADDR1    28'h2000

`define CT_SYSMAP_FLG1          2'b10
```

- Definition of attributes of address space 2: 40' h200_0000 <= addr2[39:0] < 40' h8000_0000, flg2 = 2' b10. This address space segment mainly corresponds to APB interface, configured with the strong ordered attribute, ensuring that the operations are guaranteed to be issued from the core to the bus. The attribute definition is as follows:

```
`define CT_SYSMAP_BASE_ADDR2    28'h8_0000

`define CT_SYSMAP_FLG2          2'b10
```

- Definition of attributes of address space 3: 40' h8000_0000 <= addr3[39:0] < 40' hb000_0000, flg3 = 2' b01. The attribute definition is as follows:

```
`define CT_SYSMAP_BASE_ADDR3    28'hb_0000

`define CT_SYSMAP_FLG3          2'b01
```

- Definition of attributes of address space 4: 40' hb000_0000 <= addr4[39:0] < 40' hffff_f000, flg4 = 2' b10. The attribute definition is as follows:

```
`define CT_SYSMAP_BASE_ADDR4    28'hf_ffff

`define CT_SYSMAP_FLG4          2'b10
```

- Definition of attributes of address space 5: 40' hffff_f000 <= addr5[39:0] < 40' h40_0000_0000, flg5 = 2' b01. The attribute definition is as follows:


```
`define CT_SYSMAP_BASE_ADDR5    28'h400_0000

`define CT_SYSMAP_FLG5          2'b01
```

- Definition of attributes of address space 6: 40’ h 40_0000_0000 <= addr6[39:0] <40’ h50_0000_0000, flg6 = 2’ b10. The attribute definition is as follows:

```
`define CT_SYSMAP_BASE_ADDR6    28'h500_0000

`define CT_SYSMAP_FLG6          2'b10
```

- Definition of attributes of address space 7: 40’ h50_0000_0000, <= addr7[39:0] < 40’ hff_ffff_f000, flg7 = 2’ b01. The attribute definition is as follows:

```
`define CT_SYSMAP_BASE_ADDR7    28'hfff_ffff

`define CT_SYSMAP_FLG7          2'b01
```

6.2 MMU

6.2.1 MMU Overview

C920 memory management unit (MMU) complies with RISC-V SV39/SV48 standard. C920 MMU is specilized in the following main features:

- **Address translation**: Translates 39-bit/48-bit virtual addresses to 40-bit physical addresses.
- **Page protection**: Checks the read/write/execution permissions of page visitors.
- **Page attribute management**: Extends address attribute bits and obtains page attributes based on access addresses for further processing by system.

In C920, SXLEN is fixed at 64 bits, and MMU performs SV39/SV48 address translation according to 64-bit virtual addresses.

C920 supports U Mode configuration as either 32-bit or 64-bit. When UXL is configured as 32 bits:

- MMU performs SV39/SV48 address translation according to 32-bit virtual addresses.
- The upper 32 bits of the virtual address are not allowed to have non-zero values, otherwise a page fault exception will occur.

6.2.2 TLB Organization

MMU achieves the above functionality mainly by Translation Look-aside Buffer(TLB). TLB takes the virtual address used by CPU for memory access as input, checks the page attributes of TLB before performing the translation, and then outputs the corresponding physical address for that virtual address.

C920 MMU adopts two levels of TLB. The first level is uTLB, consisting of instruction I-uTLB and data D-uTLB, while the second level is jTLB. After the processor reset, hardware invalidates all entries in uTLB and jTLB, while software initialization is not required.

I-uTLB contains 16 fully associative entries, and can store a mixture of 4K, 2M, and 1G pages. When a fetch request hits I-uTLB, the physical address and corresponding permission attributes can be obtained in a single cycle.

D-uTLB contains 16 fully associative entries, and can store a mixture of 4K, 2M, and 1G pages. When a load/store request request hits D-uTLB, the physical address and corresponding permission attributes can be obtained in a single cycle.

jTLB is shared for instructions and data, with a 4-way set-associative structure. The table size can be configured as 1024 or 2048, and can store a mixture of 4K, 2M, and 1G pages. When a request misses the uTLB but hits the jTLB, the physical address and corresponding permission attributes will be returned within at least three cycles.

6.2.3 Address Translation Process

The main feature of MMU is to translate virtual addresses to physical addresses and perform corresponding permission check. Specific address mapping and corresponding permissions are configured by the operating system and stored in page tables. C920 implements address translation through indexing by at most three levels of page tables:

- Accessing the first-level page table to obtain the base address of the second-level page table and corresponding permission attributes;
- Accessing the second-level page table to obtain the base address of the third-level page table and corresponding permission attributes;
- Accessing the third-level page table to obtain the final physical address and corresponding permission attributes.

Each level of access may yield the final physical address, leaf table entry. The virtual page number (VPN) has 27 bits, divided into three 9-bit VPN[i] segments. Each access uses a portion of VPN for indexing. The contents of the leaf table entry (i.e., the physical address and corresponding permission attributes translated from virtual address translation) are cached in the TLB to accelerate address translation. If there is a miss mapping in the uTLB, the jTLB will be accessed. If there is a further miss mapping in the jTLB, MMU will initiate a Hardware Page Table Walk, accessing memory to obtain the final address translation result.

C920 page table is applied to storing the entry addresses of the next-level page tables or the physical information of the final page table. Its structure is shown in Fig. 6.2.

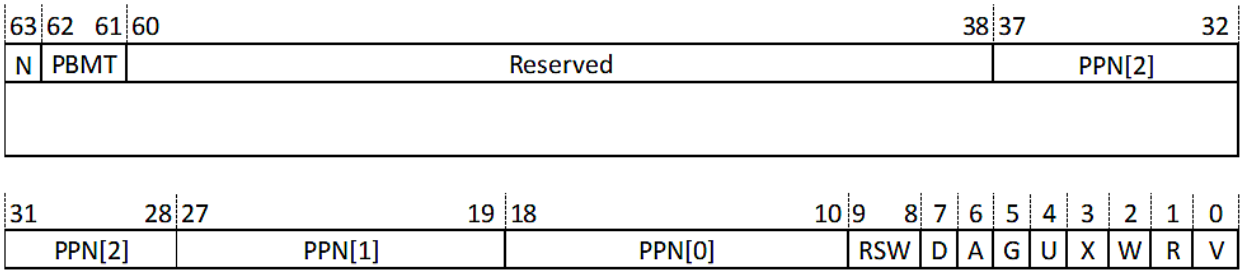


Fig. 6.2: **Page Table Structure**

Attributes of each bit in C920 page table:

N: NAPOT (Naturally Aligned Power of Two)

The N bit indicates that the current table entry is a NAPOT-size extension entry, which covers a contiguous region represented by a power of two. The specific size is defined in Table 6.3.

Table 6.3: **NAPOT Extended Page Size**

i	pte.ppn[i]	Description	pte.napot bits
0	x xxxx xxx1	Reserved	-
0	x xxxx xx1x	Reserved	-
0	x xxxx x1xx	Reserved	-
0	x xxxx 1000	64 KiB contiguous region	4
0	x xxxx 0xxx	Reserved	-
>=1	x xxxx xxxx	Reserved	-

The current standard only defines 64KiB. When the N bit is valid and the value of PPN is applied in other circumstances, then it is considered as a page fault with invalid table entry content. For specific information, please refer to *RISC-V SVNAPOT Standard Expansion Specification*.

PBMT: Page-Based Memory Types

The PBMT bit indicates the Physical Memory Attribute (PMA) of virtual address. The specific definitions are variable in different scenarios, as shown in Table 6.4 (**Note:** This feature requires enabling the PBMTE bit of menvcfg register.)

Table 6.4: **PBMT Definition**

Mode	Value	Requested Memory Attributes
PMA	0	None
NC	1	Non-cacheable, idempotent, weakly-ordered (RVWMO or RVTSO), main memory
IO	2	Non-cacheable, non-idempotent, strongly-ordered (I/O ordering), I/O
-	3	Reserved for future standard use

Notes:

- The PMA mode is compatible with the original system attribute settings when the PMA of virtual address was not defined, and follows the original PMA attributes, i.e., the sysmap attribute.
- In NC mode, force override the non-cacheable, idempotent, weakly ordered attributes, generally applied in memory.
- In IO mode, force override the non-cacheable, non-idempotent, strongly ordered attributes, generally applied in device.
- When the PBMT bit is set to 3, it is a reserved mode for future definition.

- For PMA attribute that is not defined in SVPBMT, it will follow the PMA attribute of the original physical address. However, it will adhere to the attribute defined in PBMT for forcefully overridden attribute.

For detailed information, please refer to *RISC-V SVPBMT Standard Extension Specification*.

RSW: Reserved for use by Supervisor software:The page table attribute bits are reserved for privileged software, not processed by hardware.

D: Dirty bit : Page is writeable.

A: Access bit: Page is accessible.

G: Global bit: Page page is globally scoped.

U: User bit: Page is dedicated to User Mode (U-mode).

X: Executable bit: Page is executable.

W: Writeable bit: Page is writeable.

R: Readable bit: Page is readable.

V: Valid bit: Page is valid.

The definition of XWR bit combination in C920 page table is illustrated in Table 6.5.

Table 6.5: **Definition of XWR Bit Combination in C920 Page Table**

X	W	R	Definition
0	0	0	The current page table is not leaf table entry
0	0	1	Readable pages
0	1	0	Reserved configuration, page fault
0	1	1	Readable, writeable pages
1	0	0	Fetchable pages
1	0	1	Readable, fetchable pages
1	1	0	Reserved configuration, page fault
1	1	1	Readable, writeable, fetchable pages

The detailed address translation process is described as follows:

If TLB is hit when CPU attempts to access a virtual address, CPU directly obtains the physical address and the corresponding attributes from the TLB. But if the TLB is missed, the MMU performs the following steps to translate the virtual address:

1. Obtain the access address {satp.ppn, VPN[2], 3' b0} of the L1 page table, based on satp.ppn and VPN[2]. Accessing Dcache/memory with this address retrieves the 64-bit L1 Page Table Entry (PTE);
2. Check whether the PTE conforms to the physical memory protection (PMP) permission. If it does not comply, the corresponding access error exception will be generated. If it complies, determine whether the X/W/R bit meets the condition of the leaf page table based on the rules shown in Table 6.5. If it complies with the corresponding rules, then the final physical address has been found. And continue to step 3. Otherwise, go back to step 1, replace satp.ppn with pte.ppn and vpn with the next-level vpn, and then splice 3' b0 to continue step 1 process;

3. After the leaf page table is found, compare the X/W/R/L bit in PMP with the X/W/R bit in PTE to obtain the minimum permissions, check the permissions, and write the content of PTE back to jTLB.
4. If permission violation is found in any PMP check, generate the corresponding access error exception based on the access type.
5. The corresponding page fault exception will be generated in the following 3 cases: If the leaf page table is found but the access type does not conform to the setting of the A/D/X/W/R/U-bit; No leaf page table is found after three accesses; An access error is generated during access to D-Cache/memory.
6. If a leaf page table is obtained in less than three accesses, it indicates that a large page table has been obtained. Check if Page Physical Number (PPN) of the large page table is aligned based on the page table size. If it is not aligned, a page fault exception will occur.

6.2.4 System Control Registers

6.2.4.1 MMU Address Translation Register (SATP)

SATP is an MMU control register of the SV39 specification.

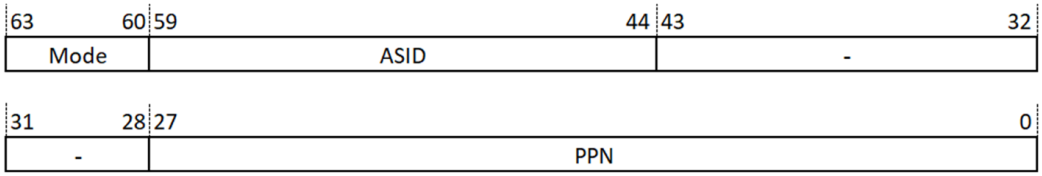


Fig. 6.3: **SATP Register Specification**

Mode - MMU Address Translation Mode

Table 6.6: **MMU Address Translation Mode**

RV64		
Value	**Name**	**Description**
0	Bare	No translation or protection
1-7	-	Reserved
8	Sv39	Page-based 39-bit virtual addressing
9	Sv48	Page-based 48-bit virtual addressing
10	Sv57	Reserved for page-based 57-bit virtual addressing
11	Sv64	Reserved for page-based 64-bit virtual addressing
12-15	-	Reserved

ASID - Current ASID

Indicates the ASID number of the current program.

PPN - Hardware refill root PPN

PPN for the first-level hardware refill.

6.3 MMU Parity Check

MMU supports configurable parity check for TAG and DATA in jTLB. When the check mechanism is enabled, the jTLB performs parity encoding on the data during write operations and performs a check during read operations. If a 1-bit error is detected, it can report the error information, invalidate the cache lines in the jTLB that are in error, and treat the request as a jTLB miss. It then initiates a hardware Page Table Walk and performs refill. Software can query the MCER/SCER registers to retrieve relevant error information, such as whether a jTLB check error was generated and the location of the error. For detailed information of control registers, please refer to the description of MCER/SCER register in *Debug/Trace Register Group (Shared with Debug Mode)*.

Errors of more than 1 bit cannot be detected or corrected.

C920 MMU supports software-injected error functionality. For detailed information on control registers, please refer to the description of MEICR register in *Debug/Trace Register Group (Shared with Debug Mode)*.

6.4 PMP

6.4.1 PMP Overview

C920 PMP complies with the RISC-V standard. PMP unit is designed to check the access permission of a physical address, to determine whether the CPU has the read/write/execution permissions of the address in current mode.

The PMP unit of C920 provides the following main features:

- Supports 8/16/32/64 PMP entries, identified and indexed by by a number ranging from 0 to 63.
- Supports the minimum address split granularity of 4 KB.
- Supports the OFF, Top of Range (TOR), and naturally Aligned Power-of-2 Regions (NAPOT) address matching modes, but not the Naturally Aligned Four-byte region (NA4) mode.
- Supports configuration of three permissions: readable, writable, and executable.
- PMP entries support software Lock.
- Support additional configuration of EPMP functionality.

6.4.2 PMP Control Registers

A PMP entry mainly consists of an 8-bit configuration register and a 64-bit address register. All PMP control registers can only be accessed in Machine Mode (M-mode), but accesses in other modes will trigger illegal instruction exceptions.

6.4.2.1 PMPCFG Register

Physical Memory Protection Configuration (pmpcfg) register supports permission configuration of 8 entries.

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	
entry7_cfg	entry6_cfg	entry5_cfg	entry4_cfg	entry3_cfg	entry2_cfg	entry1_cfg	entry0_cfg	pmpcfg0
8	8	8	8	8	8	8	8	

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	
entry15_cfg	entry14_cfg	entry13_cfg	entry12_cfg	entry11_cfg	entry10_cfg	entry9_cfg	entry8_cfg	pmpcfg2
8	8	8	8	8	8	8	8	

Fig. 6.4: **Overall Distribution of PMPCFG Registers**

7	6 5	4 3	2	1	0
L(WARL)	0(WARL)	A(WARL)	X(WARL)	W(WARL)	R(WARL)
1	2	2	1	1	1

Fig. 6.5: **PMP Configuration Register**

Detailed information of PMP control register is described in Table 6.7.

Table 6.7: **Description of PMP Control Register**

Bit	Name	Description
0	R	The readable attribute of the entry: **0**: The address matching the entry is non-readable. **1**: The address matching the entry is readable.
1	W	The writable attribute of the entry: **0**: The address matching the entry is non-writable. **1**: The address matching the entry is writable.
2	X	The executable attribute of the entry: **0**: The address matching the entry is non-executable. **1**: The address matching the entry is executable.
4:3	A	The address matching mode of the entry. **00**: The OFF mode, in which the entry is invalid. **01**: The TOR mode, in which the address of the adjacent entry is used as the matching range. **10**: The NA4 mode, in which the matching range is 4 bytes. This mode is not supported. **11**: The NAPOT mode, in which the matching range is a power of 2 and is at least 4 KB.

Continued on next page

Table 6.7 – continued from previous page

Bit	Name	Description
7	L	The lock enable bit of the entry. **0**: That access in M-mode will succeed, and access results in System Mode(S-mode)/User Mode(U-mode) depend on the R/W/X settings **1**: The entry is locked and cannot be modified. In TOR mode, the address register of the previous entry cannot be modified either. Access results in all modes depend on the R/W/X settings.

In TOR mode, assuming that the access address is A, the condition of hitting entry i is as follows: pmpaddr(i-1) <= A < pmpaddr(i). The lower boundary of entry 0 is 0.

The relationship of addresses and corresponding protection region size in NAPOT mode are shown in Table 6.8.

Table 6.8: **Protection Region Interval Encoding**

pmpaddr[37:9]	pmpcfg.A	Protection Region Size	Remarks
a_aaaa_aaaa_aaaa_aaaa_aaaa_aaaa_aaa0	NAPOT	4KB	Support
a_aaaa_aaaa_aaaa_aaaa_aaaa_aaaa_aa01	NAPOT	8KB	Support
a_aaaa_aaaa_aaaa_aaaa_aaaa_aaaa_a011	NAPOT	16KB	Support
a_aaaa_aaaa_aaaa_aaaa_aaaa_aaaa_0111	NAPOT	32KB	Support
a_aaaa_aaaa_aaaa_aaaa_aaaa_aaa0_1111	NAPOT	64KB	Support
a_aaaa_aaaa_aaaa_aaaa_aaaa_aa01_1111	NAPOT	128KB	Support
a_aaaa_aaaa_aaaa_aaaa_aaaa_a011_1111	NAPOT	256KB	Support
a_aaaa_aaaa_aaaa_aaaa_aaaa_0111_1111	NAPOT	512KB	Support
a_aaaa_aaaa_aaaa_aaaa_aaa0_1111_1111	NAPOT	1M	Support
a_aaaa_aaaa_aaaa_aaaa_aa01_1111_1111	NAPOT	2M	Support
a_aaaa_aaaa_aaaa_aaaa_a011_1111_1111	NAPOT	4M	Support
a_aaaa_aaaa_aaaa_aaaa_0111_1111_1111	NAPOT	8M	Support
a_aaaa_aaaa_aaaa_aaa0_1111_1111_1111	NAPOT	16M	Support
a_aaaa_aaaa_aaaa_aa01_1111_1111_1111	NAPOT	32M	Support
a_aaaa_aaaa_aaaa_a011_1111_1111_1111	NAPOT	64M	Support
a_aaaa_aaaa_aaaa_0111_1111_1111_1111	NAPOT	128M	Support
a_aaaa_aaaa_aaa0_1111_1111_1111_1111	NAPOT	256M	Support
a_aaaa_aaaa_aa01_1111_1111_1111_1111	NAPOT	512M	Support
a_aaaa_aaaa_a011_1111_1111_1111_1111	NAPOT	1G	Support
a_aaaa_aaaa_0111_1111_1111_1111_1111	NAPOT	2G	Support
a_aaaa_aaa0_1111_1111_1111_1111_1111	NAPOT	4G	Support
a_aaaa_aa01_1111_1111_1111_1111_1111	NAPOT	8G	Support
a_aaaa_a011_1111_1111_1111_1111_1111	NAPOT	16G	Support
a_aaaa_0111_1111_1111_1111_1111_1111	NAPOT	32G	Support
a_aaa0_1111_1111_1111_1111_1111_1111	NAPOT	64G	Support
a_aa01_1111_1111_1111_1111_1111_1111	NAPOT	128G	Support
a_a011_1111_1111_1111_1111_1111_1111	NAPOT	256G	Support
a_0111_1111_1111_1111_1111_1111_1111	NAPOT	512G	Support

Continued on next page

Table 6.8 – continued from previous page

pmpaddr[37:9]	pmpcfg.A	Protection Region Size	Remarks
0_1111_1111_1111_1111_1111_1111_1111	NAPOT	1T	Support
1_1111_1111_1111_1111_1111_1111_1111	Reserved	-	-

Note:

PMP NAPOT mode in C920 supports the minimum granularity of 4KB and does not support NA4 mode.

6.4.2.2 PMPADDR Register

The PMP unit implements a total of 8/16/32/64 address registers pmpaddr0 ~ pmpaddr7/15/31/63 to store the physical addresses of the entries.

RISC-V specifies that Physical Memory Protection Address (pmpaddr) Register stores the [39:2] bits of the physical address. Since the minimum granularity of the C920 PMP entry is 4KB, bits [8:0] will not be applied in address authentication logic.

	63 38	37 9	8 0
	0	address[39:11] (WARL)	0 (WARL)
Reset	0	0	0

Fig. 6.6: **PMPADDR Register**

6.5 Memory Access Order

In different scenarios, the access process of C920 to the address space can be summarized as follows:

Scenario 1: without Virtual Address (VA) - Physical Address (PA) translation

1. CPU access PA:
2. Obtain the address attribute from the sysmap.h file.
3. Perform PMP checks to determine whether the XWR permissions conform to the PMP settings.
4. Access the address.

Scenario 2: with VA-PA translation

1. CPU access VA:
2. Translate the address by MMU to obtain the corresponding PTE.
3. Obtain information from the PTE: PA, address attribute, and XWR permissions.
4. Perform PMP checks to determine whether the XWR permissions conform to the PMP settings. (The final XWR permissions are determined by taking the ‘minimum value’ of PMP and PTE)
5. Access the address.

CHAPTER 7

Memory Subsystem

7.1 Memory Subsystem Overview

Each core of C920 has its own Instruction Cache (I-Cache) and Data Cache (D-Cache). Four cores share one L2 cache. Data coherence among multiple cores is maintained by hardware.

7.2 L1 I-Cache

7.2.1 Overview

The L1 I-Cache is specialized in the following key features:

- Instruction cache size is hardware configurable, supporting 32KB/64KB.
- 2-way set-associative, with a cache line size of 64 bytes;
- Virtually indexed, physically tagged (VIPT);
- Data width for access: 128 bits;
- Supports First-in, first-out (FIFO) replacement strategy;
- Supports invalidation for all I-Cache and a single cache line;
- Supports instruction prefetch;
- Supports branch prediction;
- Supports parity check;
- D-Cache snooping after a request misses the I-Cache (this feature can be enabled and disabled).

7.2.2 Branch Prediction

C920 I-Cache adopts the 2-way set-associative structure. C920 implements I-Cache branch prediction to reduce power consumption in parallel access to two caches. When branch prediction information is valid, access to invalid data paths is disabled, and only the data from the predicted path is accessed. You can configure Implicit Operand Register MHINT.IWPE to enable I-Cache branch prediction.

Branch prediction can be classified into the following two types by different fetching behaviors:

Sequential access : When performing consecutive in-line instruction fetching, the prediction of the accessed path is based on the previous path hit information.

Jump access : In addition to obtaining the target jump address, branch instructions also fetch path prediction information of the target cache line, and access one path of the cache based on that information.

7.2.3 Loop Acceleration Buffer

C920 provides a 32-byte loop acceleration buffer to cope with a large number of short loops in programs. When detecting a short-loop instruction sequence, the CPU loads it to the loop acceleration buffer. When a subsequent instruction fetch request hits the buffer, the CPU directly obtains the instruction and target jump address from the buffer, and disables access to I-Cache, branch history table, and branch jump target predictor, so as to reduce the dynamic power consumption of instruction fetch.

You can configure Implicit Operand Register MHINT.IWPE to enable short-loop acceleration.

7.2.4 Branch History Table

C920 processor provides a branch history table to predict the branch direction of conditional branches. The branch history table has a capacity of 64Kb and takes a BI-MODE predictor as the prediction mechanism, supporting one branch result prediction per cycle.

The branch history table consists of two parts: the predictor and the selector. And the predictor is further divided into a jump predictor and a non-jump predictor, which are dynamically maintained based on the branch history information. The branch history table indexes each path based on the branch history information and the current branch instruction address, to obtain the predicted result of the branch instruction’ s jump direction.

The conditional branch instructions predicted by the branch history table include:

BEQ, BNE, BLT, BLTU, BGE, BGEU, C.BEQZ, C.BNEZ.

7.2.5 Branch Jump Target Predictor

C920 provides branch jump target predictor to predict jump target addresses of branch instructions. The branch jump target predictor records the historical target addresses of branch instructions. If the current branch instruction hits the branch jump target predictor, the recorded target address is used as the predicted target address of the current branch instruction.

The branch jump target predictor provides the following main features:

- Hardware configurable, supporting 1024 table entries and 2048 table entries.
- Supports the 2-way set-associative structure, and supports selection and replacement based on the Program Counter (PC) in the lower bits of a branch instruction.
- Maintains I-Cache branch prediction information.
- Supports indexing by using a portion of the current branch instruction' s PC.
- The branch instructions predicted by the branch target predictor include:
  BEQ, BNE, BLT, BLTU, BGE, BGEU, C.BEQZ, C.BNEZ, JAL and C.J

7.2.6 Indirect Branch Predictor

C920 adopts the indirect branch predictor to predict the target addresses of an indirect branch. Indirect branch instructions acquire target addresses from registers. One indirect branch instruction can contain multiple branch target addresses, which cannot be predicted by the conventional branch jump target predictor. Therefore, C920 applies the indirect branch prediction mechanism based on branch history, to associate the historical target addresses of indirect branch instructions with the branch history information prior to that branch. And C920 discretizes different target addresses of the same indirect branch based on different branch history information, so as to enable predictions for multiple different target addresses.

Indirect branch instructions include:

- JALR: Excluding source registers x1 or x5
- C.JALR: Excluding source registers x5
- C.JR: Excluding source registers x1 or x5

7.2.7 Return Address Predictor

Return address predictor is designed to predict a quick and accurate return address when a function call ends. When the Instruction Fetch Unit (IFU) obtains a valid function return instruction through decoding, it pulls a function return address from the return address predictor stack. The return address predictor supports a maximum of 12 levels of nested function calls, exceeding which will occur incorrect target address prediction.

- The function call instructions include JAL, JALR, and C.JALR.
- The function return instructions include JALR, C.JR, and C.JALR.

The specific division of instruction functionality is illustrated in Table 7.1.

Table 7.1: **Specific Division of Instruction Functionality**

rd	rs1	rs1=rd RAS action
!link	!link -	none
!link	link -	pop
link	!link -	push
link	link 0	push and pop
link	link 1	push

7.2.8 Fast Jump Target Predictor

To speed up the fetch efficiency of IFU in consecutive jumps, C920 adds a fast jump target predictor at level 1 of the IFU. When consecutive jumps occur, the fast jump target predictor records the address of the second jump instruction and the target address of the jump. If an instruction fetch request hits the fast jump target predictor, the IFU starts at level 1, reducing the performance loss of at least one cycle.

The branch instructions predicted by the fast jump target predictor include:

- BEQ, BNE, BLT, BLTU, BGE, BGEU, C.BEQZ, C.BNEZ
- JAL, C.J
- Function return instructions

7.3 L1 D-Cache

7.3.1 Overview

The L1 D-Cache is specialized in the following main features:

- Instruction cache size is hardware configurable, supporting 32KB/64KB.
- 2-way set-associative, with a cache line size of 64 bytes;
- Physically indexed, physically tagged (PIPT);
- The maximum data width per read access: 128 bits, supporting byte, halfword, word, doubleword, and quadword access;
- Maximum data width per write access: 256 bits, supporting accesses with any combinations of bytes;
- Write policy supports write-back with write-allocate mode and write-back with write-no-allocate mode;
- Supports First-in, first-out (FIFO) replacement strategy;
- Supports invalidation and cleaning of the entire D-Cache and individual cache line.
- Supports multi-channel data prefetch for instructions.
- Supports Error Correcting Code (ECC) and parity check.

7.3.2 L1 D-Cache Coherence

The hardware maintains data coherence in L1 D-Cache across different cores for requests with page attributes configured as shareable and cacheable.

While the CPU does not maintain data coherence in L1 D-Caches for requests with page attributes configured as non-shareable and cacheable. If non-shareable and cacheable pages need to be shared across cores, software is required to maintain data coherence.

C920MP L1 cache maintains D-Cache coherence across mlutiple cores, based on the MESI protocol. MESI represents the four states of each cache line in D-Cache, which are:

- M: The cache line is available only in this D-Cache and is dirty (UniqueDirty).
- E: The cache line is available only in this D-Cache and is clean (UniqueClean).
- S: The cache line may be available in multiple D-Caches and is clean (ShareClean).
- I: The cache line is not available in this D-Cache (Invalid).

7.3.3 Exclusive Access

C920 supports exclusive memory access instructions: Load-Reserved (LR) and Store-Conditional (SC). You can use the two instructions to constitute a synchronization primitive such as an atomic lock, to synchronize data among different processes of a core or among different cores. The LR instruction marks the address to be exclusively accessed, and the SC instruction determines whether the tagged address is preempted by other processes. C920 provides a local monitor in the L1 D-Cache and a global monitor in the L2 cache for each core. Each monitor consists of a state machine and an address buffer. And the state machine has two states: IDLE and EXCLUSIVE.

Exclusive access to a cacheable page can be implemented with the local monitor. When the LR instruction is executed, it sets the state machine of the local monitor to EXCLUSIVE state and stores the address to be accessed and the size to the buffer; When the SC instruction is executed, it reads the state, address, and size of the local monitor. If the state is EXCLUSIVE and the address exactly matches the size, the write operation is performed, returning a successful write and resetting the state machine to IDLE state. If the state or address/size does not meet the conditions, or if the D-Cache is not enabled, the write operation is not executed, returning a write failure and resetting the state machine to IDLE state. When other cores' write operations match the local detector at the same cache line address, the state machine is also reset to IDLE state. Local detector is not affected by write operations within the same core or exclusive accesses with different addresses. Additionally, the local detector needs to be cleared during process switching.

Exclusive access to a non-cacheable page is implemented with both the local monitor and the global monitor. When the LR instruction is executed, it must set both the local monitor and the global monitor. After passing the local detector check, the SC instruction needs to further check the global detector. Only when the global detector also passes the check, the write operation is executed, returning a successful write and clearing the state machine. Otherwise, the write operation is not performed, returning a write failure and resetting the state machine. When other cores' write operations match a specific global detector address, the state of that global detector is reset to IDLE state.

It is recommended to apply LR and SC instructions to implement atomic locks in C920 system. If the address attribute of an atomic lock is cacheable (either shareable or non-shareable), no special design is required for the SoC system, which is a typical case. While if the address attribute of an atomic lock is non-cacheable, device, or strongly ordered, you need to integrate exclusive monitor functionality within the system (e.g. Slave side). Using any other method, the result of the operation would be UNPREDICTABLE.

7.4 L2 Cache

7.4.1 L2 Cache Overview

L2 cache is specialized in the following key features:

- Cache size is hardware configurable, supporting 256KB/512KB/1MB/2MB/4MB/8MB;

- 16-way set-associative, with a cache line size of 64 bytes;
- Strictly inclusive relationship of the L1 D-Cache and L2 Cache. And non-strictly inclusive relationship of the L1 I-Cache and L2 Cache;
- Physically indexed, physically tagged (PIPT);
- The maximum data width per access is 64 bytes;
- Write policies support write-back with write-allocate, and write-back with write-no-allocate;
- Supports First-in, first-out (FIFO) replacement strategy;
- Supports programmable RAM latency;
- Supports optical ECC check mechanism;
- Supports instruction prefetch and Translation Lookaside Buffer (TLB) prefetch;
- Adopt segmented pipeline technology.

7.4.2 L2 D-Cache Coherence

C920MP L2 cache adopts MESI protocol to maintain D-Cache coherence across multiple processor cores. MESI represents the four states of each cache line in D-Cache, which are:

- M: The cache line is available only in this D-Cache and is dirty (UniqueDirty).
- E: The cache line is available only in this D-Cache and is clean (UniqueClean).
- S: The cache line may be available in multiple D-Caches and is clean (ShareClean).
- I: The cache line is not available in this D-Cache (Invalid).

7.4.3 Structure

The L2 cache of C920MP builts on a block-based pipelining architecture, where access addresses are dispersed across two different segments. This allows for parallel processing of multiple accesses, thereby improving access efficiency.

The block mechanism is shown in Fig. 7.1.

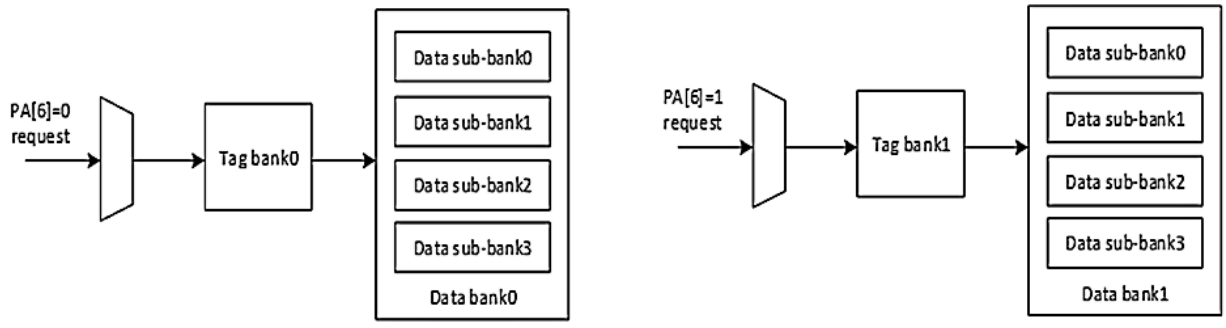


Fig. 7.1: **L2 Cache Structure**

- TAG RAM is divided into two tag sub-blocks by Physical Address (PA) [6]: Tag bank 0 and Tag bank 1, to handle two access requests in parallel within the same clock cycle.
- Similarly, DATA RAM is divided into two data sub-blocks by PA[6]: Data bank 0 and Data bank 1. Each data sub-block is further divided into four 128-bit micro blocks, so as to achieve parallel retrieval of a cache line.

7.4.4 RAM Latency

The access latency of L2 Cache is longe because of its large cache size, typically requiring multiple clock cycles to complete the access. C920MP provides configurable access latency and can be manually set, based on setup time and latency of RAM for different processes. The configuration details are illustrated in Table 7.2.

Table 7.2: **Configuration of RAM Latency**

Configuration Options	Feature	Description
L2 TAG setup	L2 Cache Tag RAM setup: **1b0** 0 cycle. **1b1** 1 cycle.	L2 Cache Tag RAM configuration only affects TAG RAM access.
L2 TAG latency	L2 Cache Tag RAM latency: **3b000**: 1 cycle. **3b001**: 2 cycles. **3b010**: 3 cycles. **3b011**: 4 cycles. **3b1xx**: 5 cycles.	
L2 DATA setup	L2 Cache Data RAM setup: **1b0** 0 cycle. **1b1** 1 cycle.	L2 Cache Data RAM configuration only affects DATA RAM access.
L2 DATA latency	L2 Data RAM latency: **3b000**: 1 cycle. **3b001**: 2 cycles. **3b010**: 3 cycles. **3b011**: 4 cycles. **3b100**: 5 cycles. **3b101**: 6 cycles. **3b110**: 7 cycles. **3b111**: 8 cycles.	

You should configure setup/latency parameter, based on the access time of RAM.

The default value of setup/latency is the same as the hardware configuration value. When the latency hardware configuration value is larger or equal to 2 cycles, the RAM clock adopts 2-fold frequency divided clock.

The number of access cycles obtained will be shown in Table 7.3 after configuring the above options.

Table 7.3: **Valid Access Latency of TAG RAM**

TAG latency	Valid Access Latency of TAG RAM	
	TAG setup = 0	TAG setup = 1
000	1	2
001	2	3
010	3	4
011	4	5

Continued on next page

Table 7.3 – continued from previous page

TAG latency	Valid Access Latency of TAG RAM	
	TAG setup = 0	TAG setup = 1
1xx	5	5

Table 7.4: **Valid Access Latency of DATA RAM**

DATA latency	Valid Access Latency of DATA RAM	
	DATA setup = 0	DATA setup = 1
000	1	2
001	2	3
010	3	4
011	4	5
100	5	6
101	6	7
110	7	8
111	8	9

Note:

- The maximum effective delay of L2 Tag latency is 5 cycles;
- When TAG setup is set to 1, an additional cycle is added to the access time; The SRAM input signals are flopped before accessing the SRAM;
- The maximum effective delay of L2 Data latency is 9 cycles;
- When DATA setup is set to 1, an additional cycle is added to the access time; similarly; The SRAM input signals are flopped before accessing the SRAM SRAM access.

7.5 Accelerated Memory Access

This section describes the accelerated memory access features of L1 and L2 cache in C920.

7.5.1 L1 I-Cache Instruction Prefetch

L1 I-Cache supports instruction prefetch, which can be implemented by configuring implicit register MHINT.IPLD. When an instruction access request misses the current cache line, the next consecutive cache line is prefetched and stored to the prefetch buffer. When the instruction access request hits the prefetch buffer, the instruction is directly obtained from the prefetch buffer and written back to I-Cache, so as to reduce the instruction fetch latency.

This feature requires that the prefetched cache line and the current accessed cache line be on the same page, to ensure security of the instruction fetch address. In addition, read-sensitive device address spaces can not be allocated to instruction spaces.

7.5.2 Multi-channel Data Prefetch of L1 D-Cache

C920 supports data prefetch to reduce the access latency of large-sized memory such as DDR SDRAMs. C920 detects D-Cache misses to determine a fixed access mode through matching. Then the hardware automatically prefetches cache lines and writes them back to L1 D-Cache.

C920 supports up to 8-way data prefetch and two different prefetch methods: consecutive prefetch and interval prefetch (stride <= 32 cache lines).

C920 also implements forward prefetch and backward prefetch (the stride is negative) to support various possible access modes.

Data prefetch is disabled when the CPU invalidates or clears D-Cache.

You can configure implicit register MHINT.IPLD to enable data prefetch and MHINT.DPLD_DIS to determine the number of cache lines to be prefetched at a time.

The following instructions support data prefetch:

- LB、LBU、LH、LHU、LW、LWU、LD
- FLW、FLD
- LRB、LRH、LRW、LRD、LRBU、LRHU、LRWU、LURB、LURH、LURW、LURD、LURBU、LURHU、LURWU、LBI、LHI、LWI、LDI、LBUI、LHUI、LWUI、LDD、LWD、LWUD

7.5.3 L1 Adaptive Write Allocation Mechanism

C920 L1 implements adaptive write allocation. When CPU detects consecutive memory write operations, the write allocation attribute of pages is automatically disabled.

You can configure implicit register MHINT.AMR to enable L1 adaptive write allocation.

Adaptive write allocation is automatically disabled when CPU invalidates or clears D-Cache. Then CPU redetects consecutive memory write operations after the invalidation and clearing operation.

The following instructions support adaptive write allocation:

- SB、SH、SW、SD
- FSW、FSD
- SRB、SRH、SRW、SRD、SURB、SURH、SURW、SURD、SBI、SHI、SWI、SDI、SDD、SWD

7.5.4 L2 Prefetch Mechanism

L2 cache supports instruction prefetch and TLB access prefetch. L2 cache is specialized in the following features:

- The number of software-configurable instruction prefetch quantities is 0, 1, 2, or 3. All prefetches will be refilled into the L2 cache.
- The TLB prefetch quantity is fixed at 1.
- The prefetch mechanism operates with a 4KB page boundary, and it actively stops prefetching when encountering addresses that cross this boundary.

- Prefetch mechanism can be configured by M-mode L2 Cache Enable Register (mccr2).

7.6 L1/L2 Cache Operation Instruction and Register

I-Cache and D-Cache are automatically invalidated and disabled by default after CPU reset.

Similarly, L2 cache is automatically invalidated after CPU reset. Then L2 cache is automatically enabled and cannot be disabled after the invalidation. It is worth noting that L2 still initiates a refill operation on a miss when the L1 cache is disabled.

7.6.1 Extented Register of L1 Cache

C920 extented registers of L1 cache are mainly classified by features as follows:

- Cache enable and mode configuration: Machine Mode (M-mode) Hardware Configuration Register (mhcr) enables/disables I-Cache/D-Cache and configure the write allocation and writeback modes. Supervisor-mode (S-mode) Hardware Configuration Register (shcr) is a read-only register mapped to the mhcr register.
- Dirty page table entry clearing and invalidation: M-mode Cache Operation Register (mcor) allows you to clear and invalidate entries in I-Cache and D-Cache.
- Cache read operation: M-mode Cache Access Instruction Register (mcins), M-mode Cache Access Index Register (mcindex), and M-mode Cache Access Data Register 0/1 (mcdata0/1) allow you to read data from I-Cache and D-Cache.

For detailed specification of control registers, please refer to *Debug/Trace Register Group (Shared with Debug Mode)* and *Debug Mode Register Group/Trace Register Group*.

7.6.2 Extented Register of L2 Cache

C920 extented registers of L2 cache are mainly classified by features as follows:

- L2 cache enable and latency configuration: M-mode L2 Cache Enable Register (mccr2) allows you to set the access latency of L2 cache.
- L2 cache read operation: The mcins, mcindex, and mcdata0/1 registers allow you to read data from L2 cache.

For detailed definition and specification of control registers, please refer to *Debug/Trace Register Group (Shared with Debug Mode)* 和*Debug Mode Register Group/Trace Register Group*.

7.6.3 L1/L2 Cache Operation Instruction

C920 extends L1/L2 cache operation instructions that invalidate by address, invalidate all, flush entries by address, flush all cache lines, flush and invalidate entries by address, and flush and invalidate all cache lines. For detailed information, please refer to Table 7.5.

Table 7.5: **L1/L2 Cache Operation Instruction**

Instruction	Description
ICACHE.IALL	Invalidates all entries in I-Cache.
ICACHE.IALLS	Invalidates all entries in I-Cache through broadcasting.
ICACHE.IPA	Invalidates entries in the I-Cache that match the specified physical addresses.
ICACHE.IVA	Invalidates entries in the I-Cache that match the specified virtual addresses.
DCACHE.CALL	Clears all dirty entries in D-Cache.
DCACHE.CIALL	Clears and invalidates all dirty entries in D-Cache.
DCACHE.CIPA	Clears entries in D-Cache that match the specified physical addresses and invalidates the entries.
DCACHE.CISW	Clears entries in D-Cache by set/way and invalidates the entries.
DCACHE.CIVA	Clears entries in D-Cache that match the specified virtual addresses and invalidates the entries.
DCACHE.CPA	Clears entries in D-Cache that match the specified physical addresses.
DCACHE.CPAL1	Clears entries in L1 D-Cache that match the specified physical addresses.
DCACHE.CVA	Clears entries in D-Cache that match the specified virtual addresses.
DCACHE.CSW	Clears entries in D-Cache by set/way.
DCACHE.CVAL1	Clears entries in the L1 D-Cache that match the specified virtual addresses.
DCACHE.IPA	Invalidates entries in D-Cache that match the specified physical addresses.
DCACHE.ISW	Invalidates entries in D-Cache by set/way.
DCACHE.IVA	Invalidates entries in D-Cache that match the specified virtual addresses.
DCACHE.IALL	Invalidates all entries in D-Cache.

For detailed instruction information, please refer to *Appendix B-1 Cache Instructions*.

7.7 L1/L2 Cache Protection Mechanism

C920 implements cache protection mechanism, which includes: L1 I-Cache Parity Check、jTLB Parity Check、L1 D-Cache ECC check and L2 Cache ECC check. Various mechanisms for detection/correction capability and interrupt reporting are illustrated in Table 7.6.

Table 7.6: **ECC/Parity Check Detect/Correct Capability and Interrupt Report**

Cache Type	1 Bit Error	2 Bit Error	Errors of 2 Bits or More
L1 I-Cache	Detectable Without issuing exceptions or interrupts	Undetectable Without issuing exceptions or interrupts	Undetectable Without issuing exceptions or interrupts
L1 D-Cache	Detectable and correctable Without issuing exceptions or interrupts	Detectable Issuing interrupt request	Undetectable Without issuing exceptions or interrupts

Continued on next page

Table 7.6 – continued from previous page

jTLB	Detectable Without issuing exceptions or interrupts	Undetectable Without issuing exceptions or interrupts	Undetectable Without issuing exceptions or interrupts
L2 Cache	Detectable and correctable Without issuing exceptions or interrupts	Detectable Issuing interrupt request	Undetectable Without issuing exceptions or interrupts

7.7.1 L1 I-Cache Parity Check

L1 I-Cache supports configurable parity check mechanism, which checks the tag array of the I-Cache with the granularity of 28 bits and the data array with the granularity of 32 bits.

L1 CACHE parity check/ECC feature is enabled by bit 19 in MHINT register. When the feature is enabled, the I-Cache performs parity encoding during data writes and checks for errors during data reads. It detects and invalidates the error data in case of a 1-bit data error, reinitiates a fetch request from the bus, and refills the cache again. Additionally, it records error information, including the way and index information, which can be queried in MCER/SCER registers. The errors can only be cleared in M-Mode by writing to the M-Mode L1 Cache ECC Register (MCER). For detailed control register specifications, please refer to information about MCER/SCER registers in *Debug/Trace Register Group (Shared with Debug Mode)*. Errors of more than 1 bit can not be detected or corrected.

C920 L1 Instruction Cache also supports software-injected errors. For specific control register details, refer to MEICR register *Debug/Trace Register Group (Shared with Debug Mode)*.

Furthermore, parity check for jTLB has been implemented to detect 1-bit errors.

7.7.2 L1 D-Cache ECC Check

L1 D-Cache supports configurable ECC check mechanism, for detailed information, please refer to Table 7.7.

Table 7.7: **L1 D-Cache Check**

RAM	Check Granularity	Check Bit	Check Method
TAG	29	7	ECC
DATA	32	7	ECC

L1 CACHE parity check/ECC feature is enabled by bit 19 of the MHINT register. When the feature is enabled, the L1 D-Cache performs ECC encoding during write operations and performs verification during read operations. When a 1-bit ECC error is detected, it can automatically correct the error and return the correct data. When a 2-bit error occurs, it can detect the error, initiate a verification error interrupt, and invalidate the cache line in L1 D-cache where the error occurred. Errors of 2 bits or more cannot be accurately detected or corrected.

Software can query the MCER/SCER registers to obtain relevant information about the errors, such as whether a 2-bit error occurred and the location of the error in D-Cache. The clearing of errors can only be done in M-Mode by writing to the MCER register. For detailed control register specification, please refer to information about MCER/SCER registers in *Debug/Trace Register Group (Shared with Debug Mode)*

The interrupt caused by a 2-bit error in L1 D-Cache is triggered by directly enabling the MCIP bit, with the in-core interrupt vector number 16. For detailed control register specification, please refer to MIP information in *Debug/Trace Register Group (Shared with Debug Mode)*.

7.7.3 L2 ECC Check

L2 Cache supports configurable ECC check and Tag RAM and Data RAM check. The corresponding check granularity is illustrated in Table 7.8.

Table 7.8: **L2 ECC Check Granularity**

RAM	**Check Granularity**	**Check Bit**
TAG	23/24/25/26/27	7(ECC)
Dirty	8bit (except fifo)	5(ECC)
DATA	64	8(ECC)

L2 CACHE ECC is enabled by setting bit 1 in the Machine Mode L2 Cache Control Register (MCCR2). When the feature is enabled, the L2 cache performs ECC encoding on data during write operations and performs ECC checking during read operations. When a 1 bit ECC error is detected, the L2 cache can correct the error and return the correct data. When a 2-bit error occurs, the L2 cache can detect the error, generate an ECC interrupt to report the issue, return the error data, and invalidate the cache line where the error occurred. Errors with more than 2 bits cannot be accurately detected or corrected.

Software can query the MCER2/SCER2 registers to obtain information about the errors, such as whether a 2-bit error occurred and the location of the error within the L2 cache. The errors can only be cleared in M-Mode by writing to the Machine Mode L2 Cache ECC Register (MCER2).For detailed control register specification, please refer to infomration about MCER2/SCER2 registers in *Debug/Trace Register Group (Shared with Debug Mode)*.

The L2 ECC interrupt serves as an interrupt source input to the PLIC (Platform-Level Interrupt Controller), where it is assigned a fixed Interrupt ID of 1 (internal to the PLIC). When a CPU responds to an external interrupt, it can query the Interrupt Claim/Completion Register (PLIC_CLAIM) to retrieve this particular ID. The configuration, maintenance, and triggering process for such interrupts can be referred to *Interrupt Controller*.

C920 L2 memory subsystem supports software-injected error functionality, and for detailed control register specification, please refer to information about MEICR2 in *Debug/Trace Register Group (Shared with Debug Mode)*.

CHAPTER 8

Vector Computation

8.1 supporting Version

C920 is compatible with *RISC-V "V" Vector Extension, Version 1.0*

8.2 Vector Programming Model

C920 supports the following vector extension features:

- 32 independent vector registers from v0 to v31. The width of the vector register is 128 bits, depending on the vector capability option.
- Vector floating-point instructions support the BF16、FP16、FP32 and FP64 elements (SEW=16/32/64).
- Vector integer instructions support the INT8、INT16、INT32 and INT64 elements (SEW=8/16/32/64).
- Vector register groups are supported to improve the efficiency of vector computation. Four types of vector register groups are supported: 32, 16, 8, or 4 vector groups, each of which contains 1, 2, 4, or 8 vector registers respectively.

Notes:

The SO attribute value of the destination address of the vector memory access instruction can not be 1.

8.3 Vector Control Register

Seven non-privileged control and status registers (CSRs) are added in C920:

- vstart

The vstart register specifies the position of the first element when a vector instruction is executed. Vstart is reset to zero after a vector instruction is executed. In most cases, software does not need to modify vstart. In C920, only vector load/store instructions support non-zero vstart registers. While, all computational vector instructions require vstart=0, otherwise an illegal instruction exception will be generated.

- vxsat

The vxsat register is valid only when the bit is set to 0, which indicates whether the result of a fixed-point instruction is overflow.

- vxrm

The vxrm register provides four rounding modes: Round up, round to even, round towards zero, and round to odd.

- vcsr

Vector control core status register.

- vl

The VL register specifies the range of elements in the destination register that will be updated by the vector instruction. Specifically, the vector instruction updates elements whose numbers are smaller than vl in the target register and clears the elements whose element numbers are greater than or equal to VL. In special cases, if vstart is greater than or equal to vl, then all elements in the destination register will not be updated.

- vtype

The vtype register defines basic data attributes for vector computation, including: invalid flag, element width setting, and vector register grouping setting.

- vlenbvtyp

The vlenbvtyp register represents vector width in bytes in C920.

In addition, C920 supports vector status maintenance and defines VS bit in mstatus[10:9], to determine if it is necessary to save vector-related registers during context switching.

8.4 Vector-related Exception

Vector instructions are classified into the following 3 categories:

- Vector load;
- Vector computation;
- Vector store.

Vector computation does not trigger exceptions. Vector store does not trigger exceptions either, because the bus ignores BRESP error message. Therefore, only vector load may trigger exceptions. When an exception is triggered by vector load, CPU discards the data that has already been read, resets vstart to 0, and sets the Machine Exception Program Counter (mepc) to point to this instruction. (Exception: mepc may point to the subsequent instruction for imprecise exceptions.)

The CPU handles vector instruction interrupts in the same way as regular instructions. The CPU completes the current instruction and the mepc points to the next instruction, and the remaining steps are the same as those in handling regular interrupts.

CHAPTER 9

Security Design

9.1 Security Requirement

This chapter provides software and hardware security design to meet the system security requirements in Trusted Execution Environment (TEE). And the requirements mainly include the following aspects:

- Support independent executable Zone.
- Support Zone isolation by code execution, memory access, peripheral devices, or I/O resources.
- Support isolation between applications and isolation between applications and the kernel within a Zone.
- Support the multi-core SMP architecture.
- Support shared memory access among Zones.
- Support RISC-V 32-bit and 64-bit architectures.
- Support trustworthy communication among Zones.
- Support TEEs that comply with the GP specification.

9.2 Processor Security Model

The RISC-V ISA architecture supports the following 3 privileged modes: Machine Mode (M-mode), Supervisor Mode (S-mode), and User Mode (U-mode), which are distinguished by execution and access permissions:

- In U-mode, only non-privileged instructions can be executed. Generally, user applications are run in this mode.
- S-mode supports the execution of additional instructions with superuser privileges and Memory Management Unit (MMU) management permissions. In most cases, complex operating systems such as Linux are run in this mode.

- The M-mode provides the most execution and access privilege, including interrupt/exception handling and management, Physical Memory Protection (PMP), privileged access control and other management privileges.

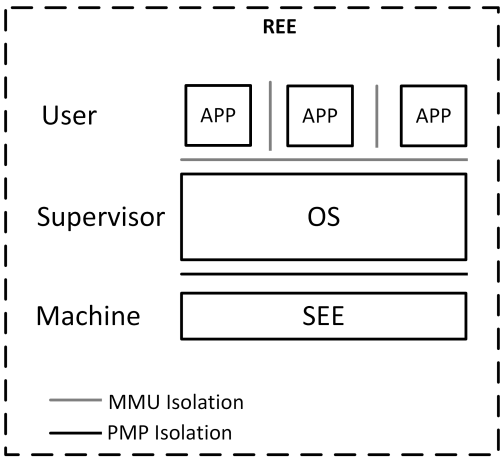


Fig. 9.1: **RISC-V Privilege Mode**

The S-mode and U-mode of RISC-V have no much difference from other mainstream processor architectures, such as S-mode and U-mode of ARM. In U-mode, only non-privileged instructions can be executed. Applications running in U-mode can only access system resources under the management of the operating system by triggering a system call trap to enter S-mode. S-mode not only supports non-privileged instructions, but alao privileged instructions and the permissions to access Control and Status Register (CSR) in S-mode. In addition, S-mode provides permissions to access MMU. Memory protection and isolation in user mode and kernel mode are implemented through virtual memory management. The M-mode provides the most execution and access privilege. The RISC-V architecture adds privileged instructions that can be executed only in M-mode and system registers can be accessed only in M-mode, such as PMP and so on. The most significant feature of M-mode is exception interception and handling. During exception handling, the processor traps all exceptions to M-mode by default. The M-mode exception handler then "forwards" interrupts to S-mode. The M-mode typically runs Trusted Firmware (TF) to adjust, allocate, and manage software and hardware resources.

Xuantie C series processors have been extended for security on the basis of the RISC-V architecture, to to satisfy the isolation requirements of TEE. These processors can create multiple virtual zones based on software coordination. Fig. 9.2 shows the overall architecture. The operating system runs independently in its own zone and applications based on that operating system, where operating system runs in S-mode and applications run in U-mode. The processor can switch to different Zones to run as needed. When the processor switches to run in a particular Zone, it will occupy the entire physical core immediately, and the processor' s domain identifier will also be updated to that of the execution domain. The switching of Zones is performed by TF running in the highest mode (M-mode).

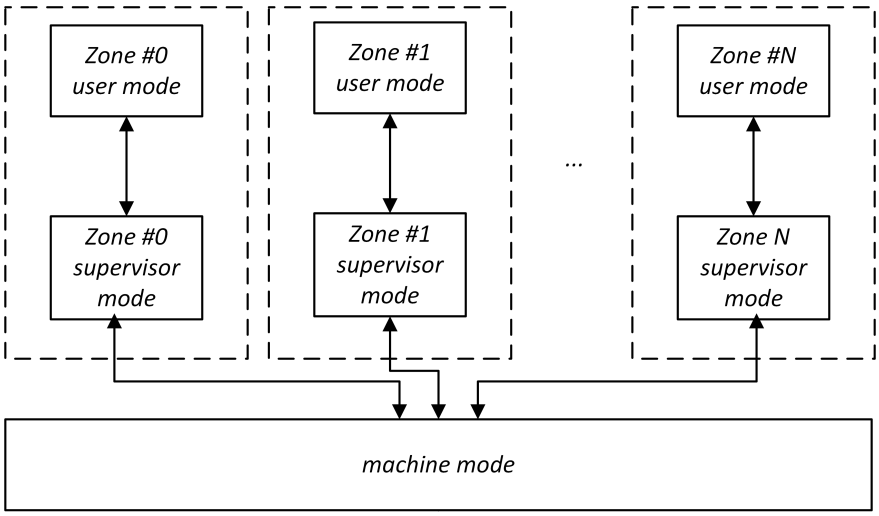


Fig. 9.2: **Zones and Privilege Modes in Xuantie RISC-V Processors**

9.3 System Security Architecture

9.3.1 Secure Memory Management

Each hardware thread can run in different zones through time-sharing. When a hardware thread runs in a particular zone, memory access needs to be isolated to the corresponding Zone, and other Zones are not allowed to access the memory resources of that Zone without authorization. In the same time, the Zone without authorization is not allowed to access the memory resources belonging to other Zones. Zones can pass data via shared memory.

PMP

The RISC-V architecture provides the Physical Memory Protection (PMP) mechanism to isolate memory access in M-mode from S-mode and U-mode. PMP can be configured only in M-mode. PMP consists of multiple groups (8 to 16 groups in general) of address registers and the related configuration registers. And these configuration registers can grant or deny read, write, and execute permissions of S-mode and U-mode. PMP can also protect memory mapping I/O (MMIO). And the TF in M-mode can configure PMP to constrain the processor' s access to I/O peripherals.

When a hardware thread switches from one zone to another zone, the PMP configuration also needs to be switched. The TF in M-mode trusted needs to save the PMP configuration in the current zone and loads the PMP configuration in the target zone to update the access permissions of memory and MMIO.

When multiple zones need to share memory, the access permissions for the memory region that needs to be accessed by multiple zones can be granted to each zone simultaneously. This means that the allowed access permissions for this memory block are written into the PMP configuration table of each zone. The PMP table will be updated by TF during zone switching. Fig. 9.3 is a typical PMP configuration diagram for multiple zones, where the SHM region represents the shared memory area allowed to be accessed by various zones.

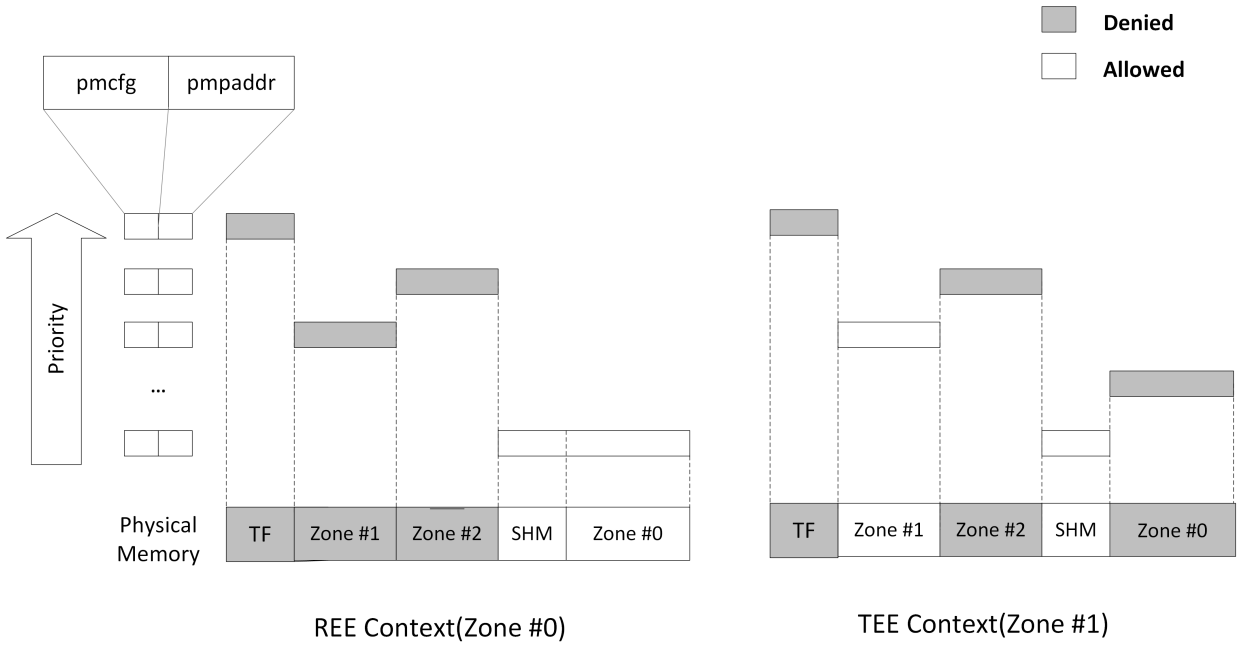


Fig. 9.3: **PMP Configuration in Different Zones**

IOPMP

The RISC-V architecture provides a PMP mechanism to protect memory and MMIO access of RISC-V processors in different privileged modes.

Other master devices connected to the bus also require memory access protection, which means I/O Physical Memory Protection (IOPMP) needs to be added. Same as PMP, IOPMP could define access permissions. It checks whether the read and write transmitted from the bus comply with the permission access rules. And only legitimate read and write can be further transmitted to the target device. Typically, two methods are used to connect to an IOPMP:

1. Connect the requester to IOPMP

An IOPMP is added between each master device and the bus, similar to the PMP in RISC-V. Each different master device needs to have its own additional IOPMP that is independent of each other. This design is relatively simple and flexible, but the IOPMPs can not be shared between master devices, which is illustrated in Fig. 9.4:

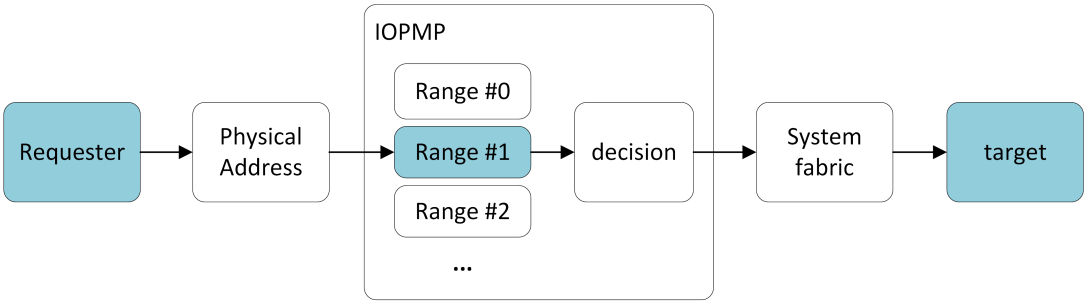


Fig. 9.4: **Connect the Requester to IOPMP**

2. Connect the destination device to IOPMP

The IOPMP of the destination device needs to distinguish requests from different master devices, which requires each access request from a master device to be accompanied by an additional Master ID. As shown in Fig. 9.5.

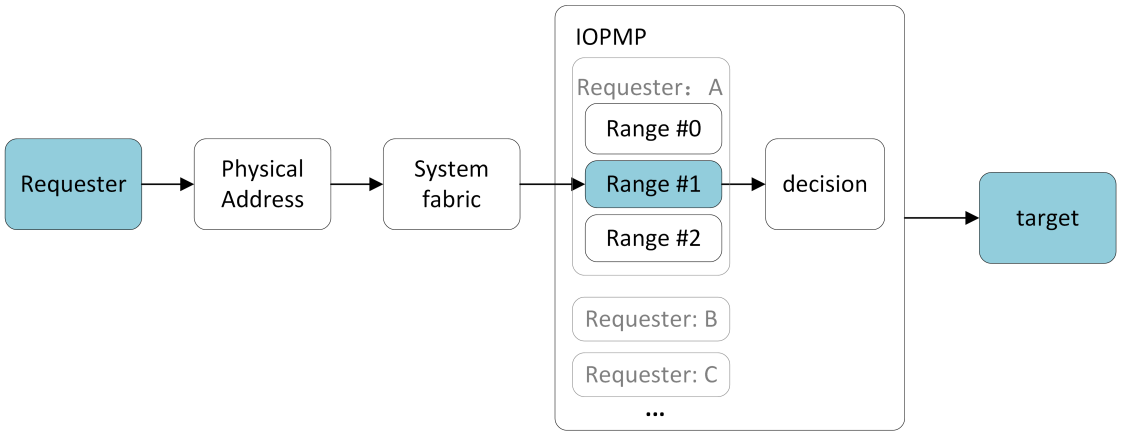


Fig. 9.5: **Connect the Destination Device to IOPMP**

Fig. 9.6 is the secure SoC system framework built by the Xuantie processor with IOPMP mounted on the requesting side.

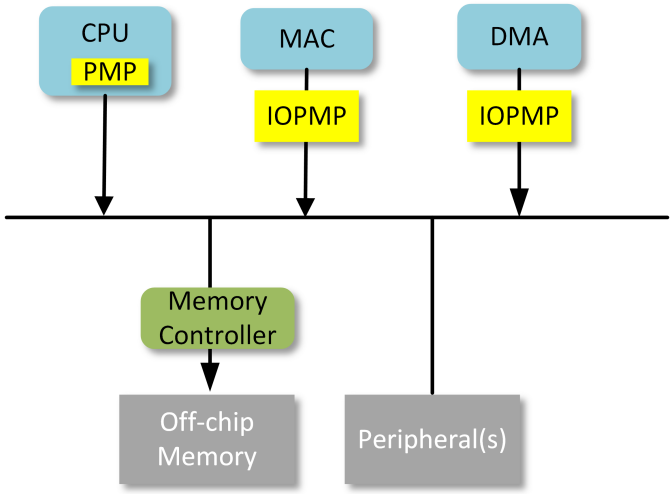


Fig. 9.6: **SoC Architecture Based on PMP and IOPMP Isolation**

MMU

Memory Management Unit (MMU) is designed to manage virtual memory in traditional operating systems. MMU implements separation of the user space and kernel space. Xuantie processor MMU integrates configurable Translation Lookaside Buffer (TLB) caches, and each TLB contains the translation mappings from virtual addresses to physical addresses and the corresponding access permissions.

Cache

Because each zone has its own independent PMP configuration when the processor is running in different zones, PMP limits the access permissions and ranges of physical memory and MMIO for each zone. In this way, PMP ensures that memory and I/O access do not interfere with or affect each other between zones.

In a Xuantie C series RISC-V processor, memory access that hits the cache is also protected by PMP, which means that any access to the cache is first checked by PMP, and further access to the cache is allowed only when the PMP check passes. And multi-core cache coherence is also protected by PMP.

DCP

Xuantie C920 provides Device Coherence Port (DCP), an AXI slave interface, through which external master devices can access the cash coherence data inside processers, so as to improve the data transmitting efficiency between processers and external masters. C920 does not provide protection to the DCP for access from external master devices, which requires the external master devices connected to the DCP be mounted to IOPMPs for aceess protection.

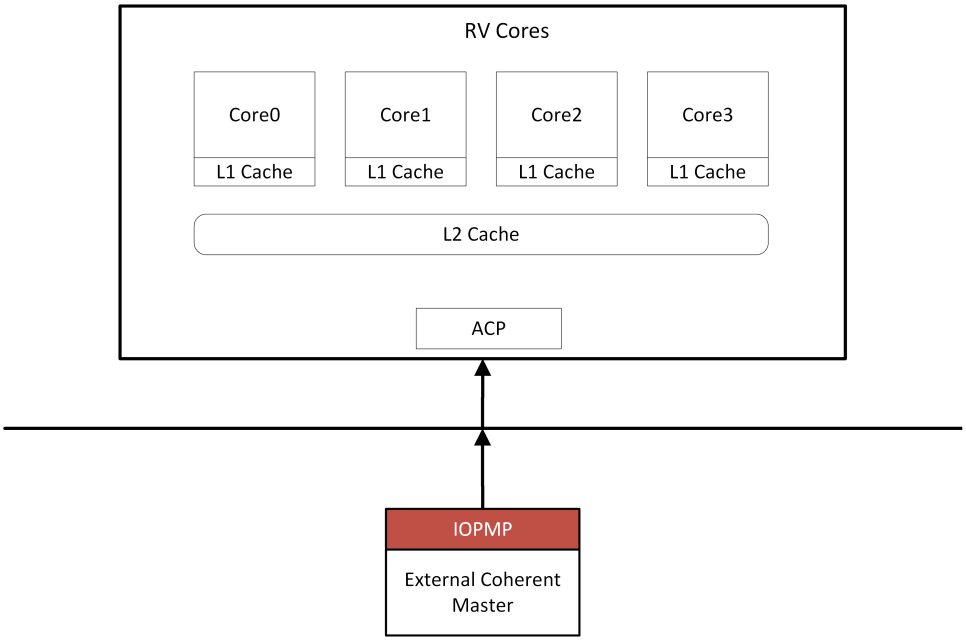


Fig. 9.7: **DCP Protection**

9.3.2 Secure Interrupts

There are two modes of interrupt sources in the Platform Level Interrupt Controller (PLIC) specification of RISC-V: M-mode interrupt sources and S-mode interrupt sources. M-mode interrupt sources are handled only in M-mode. Whiile S-mode interrupt sources can be handled in M-mode or S-mode. The M-mode has permissions to determine whether to send interrupts to S-mode for handling. The M-mode of the RISC-V architecture provides interrupt interception to help isolate interrupts of different zones. Table 9.1 describes interrupts of different modes handled.

Table 9.1: **Interrupt Response Model in RISC-V**

Target Mode of Interrupt source	Current Mode of Processors	Delegation	Whether to Respond to Interrupts	Mode to Handle Interrupt
M-mode	M-mode	Invalid	Yes	M-mode
	S-mode	Invalid	Yes	M-mode
	U-mode	Invalid	Yes	M-mode
S-mode	M-mode	0	Yes	M-mode
		1	No	-
	S-mode	0	Yes	M-mode
		1	Yes	S-mode
	U-mode	0	Yes	M-mode
		1	Yes	S-mode

Interrupts are handled in the following ways based on the interrupt interception feature of M-mode in RISC-V:

1. M-mode interrupt distribution
2. Interrupt groups

M-mode Interrupt Distribution

The M-mode supports external interrupt interception as shown in Fig. 9.8. All external interrupts are first trapped into M-mode, and TF running in M-mode will manage all external interrupts, identify the interrupt source, and forward interrupts to different zones to handle these interrupts. This interrupt handling method can satisfy the interrupt isolation requirements between different zones, but since interrupts need to be forwarded by TF. TF needs to switch the zone context during forwarding, which introduces a certain interrupt latency.

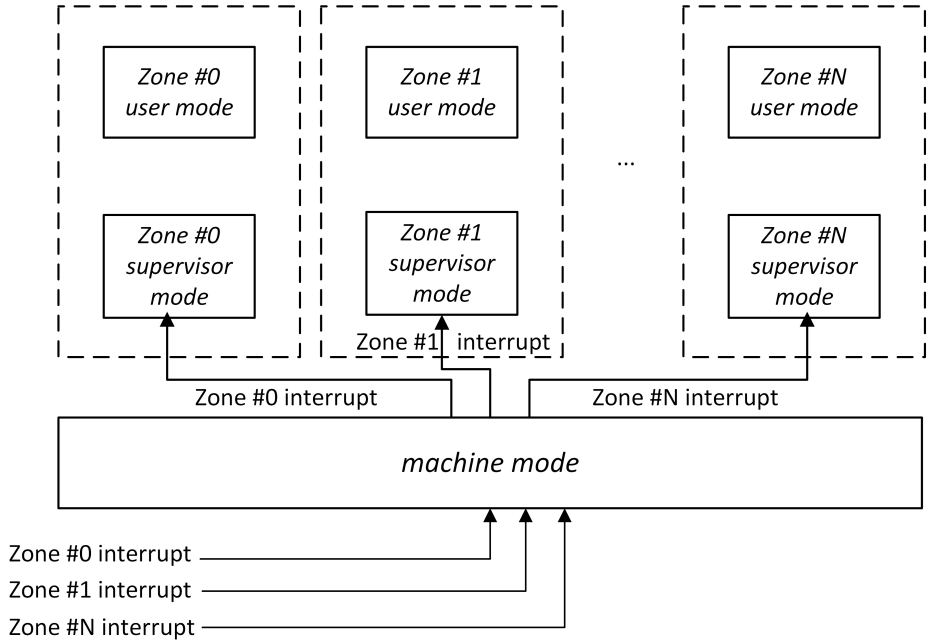


Fig. 9.8: **M-mode Interrupt Distribution in Xuantie RISC-V Processors**

In this mode, all external interrupts are sent to M-mode TF. TF first saves all the current contexts of the current zone, then reads the external interrupt number. Then, it selects the destination zone based on the pre-saved zone interrupt allocation table, and retrieves the entry by reading the stvec register. Before jumping to the interrupt entry point, TF needs to switch the PMP configuration to the destination zone, check the validity of the interrupt handler function address, and finally executes the mret instruction to switch to the destination zone. After the completion of interrupt handling, the interrupt handler function needs to return to M-mode through the ecall. The TF in M-mode will restore the original execution fileds of the interrupted zone and continue running in that zone.

Interrupt Groups

Transferring all the interrupt handling through M-mode will occur severe interrupt latency. Moreover, after the execution of the interrupt handling program, it still needs to return to M-mode through an ecall, which can cause incompatibility issues for the existing interrupt handling program (especially for Linux).

PLIC supports separate control over each interrupt source and target, which means the destination hardware thread for each interrupt source and the mode that hardware thread operates on can be configured independently. Currently, the execution environments of processors are classified into Rich Execution Environment (REE) and TEE in general. Regular interrupts are handled in REE, and secure interrupts are handled in TEEs. Most hardware interrupts are regular interrupts. Only a few number of hardware interrupts, for example secure timers, are secure interrupts. Interrupt groups are implemented to reduce the interrupt latency caused by the unified handling of M-mode interrupts.

Interrupts from interrupt sources in the current zone are handled in the zone. While the interrupts from other zones are handled in M-mode, to reduce the latency. Interrupt context scenarios are as follow:

- REE generates regular interrupts.
- REE generates secure interrupts.
- TEE generates regular interrupts.
- TEE generates secure interrupts.

REE generates regular interrupts; REE generates secure interrupts

TF needs to perform the following operations when the processor runs in the REE (Zone #0), as shown in: Fig. 9.9 .

1. Enable S-mode for the interrupt source of regular interrupts.
2. Enable M-mode for the interrupt source of secure interrupts.
3. Reset the first bit (SSIE_DELEG), fifth bit (STIE_DELEG), and ninth bit (SEIE_DELEG) of the mideleg register (Assume that software interrupts and clock interrupts are both configured as regular interrupts).
4. Enable mstatus.MIE and mstatus.SIE, and enable mie.MEIE, mie.MSIE, mie.MTIE, mie.SEIE, mie.SSIE, mie.STIE.

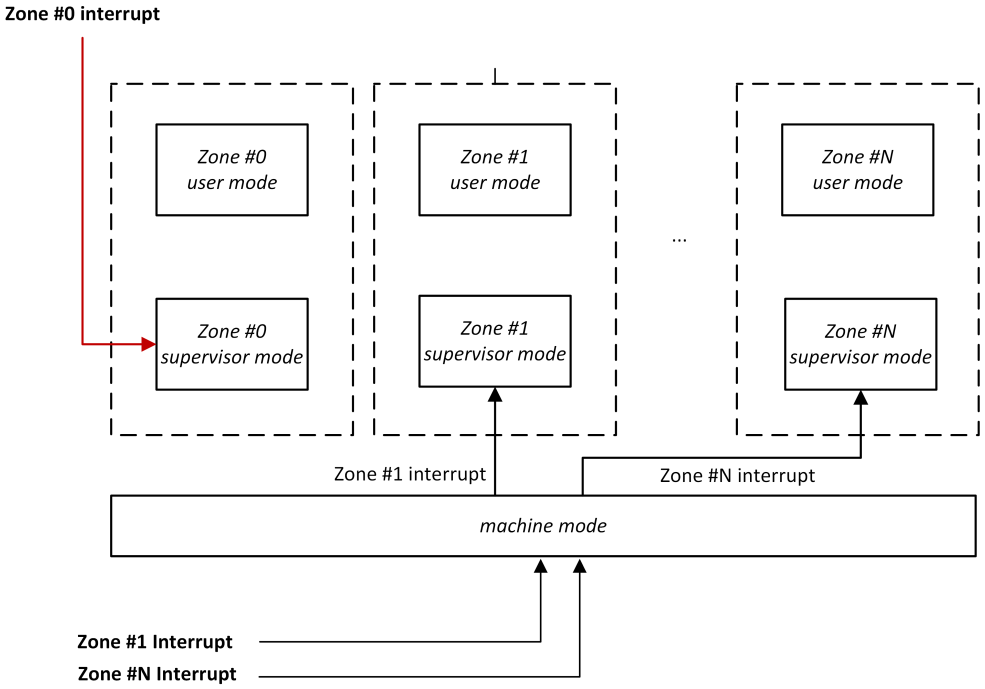


Fig. 9.9: **The Interrupt Handling Rule When the Processor Runs in Zone #0**

TEE generates regular interrupts; TEE generates secure interrupts

TF needs to perform the following operations when the processor runs in the TEE (Zone #1), as shown in Fig. 9.10 .

1. Enable S-mode for the interrupt source of regular interrupts.
2. Enable M-mode for the interrupt source of secure interrupts.

3. Reset the first bit (SSIE_DELEG), fifth bit (STIE_DELEG), and ninth bit (SEIE_DELEG) of the mideleg register (Assume that software interrupts and clock interrupts are configured as regular interrupts).
4. Enable mstatus.MIE and mstatus.SIE, and enable mie.MEIE, mie.MSIE, mie.MTIE, mie.SEIE, mie.SSIE, mie.STIE.

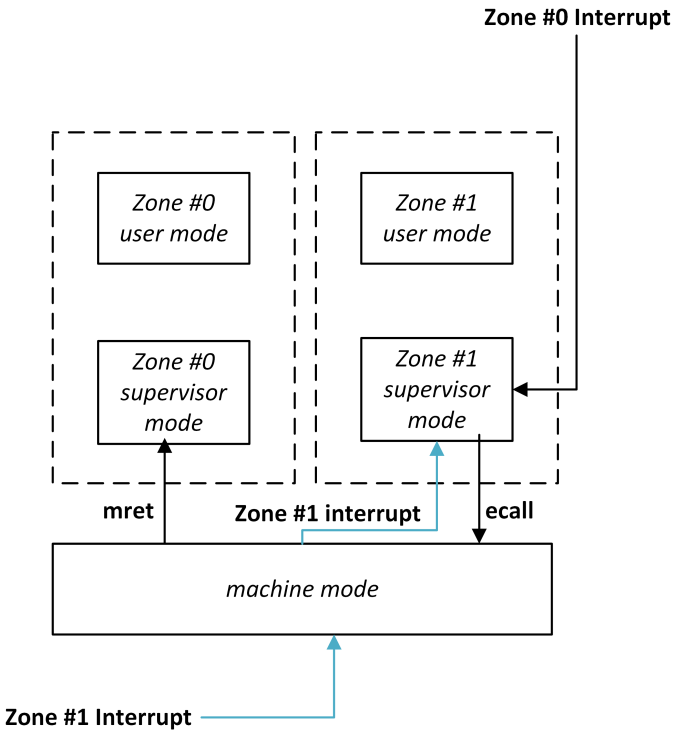


Fig. 9.10: **The Interrupt Handling Rule When the Processor Runs in Zone #1**

9.3.3 Secure Access Control

The M-mode is the most privileged mode that a hardware thread (hart) can run in RISC-V. The hart running in M-mode has full access permissions on memory, I/O, and underlying features that are required for booting and configuring the operating system. So M-mode is the privileged mode that must be implemented by all standard RISC-V processors. Actually, simple RISC-V microcontrollers only support M-mode.

The most significant feature of M-mode is exception interception and handling. By default, when an exception occurs (regardless of the privileged mode), the control permissions are transferred to the exception handler in M-mode. However, most exceptions in Linux should be handled in S-mode. The exception handler in M-mode can redirect exceptions to S-mode, but these additional operations will severely delay exception handling. RISC-V provides the exception delegation mechanism, which allows for selectively handing over interrupts and synchronous exceptions to S-mode for processing, completely bypassing M-mode. Machine Interrupt Delegation (mideleg) CSR controls the interrupts or exceptions that are transferred to the S-mode.

Please note that control permissions are not transferred to a mode with less privilege when an interrupt or exception occurs, regardless of the delegation settings. Interrupts and exceptions in M-mode are handled only in M-mode.

While interrupts and exceptions in S-mode are handled in M-mode or S-mode depending on the delegation settings, definately not in U-mode.

The M-mode is sufficient for simple embedded systems, but it is applicable only when the entire code repository is trusted, because M-mode provides full access to the hardware platform. The more common scenario is that not all application code can be trusted, as this cannot be known in advance, or it is too large and difficult to prove its correctness. RISC-V provides the mechanism to protect systems against untrusted code and isolate untrusted processes. These untrusted codes must be restricted to accessing only their own memory. Processors that have implemented M and S/U modes have a feature called Physical Memory Protection (PMP), which allows M-mode to specify the memory addresses that S/U mode can access. In addition to memory, PMP can also be applied to limiting the access to Memory-Mapped I/O (MMIO). M-mode can control the access of untrusted S/U mode to memory and devices.

9.3.4 Secure Debug

When the CPU secure debug is disabled, the CPU will ignore external synchronous debug requests, asynchronous debug requests, and the debug requests of internal hardware breakpoints. In this scenario, the EBREAK instruction will have no effect and will be treated as the nop instruction.

CHAPTER 10

Interrupt Controller

10.1 CLINT Interrupt Controller

C920 implements Core Local Interrupt Controller (CLINT), a memory address mapping module that handles software and timer interrupts.

10.1.1 CLINT Register Address Mapping

The CLINT controller occupies 64 KB memory space, where the upper 13 bits of the address are determined by the SoC hardware integration, and the lower 27 bits of the address are mapped as shown in Table 10.1. All registers only support word-aligned access. The CLINT adopts a contiguous addressing scheme, and for multi-cluster multi-core systems, the CLINT does not care about the number of clusters, but only focus on the number of cores. The address space for each core is contiguous. For example, there are two clusters, in which cluster 0 has 2 cores, and cluster 1 has 4 cores. And the register addresses for 2 cores in cluster 0 are described in core o and core 1. And the register addresses for 4 cores in cluster 1 are illustrated in core 2, core 3, core 4 and core 5. These corresponding register addresses are shown in the following table. CLINT supports a maximum of 256 cores.

Table 10.1: **Memory-mapped Address of CLINT**

Register	Address	Name	Type	Initial value	Description
MSIP	0x4000000	MSIP0	Read/Write	0x00000000	The machine-mode (M-mode) software interrupt configuration register for core 0. The upper bits are tied to 0, and bit [0] is valid
	0x4000004	MSIP1	Read/Write	0x00000000	The M-mode software interrupt configuration register for core 1. The upper bits are tied to 0, and bit [0] is valid.
	...	...	...	...	...
	0x400003c	MSIP15	Read/Write	0x00000000	The M-mode software interrupt configuration register for core 15. The upper bits are tied to 0, and bit [0] is valid.
	0x4000040	MSIP16	Read/Write	0x00000000	The M-mode software interrupt configuration register for core 16. The upper bits are tied to 0, and bit [0] is valid.

Continued on next page

Table 10.1 – continued from previous page

Register	Address	Name	Type	Initial value	Description
	0x4000044	MSIP17	Read/Write	0x00000000	The M-mode software interrupt configuration register for core 17. The upper bits are tied to 0, and bit [0] is valid.
	...	...	...	...	...
	0x4000000+4*n	MSIPn	Read/Write	0x00000000	n=hart_id, n<256
MTIMECMP	0x4004000	MTIMECMPL0	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the lower 32 bits) for core 0.
	0x4004004	MTIMECMPH0	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the upper 32 bits) for core 0.
	0x4004008	MTIMECMPL1	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the lower 32 bits) for core 1.
	0x400400c	MTIMECMPH1	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the upper 32 bits) for core 1.
	...	...	...	...	...
	0x4004078	MTIMECMPL15	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the lower 32 bits) for core 15.

Continued on next page

Table 10.1 – continued from previous page

Register	Address	Name	Type	Initial value	Description
	0x400407c	MTIMECMPH15	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the upper 32 bits) for core 15.
	0x4004080	MTIMECMPL16	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the lower 32 bits) for core 16.
	0x4004084	MTIMECMPH16	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the upper 32 bits) for core 16.
	0x4004088	MTIMECMPL17	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the lower 32 bits) for core 17.
	0x400408c	MTIMECMPH17	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the upper 32 bits) for core 17.
	...	...	...	...	...
	0x4004000+8*n	MTIMECMPLn	Read/Write	0xFFFFFFFF	n=hart_id, n<256
	0x4004000+8*n+4	MTIMECMPHn	Read/Write	0xFFFFFFFF	n=hart_id, n<256
CLINT_MTIME	0x400bff8	CLINT_MTIMEL	Read-only	0x00000000	The M-mode clock timer (Xuantie Self-Expanding Register)

Continued on next page

Table 10.1 – continued from previous page

Register	Address	Name	Type	Initial value	Description
	0x400bffc	CLINT_MTIMEH	Read-only	0x00000000	The M-mode clock timer (Xuantie Self-Expanding Register)
SSIP	0x400c000	SSIP0	Read/Write	0x00000000	The supervisor-mode (S-mode) software interrupt for core 0. The upper bits are tied to 0, and bit [0] is valid.
	0x400c004	SSIP1	Read/Write	0x00000000	The S-mode software interrupt for core 1. The upper bits are tied to 0, and bit [0] is valid.
	...	...	...	...	...
	0x400c03c	SSIP15	Read/Write	0x00000000	The S-mode software interrupt for core 15. The upper bits are tied to 0, and bit [0] is valid.
	0x400c040	SSIP16	Read/Write	0x00000000	The S-mode software interrupt for core 16. The upper bits are tied to 0, and bit [0] is valid.
	0x400c044	SSIP17	Read/Write	0x00000000	The S-mode software interrupt for core 17. The upper bits are tied to 0, and bit [0] is valid.

Continued on next page

Table 10.1 – continued from previous page

Register	Address	Name	Type	Initial value	Description
	...	...	...	...	...
	0x400c000+4*n	SSIPn	Read/Write	0x00000000	n=hart_id, n<256
STIMECMP	0x400d000	STIMECMPL0	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the lower 32 bits) for core 0.
	0x400d004	STIMECMPH0	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the upper 32 bits) for core 0.
	0x400d008	STIMECMPL1	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the lower 32 bits) for core 1.
	0x400d00c	STIMECMPH1	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the upper 32 bits) for core 1.
	...	...	...	...	...
	0x400d078	STIMECMPL15	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the lower 32 bits) for core 15.
	0x400d07c	STIMECMPH15	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the upper 32 bits) for core 15.

Continued on next page

Table 10.1 – continued from previous page

Register	Address	Name	Type	Initial value	Description
	0x400d080	STIMECMPL16	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the lower 32 bits) for core 16.
	0x400d084	STIMECMPH16	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the upper 32 bits) for core 16.
	0x400d088	STIMECMPL17	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the lower 32 bits) for core 17.
	0x400d08c	STIMECMPH17	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the upper 32 bits) for core 17.
	...	...	...	...	...
	0x400d000+8*n	STIMECMPLn	Read/Write	0xFFFFFFFF	n=hart_id, n<256
	0x400d0+8*n+4	STIMECMPHn	Read/Write	0xFFFFFFFF	n=hart_id, n<256
CLINT_STIME	0x400fff8	CLINT_STIMEL	Read-only	0x00000000	The S-mode clock timer (Xuantie Self-Expanding Register)
	0x400fffc	CLINT_STIMEH	Read-only	0x00000000	The S-mode clock timer (Xuantie Self-Expanding Register)

10.1.2 Software Interrupts

CLINT supports generating software interrupts.

Software interrupts are controlled by the software interrupt configuration registers configured with address mapping, in which, M-mode software interrupts are controlled by M-mode Software Interrupt Pending (MSIP) register, and S-mode software interrupts are controlled by the S-mode Software Interrupt Pending (SSIP) register.

You can set the xSIP bit to 1 to generate software interrupts and clear software interrupts by resetting the xSIP bit to 0. CLINT S-mode software interrupt requests are valid only when the CLINTEE bit is enabled for the corresponding core.

In M-mode, all software interrupt registers support accessing and modifying. In S-mode, only the SSIP register supports accessing and modifying. But software interrupt registers in user mode (U-mode) does not support that.

MSIP and SSIP registers have the same structure. And the bit layout and definition of the registers are shown in Fig. 10.1 and Fig. 10.2.

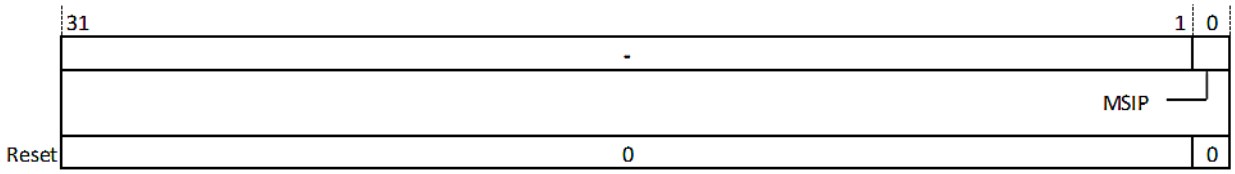


Fig. 10.1: **M-mode Software Interrupt Pending (MSIP) Register**

MSIP: the M-mode software interrupt pending bit

This bit indicates the interrupt status of M-mode software interrupts.

- When the MSIP bit is 1, valid M-mode software interrupt requests are available.
- When the MSIP bit is 0, valid M-mode software interrupt requests are unavailable.

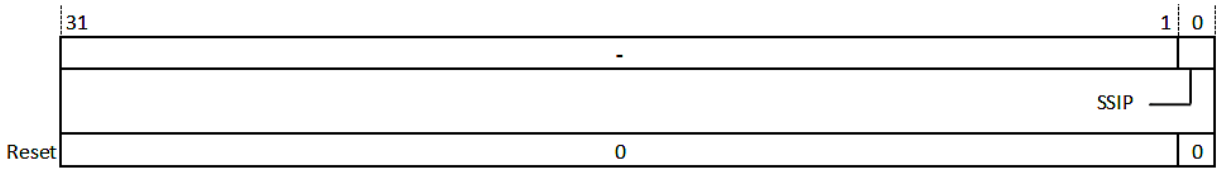


Fig. 10.2: **S-mode Software Interrupt Pending (SSIP) Register**

SSIP: the S-mode software interrupt pending bit

This bit indicates the interrupt status of S-mode software interrupts.

- When the SSIP bit is 1, valid S-mode software interrupt requests are available.
- When the SSIP bit is 0, no valid S-mode software interrupt requests are available.

10.1.3 Timer

In a multi-core multi-cluster system, there is only one 64-bit system timer that operates in the always-on voltage domain. The system timer is not writable and can only be cleared through a reset. The current value of the system timer can be obtained by reading the M-Mode Clock Timer Register (CLINT_MTIME) and S-mode Timer Register (CLINT_STIME), or by reading the TIME register of the Performance Monitoring Unit (PMU). The key feature of the system timer is to provide a unified event reference for multiple cores.

In a multi-core multi-cluster system, there is only one set of 64-bit M-mode Clock Timer registers (CLINT_MTIMEL, CLINT_MTIMEH) and one set of 64-bit S-mode Clock Timer registers (CLINT_STIMEL, CLINT_STIMEH). These registers can be read by the upper or lower 32 bits of these registers through word-aligned address access.

Notes:

CLINT_MTIME and CLINT_STIME are Xuantie self-extending registers.

CLINT_MTIMEH/CLINT_MTIMEL: The high/low bits of the M-mode clock timer register, storing the value of the clock timer.

- CLINT_MTIMEH: The high 32 bits of clock timer;
- CLINT_MTIMEL: The lower 32 bits of clock timer.

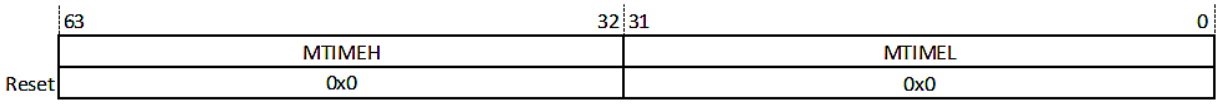


Fig. 10.3: **CLINT_MTIME Register**

CLINT_STIMEH/CLINT_STIMEL: The high/low bits of the S-mode clock timer register, storing the value of the clock timer.

- CLINT_STIMEH: The high 32 bits of clock timer;
- CLINT_STIMEL: The high 32 bits of clock timer;

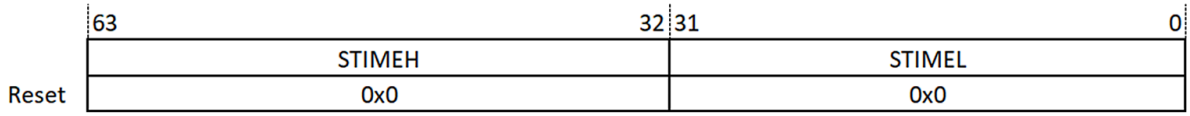


Fig. 10.4: **CLINT_STIME Register**

10.1.4 Timer Interrupts

CLINT can be used to generate timer interrupts. Each core of C920 has a set of 64-bit M-mode Clock Timer Compare Value Registers (MTIMECMPL, MTIMECMPH) and a set of 64-bit S-mode Clock Timer Compare Value Registers (STIMECMPL, STIMECMPH). These registers can modify either the high 32 bits or the low 32 bits by word-aligned address access. The register structure is the same in each set, and the bit distribution and definitions are illustrated in Fig. 10.5 and Fig. 10.6.

In M-mode, all timer interrupt registers support accessing and modifying. In S-mode, only the S-mode Clock Timer Compare Value Register (STIMECMPL, STIMECMPH) supports accessing and modifying. But software interrupt registers in U-mode does not support that.

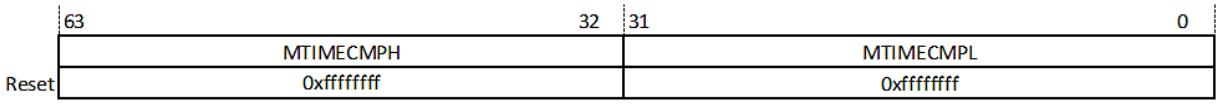


Fig. 10.5: **M-mode Timer Interrupt Compare Value Register (High Bit/Low Bit)**

MTIMECMPH/MTIMECMPL: The Low Bit/High Bit of M-mode Timer Interrupt Compare Value Register

- MTIMECMPH: The upper 32 bits of timer compare value;
- MTIMECMPL: The lower 32 bits of timer compare value.

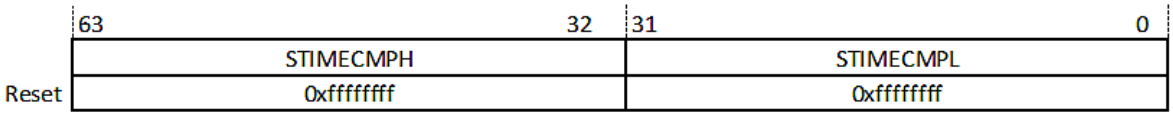


Fig. 10.6: **S-mode Timer Interrupt Compare Value Register (High Bit/Low Bit)**

STIMECMPH/STIMECMPL: The Low Bit/High Bit of S-mode Timer Interrupt Compare Value Register

- STIMECMPH: The upper 32 bits of timer compare value;
- STIMECMPL: The lower 32 bits of timer compare value.

CLINT determines whether to generate a timer interrupt by comparing the value of {CMPH[31:0], CMPL[31:0]} with the current value of the system timer:

If {CMPH[31:0], CMPL[31:0]} is greater than the value of the system timer, no interrupt is generated;

If {CMPH[31:0], CMPL[31:0]} is less than or equal to the value of the system timer, CLINT generates the corresponding timer interrupt.

Software can clear the corresponding timer interrupt by modifying the value of MTIMECMP/STIMECMP. In this scenery, the S-mode timer interrupt request is effective only when the corresponding core enables the CLINTEE bit and the STCE field of the corresponding core’ s M-mode Environment Configuration Register (MENVCFG) are set to zero.

10.2 PLIC

The Platform-level interrupt controller (PLIC) supports sampling, priority arbitration, and distribution of external interrupt sources.

In the PLIC model, the M-mode and S-mode of each core can act as valid interrupt targets.

The basic features of PLIC in C920 are as follow:

- PLIC supports up to 256 cores, and each core provides 2 targets: M-mode and S-mode.
- Supports up to 1023 interrupt sources sampling and both valid or pulse interrupt interrupts.
- Supports 32 interrupt priorities;
- Supports independent maintenance of interrupt enable for each interrupt target.
- Supports independent maintenance of interrupt threshold for each interrupt target.
- Supports configurable access permissions on PLIC registers.

10.2.1 Arbitration of Interrupts

In PLIC, only interrupt sources that meet certain conditions will participate in the arbitration for a particular interrupt target. And the conditions are as follows:

- The interrupt source is in the pending state (IP = 1).
- The interrupt priority is greater than 0.
- The enable bit for the interrupt target is enabled.

In PLIC, when there are multiple interrupts in the pending state for a particular interrupt target, the PLIC selects the interrupt with the highest priority through arbitration. In the PLIC of C920, M-mode interrupts always have higher priority than S-mode interrupts. When the privilege modes are the same, the greater the value of the priority configuration register, the higher the priority. If the priority value is 0, the interrupt is invalid; If multiple interrupts have the same priority value, the one with the smaller ID will be processed first.

The PLIC updates the arbitration result in the form of an interrupt ID, and sends the ID into the corresponding interrupt response/completion register for the respective interrupt target.

10.2.2 Request and Response of Interrupts

PLIC sends the interrupt request to the interrupt target when the PLIC has a valid interrupt request for a particular interrupt target and the interrupt priority is higher than the interrupt threshold of the interrupt target. When receiving the interrupt request, the interrupt target sends an interrupt response message to the PLIC if it is able to respond to the interrupt request.

The interrupt response mechanism is as follows:

- The interrupt target initiates a read operation to the corresponding interrupt response/complete register. Then the read operation returns the current interrupt ID determined by the PLIC. After that, the interrupt target proceeds to further processing based on the interrupt ID. If the interrupt ID is 0, no valid interrupt request is available, and the interrupt target ends the interrupt handling process.
- After receiving the read operation initiated by the interrupt target and returning the related interrupt ID, the PLIC clears the IP bit of the interrupt source corresponding to the interrupt ID to 0, and blocks subsequent sampling on the interrupt source before the current interrupt is completed.

Configuring L2 ECC feature, L2 ECC FATAL interrupt ID is determined by the customer' s integration of the interrupt controller.

10.2.3 Interrupt Completion

After interrupt handling is completed, the interrupt target needs to send an interrupt completion message to the PLIC. The interrupt completion mechanism is as follows:

- The interrupt target initiates a write operation to the interrupt response/completion register, and the value of the write operation is the current completion interrupt ID. If the interrupt is a level interrupt, the external interrupt source must be cleared before the write operation is initiated.
- After receiving the interrupt completion message, the PLIC does not update the interrupt claim/complete register, but unblocks sampling on the interrupt source corresponding to the interrupt ID to end the interrupt handling process.

10.2.4 PLIC Register Address Mapping

PLIC occupies 64MB memory space, where the upper 13-bit address is determined by the SoC hardware integration, and the lower 27-bit address mapping is shown in Table 10.2. All registers support only word-aligned address access, which means PLIC registers are accessible through the load word instruction. The access results are placed in the lower 32 bits of 64-bit General Purpose Register (GPR). The CLINT adopts a contiguous addressing scheme, and for multi-cluster multi-core systems, the CLINT does not care about the number of clusters, but only focus on the number of cores. The address space for each core is contiguous. For example, there are two clusters, in which cluster 0 has 2 cores, and cluster 1 has 4 cores. And the register addresses of the 2 cores in cluster 0 are described in core o and core 1. And the register addresses of 4 cores in cluster 1 are illustrated in core 2, core 3, core 4 and core 5. These corresponding register addresses are shown in the following table.

Table 10.2: PLIC register address mapping

Register	Address	Name	Type	Initial Value	Description
PLIC_PRIO	0x0000000	-	-	-	-
	0x0000004	PLIC_PRIO1	R/W	0x0	The priority configuration register for interrupt sources from 1 to 1023.
	0x0000008	PLIC_PRIO2	R/W	0X0	
	0x000000C	PLIC_PRIO3	R/W	0x0	
	...	...	...	...	
	0x0000FFC	PLIC_PRIO1023	R/W	0x0	
PLIC_IP	0x0001000	PLIC_IP0	R/W	0x0	The interrupt pending register for interrupts 1 to 31.

Continued on next page

Table 10.2 – continued from previous page

Register	Address	Name	Type	Initial Value	Description
	0x0001004	PLIC_IP1	R/W	0x0	The interrupt pending register for interrupts 32 to 63.
	...	...	...	...	...
	0x000107C	PLIC_IP31	R/W	0x0	The interrupt pending register for interrupts 992 to 1023.
-	Reserved	-	-	-	-
PLIC_MIE PLIC_SIE	0x0002000	PLIC_H0_MIE0	R/W	0x0	The M-mode interrupt enable register for core 01 to 31.
	0x0002004	PLIC_H0_MIE1	R/W	0x0	The M-mode interrupt enable register for core 032 to 63.
	...	...	...	...	...
	0x000207C	PLIC_H0_MIE31	R/W	0x0	The M-mode interrupt enable register for core 0992 to 1023.
	0x0002080	PLIC_H0_SIE0	R/W	0x0	The S-mode interrupt enable register for core 01 to 31.
	0x0002084	PLIC_H0_SIE1	R/W	0x0	The S-mode interrupt enable register for core 032 to 63.
	...	...	...	...	...
	0x00020FC	PLIC_H0_SIE31	R/W	0x0	The S-mode interrupt enable register for core 0992 to 1023.
	0x0002100	PLIC_H1_MIE0	R/W	0x0	The M-mode interrupt enable register for core 11 to 31.
	0x0002104	PLIC_H1_MIE1	R/W	0x0	The M-mode interrupt enable register for core 132 to 63.
	...	...	...	...	...
	0x000217C	PLIC_H1_MIE31	R/W	0x0	The M-mode interrupt enable register for core 1992 to 1023.

Continued on next page

Table 10.2 – continued from previous page

Register	Address	Name	Type	Initial Value	Description
	0x0002180	PLIC_H1_SIE0	R/W	0x0	The S-mode interrupt enable register for core 11 to 31.
	0x0002184	PLIC_H1_SIE1	R/W	0x0	The S-mode interrupt enable register for core 132 to 63
	...	...	...	...	...
	0x00021FC	PLIC_H1_SIE31	R/W	0x0	The S-mode interrupt enable register for core 1992 to 1023 .
	...	...	...	...	...
	0x0002000 +0x100*n	PLIC_Hn_MIE0	R/W	0x0	Core hart_id 1 to 31; The M-mode interrupt enable register; n=hart_id , n<256
	0x0002004 +0x100*n	PLIC_Hn_MIE1	R/W	0x0	Core hart_id 32 to 63; The M-mode interrupt enable register; n=hart_id , n<256
	...	...	...	...	...
	0x000207C +0x100*n	PLIC_Hn_MIE31	R/W	0x0	Core hart_id 992 to 1023; The M-mode interrupt enable register; n=hart_id , n<256
	0x0002080 +0x100*n	PLIC_Hn_SIE0	R/W	0x0	Core hart_id 1 to 31; The S-mode interrupt enable register; n=hart_id , n<256
	0x0002084 +0x100*n	PLIC_Hn_SIE1	R/W	0x0	Core hart_id 32 to 63; The S-mode interrupt enable register; n=hart_id , n<256
	...	...	...	...	...

Continued on next page

Table 10.2 – continued from previous page

Register	Address	Name	Type	Initial Value	Description
	0x00020FC +0x100*n	PLIC_Hn_SIE31	R/W	0x0	Core hart_id 992 to 1023; The S-mode interrupt enable register; n=hart_id , n<256
PLIC_CTRL	0x01FFFFC	PLIC_CTRL	R/W	0x0	The PLIC permission control register.
PLIC_MTH PLIC_MCLAIM PLIC_STH PLIC_SCLAIM	0x0200000	PLIC_H0_MTH	R/W	0x0	The M-mode interrupt threshold register for core 0.
	0x0200004	PLIC_H0_MCLAIM	R/W	0x0	The M-mode interrupt response/complete register for core 0.
	Reserved	-	-	-	-
	0x0201000	PLIC_H0_STH	R/W	0x0	The S-mode interrupt threshold register for core 0.
	0x0201004	PLIC_H0_SCLAIM	R/W	0x0	The S-mode interrupt response/complete register for core 0.
	Reserved	-	-	-	-
	0x0202000	PLIC_H1_MTH	R/W	0x0	The M-mode interrupt threshold register for core 1.
	0x0202004	PLIC_H1_MCLAIM	R/W	0x0	The M-mode interrupt response/complete register for core 1.
	Reserved	-	-	-	-
	0x0203000	PLIC_H1_STH	R/W	0x0	The S-mode interrupt threshold register for core 1.
	0x0203004	PLIC_H1_SCLAIM	R/W	0x0	The S-mode interrupt response/complete register for core 1.
	Reserved	-	-	-	-

Continued on next page

Table 10.2 – continued from previous page

Register	Address	Name	Type	Initial Value	Description
	0x0200000 +0x2000*n	PLIC_Hn_MTH	R/W	0x0	Core hart_id; The M-mode interrupt threshold register; n=hart_id , n<256
	0x0200004 +0x2000*n	PLIC_Hn_MCLAIM	R/W	0x0	Core hart_id; The M-mode interrupt response/complete register; n=hart_id , n<256
	Reserved	-	-	-	-
	0x0201000 +0x2000*n	PLIC_Hn_STH	R/W	0x0	Core hart_id; The S-mode interrupt threshold register; n=hart_id , n<256
	0x0201004 +0x2000*n	PLIC_Hn_SCLAIM	R/W	0x0	Core hart_id; The S-mode interrupt response/complete register; n=hart_id , n<256

As shown in Fig. 10.7, PLIC and CLINT occupy 128MB in overall address space, in which the base address is determined by pad_cpu_apb_base (For the input port, please check *C920 Integration Manual*). It is noted that the attribute of this space should be set as "Strong Ordered" .

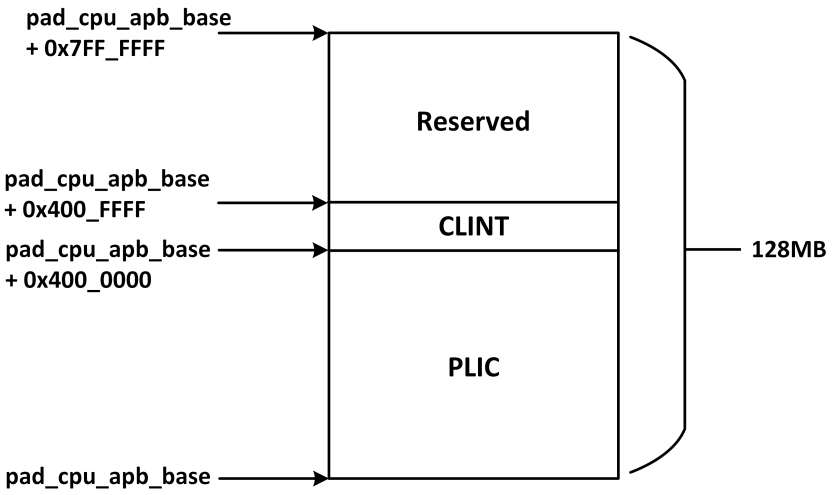


Fig. 10.7: **Address Space of PLIC&CLINT**

10.2.5 Interrupt Priority Configuration Register (PLIC_PRIO)

This PLIC_PRIO (PLIC_PRIO) register supports setting the priorities of interrupt sources. For the register read and write permissions, please refer to the descriptions of the Permission Control (PLIC_PER) register. And the corresponding bit layout and definition of the register are shown in Fig. 10.8.

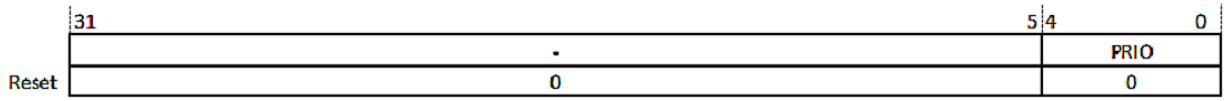


Fig. 10.8: **Interrupt Priority Configuration Register (PLIC_PRIO)**

PRIO: Interrupt priority

- The lower 5 bits of the priority configuration register are writable, which supports 32 different levels of priority. The priority setting of 0 indicates that the interrupt is invaild.
- In M-mode, interrupt priority is unconditionally higher than S-mode interrupt. When in the same mode, priority 1 is the lowest priority, and priority 31 is the highest.
- When priorities are the same, the interrupt source ID is further compared, with the smaller ID having higher priority.

10.2.6 Interrupt Pending Register (PLIC_IP)

The pending status of each interrupt source can be obtained by reading the information in the Interrupt Pending (PLIC_IP) Register. For an interrupt with ID N, the interrupt information is stored in the IP y (y = N mod 32) of the PLIC_IP x (x = N/32) register, where the first bit of the PLIC_IP0 register is fixed at 0. For the read and write permissions of the registers, please refer to the (PLIC_CTRL) register. The corresponding register bit distribution and bit definitions are as follows Fig. 10.9.

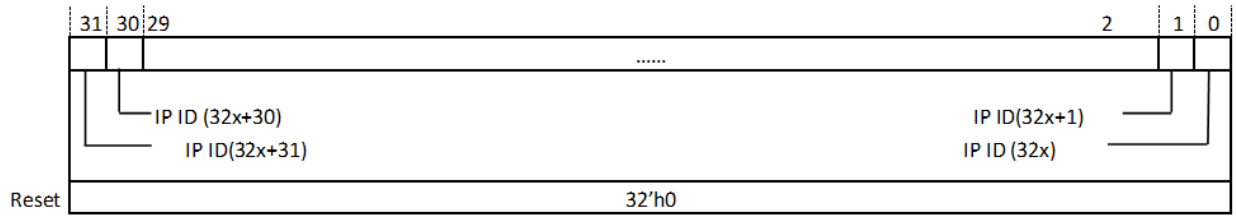


Fig. 10.9: **PLIC_IP x Interrupt Pending Register (PLIC_IP)**

IP: Interrupt pending Status

This bit indicates the interrupt pending state of the corresponding interrupt source.

- When the IP bit is 1, the current external interrupt source has interrupts pending for response. This bit can be set to 1 by a memory store instruction. When the corresponding interrupt source logic is sampled and detects a valid or pulse interrupt, this bit will also be set to 1.
- When the IP bit is 0, the interrupt source has no interrupts pending for response. This bit can be reset by a memory store instruction. PLIC clears the corresponding IP bit after an interrupt is responded.

10.2.7 Interrupt Enable Register (PLIC_IE)

Each interrupt target has an interrupt enable bit for each interrupt source, to enable the corresponding interrupts. The M-mode interrupt enable register enables M-mode external interrupts. The S-mode interrupt enable register enables S-mode external interrupts.

If the ID of an interrupt is N, the interrupt enable information is stored in IE y (y = N mod 32) in the Interrupt Enable PLIC_IE x (x = N/32) register. The IE bit corresponding to ID 0 is fixed to 0. For more information about the read and write permissions on the register, please refer to the descriptions of the PLIC_CTRL register.

The corresponding bit layout and definition of the register are shown in Fig. 10.10.

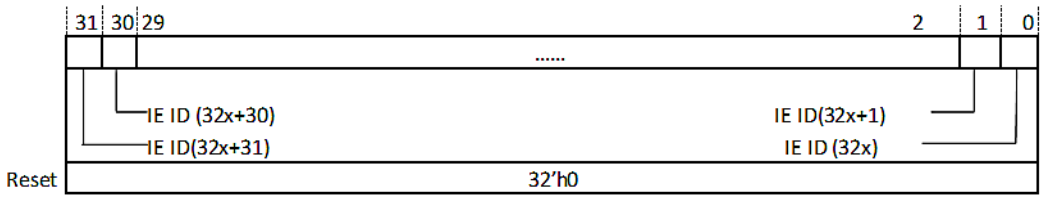


Fig. 10.10: **PLIC_IE x Interrupt Enable Register (PLIC_IE)**

IE Interrupt Enable:

This bit indicates the interrupt enable state of the corresponding interrupt source.

- When the IE bit is 1, it indicates that the interrupt is enabled for the target.
- When the IE bit is 0, it indicates that the interrupt is masked for the target.

10.2.8 PLIC Permission Control Register (PLIC_CTRL)

The Permission Control (PLIC_CTRL) register is designed to control access permissions on some PLIC registers in S-mode.

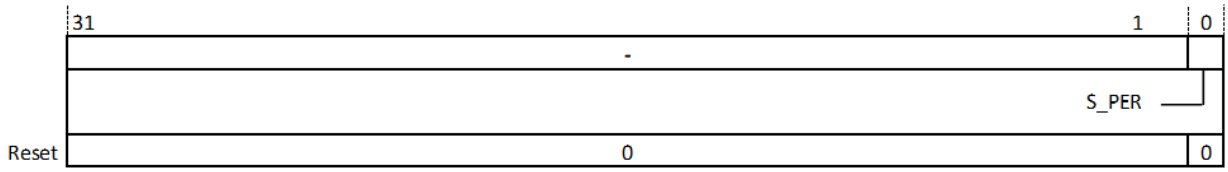


Fig. 10.11: **PLIC Permission Control Register (PLIC_CTRL)**

S_PER access permission control bit:

- When the S_PER bit is 0, Only M-mode has access to all the registers of PLIC. S-mode does not have access to PLIC permission control registers, interrupt priority configuration registers, interrupt pending registers, and interrupt enable registers, and can only access the S-mode interrupt threshold register and the S-mode interrupt response/complete register. But U-mode does not have access to any PLIC registers.
- When S_PER bit is 1, M-mode has all permissions. S-mode could access to all PLIC registers except for PLIC permission control registers. U-mode does not have access to any PLIC registers.

10.2.9 PLIC Threshold Register (PLIC_TH)

Each interrupt target has a corresponding Interrupt Thread (PLIC_TH) register. Only valid interrupts with priorities greater than the interrupt threshold will initiate an interrupt request to the interrupt target. For the read and write permissions of the register, please refer to the PLIC_CTRL register.

The corresponding bit layout and definition of the register are shown in Fig. 10.12.

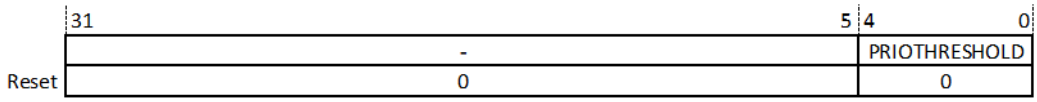


Fig. 10.12: **Interrupt Thread Register (PLIC_TH)**

PRIOTHRESHOLD Priority Threshold Value:

Indicates the interrupt threshold value of the current interrupt target. And if the value is 0, all interrupts are allowed.

10.2.10 Interrupt Response/Completion Register (PLIC_CLAIM)

Each interrupt target has a corresponding Interrupt Response/Completion (PLIC_CLAIM) register. When the PLIC completes arbitration, this register is updated to the interrupt ID obtained in the current arbitration. For more information about the read and write permissions on the register, please refer to the descriptions of the PLIC_CTRL register.

The corresponding bit layout and definition of the register are shown in Fig. 10.13.

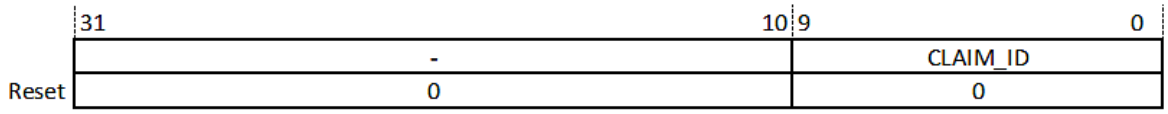


Fig. 10.13: **Interrupt Response/Completion Register (PLIC_CLAIM)**

CLAIM_ID: Interrupt request ID:

- Read operation on this register: Returns the current ID value stored in the register. This read operation indicates that the corresponding interrupt has started processing. PLIC initiates interrupt response processing.
- Write operation on this register: Indicates that the interrupt corresponding to the written value has completed processing. This write operation does not update the interrupt response/completion register. PLIC initiates interrupt completion processing.

10.3 Multi-core Interrupts

This section describes two common multi-core interrupt scenarios briefly.

10.3.1 Multiple Cores Respond to External Interrupts in Parallel

In the PLIC model, it is allowed to map a single interrupt source to multiple cores. When this interrupt source generates an interrupt request, it is in a pending state relative to multiple cores simultaneously. Each core will respond to this interrupt and read the CLAIM register to obtain the interrupt ID in a sequential manner, due to the different running states of the cores. The design of PLIC ensures that only the first core that reads the CLAIM register can obtain the real ID, while other cores will get an invalid ID (i.e., ID=0) and therefore will not process it. Thus, this interrupt will only be processed once.

Mapping a single interrupt to multiple cores can shorten the overall interrupt response time (as any one of the cores can potentially handle the interrupt), but it also occupies a portion of processor resources (as the cores that receive an invalid ID will consume bandwidth unnecessarily).

Suppose there are two external interrupt sources, Source 1 and Source 2, and the CPU is configured with 4 cores. Source 1 is mapped to Core 0, Core 1, and Core 2 simultaneously, while Source 2 is mapped to Core 1, Core 2, and Core 3, and Source 2 has a higher priority.

- When only Source 1 occurs, it can be handled by any one of Core 0, Core 1, and Core 2.
- When only Source 2 occurs, it can be handled by any one of Core 1, Core 2, and Core 3.
- When both interrupts occur simultaneously, there will be a priority arbitration between Core 1 and Core 2 and Source 2 wins as a result. Therefore, Source 2 may be handled by any of Core 1, Core 2, and Core 3, while Source 1 may be handled by Core 0.

10.3.2 Send Software Interrupts across Cores

In CLINT’ s programming model, there are specific registers for software interrupts, which are as follows:

- M-mode software interrupts: MSIP0, MSIP1, MSIP2, MSIP3
- S-mode software interrupts: SSIP0, SSIP1, SSIP2, SSIP3

The addresses of these 8 registers are the same and visible for all cores. Therefore, each core can send a software interrupt to any core (including itself) by performing write operations on these 8 registers.

CHAPTER 11

Bus Interface

11.1 Master Device Interface

Master device interface in C920MP supports AMBA 4.0 ACE or AXI protocol, please refer to *AMBA Specification —AMBA® AXI™ and ACE™ Protocol Specification*.

11.1.1 Features of the Master Device Interface

The master device interface is responsible for the address control and data transfer between C920 and the system bus. And the interface provides the following fundamental features:

- Supports the AMBA 4.0 ACE/AXI bus protocol.
- Supports a bus width of 128 bits.
- Supports different frequency ratios between the system clock and the CPU master clock.
- All output signals are flopped out, and input signals are flopped in to achieve better timing.

11.1.2 Outstanding Capability of the Master Device Interface

This section describes the outstanding capability of the master device interface in C920. The detailed information is as follows:

Table 11.1: **Outstanding Capability of the Master Device Interface**

Parameter	Value	Description
Read Issuing Capability	7n+42 n=Number of cores	Each core can issue a maximum of 7 non-cacheable and device read requests. The total maximum number of cacheable read requests is 42, with 28 requests from reading, writing data/TLB, and fetch refill to L1 and L2 prefetch, and 14 requests from writing data refill in L2.
Write Issuing Capability	12n+32 n=Number of cores	Each core can issue a maximum of 12 non-cacheable and Device write requests with up to 8 Device write requests. The total maximum number of cacheable write requests is 32, 32 Device or non-cacheable write request

Table 11.2: **ARID Encoding of the Master Device Interface**

ARID[7:0]	Applicable Scenarios	Outstanding Requests of Each ID
{2' b10, 6' b??????}	Cacheable Read requests	Each ID has no outstanding requests. All cacheable read requests are outstanding. A total of 42 outstanding requests are supported.
{1' b0, 2' b(coreid), 5' b00???}	Non-cacheable Weak-ordered Read requests	Each ID has no outstanding requests. Each core has a different ID for non-cacheable weak-ordered read requests. A total of 7 outstanding read requests with a combined with a combined total of 28 are supported.
{1' b0, 2' b(coreid), 5' h1e}	Non-cacheable Weak-ordered Lock preemption requests	Each ID has no outstanding requests.
{1' b0, 2' b(coreid), 5' h1d}	Non-cacheable Strong-ordered Read requests	A total of 7 outstanding for each ID are supported.

Table 11.3: **AWID Encoding of the Master Device Interface**

AWID[7:0]	Applicable Scenarios	Outstanding Requests of Each ID
{3' b111, 5' b?????}	Cacheable Write requests	Each ID has no outstanding requests. All cacheable write requests are outstanding. A total of 32 outstanding requests are supported.

Continued on next page

Table 11.3 – continued from previous page

AWID[7:0]	Applicable Scenarios	Outstanding Requests of Each ID
{2' b00, 1' b?, 1' b0, 4' h????}	Non-cacheable Weak-ordered write requests	Each ID has no outstanding requests. The maximum number of outstanding requests for each core with different ID is 12, and the global maximum is 3. All non-cacheable weak-ordered write requests have a maximum of 32 outstanding requests.
{1' b0, 2' b(coreid), 5' h1e}	Non-cacheable Weak-ordered Lock preemption write requests	Each ID has no outstanding requests.
{1' b0, 2' b(coreid), 5' h1d}	Non-cacheable Strong-ordered Write requests	Each ID has 8 outstanding requests. And the global maximum is 32. A total of 32 outstanding non-cacheable strong-ordered write requests are supported.

Note:

The ARID and AWID encoding may vary with evolution of the CPU version. Therefore, SoC integration should not depend on specific IDs, but conform to general-purpose rules of the AXI protocol.

11.1.3 Supported Transmission Types

The master device interface supports the following transmission types:

- BURST supports INCR and WRAP, while other burst types are not supported;
- LEN only supports transfer lengths of 1 or 4;
- Supports exclusive access;
- Transmission sizes supports quadword, doubleword, word, halfword, and byte, while other transmission sizes are not supported;
- Supports read and write.

Note:

The AXI master device interface of C920 implements only a subset of all AXI transmissions. SoC integration should not depend on specific transmission types, but conform to general-purpose rules of the AXI protocol.

11.1.4 Supported Response Types

The main device interface receives the response type from the slave device as follow:

- OKAY
- EXOKAY
- SLVERR
- DECERR

11.1.5 Behavior in Different Bus Responses

CPU behavior in different bus responses is shown in Table 11.4.

Table 11.4: **Bus Exception Handling**

RRESP/BRESP	Result
OKAY	Common transfer access succeeds or exclusive transfer access fails. If exclusive read transfer access fails, it indicates that the bus does not support exclusive transfer, and an access error exception is generated. If exclusive write transfer access fails, it indicates that lock preemption fails, and no exception is generated.
EXOKAY	Exclusive access succeeds.
SLVERR/DECERR	An access error occurs. If this error occurs in read transfer, an exception is generated. If this error occurs in write transfer, it is ignored.

11.1.6 Signals Supported by the Master Device Interface

All signals supported by the master device interface are listed in Table 11.5.

Table 11.5: **Master Interface Signal of AXI/ACE Bus**

Signal Name	I/O	Initial Value	Clock Domain	Definition
biu_pad_araddr[39:0]	O	40’ b0	SYS	Read address bus: 40-bit address bus
biu_pad_arburst[1:0]	O	2’ b0	SYS	Burst transfer indication signal: Indication transfer is part of a burst transfer. 01: INCR 10: WRAP4
biu_pad_arcache[3:0]	O	4’ b0	SYS	The cache attributes corresponding to read requests: [3]: Other Allocate; [2]: Allocate; [1]: Modifiable; [0]: Bufferable
biu_pad_arid[7:0]	O	8’ b0	SYS	Read address ID

Continued on next page

Table 11.5 – continued from previous page

Signal Name	I/O	Initial Value	Clock Domain	Definition
biu_pad_arlen[7:0]	O	8’ b0	SYS	Burst transfer length: 00000000: 1 beat; 00000011: 4 beats.
biu_pad_arlock	O	1’ b0	SYS	Access modes corresponding to read requests: 0: normal access; 1: exclusive access.
biu_pad_arprot[2:0]	O	3’ b0	SYS	Protection types of read requests: 0 | 1 [2]: Data | Instruction; [1]: Secure | Non-secure; [0]: User | Privileged.
biu_pad_arsize[2:0]	O	3’ b0	SYS	Data width per beat for read requests: 000: 8bits; 001: 16bits; 010: 32bits; 011: 64bits; 100: 128bits.
biu_pad_arvalid	O	1’ b0	SYS	Read address valid signals
pad_biu_arready	I	-	SYS	Read address channel ready signals
pad_biu_rdata[127:0]	I	-	SYS	Read data bus: 128-bit data bus
pad_biu_rid[7:0]	I	-	SYS	Read data ID
pad_biu_rresp[3:0]	I	-	SYS	Read response signals: [1:0]: 00: OKAY; 01: EXOKAY; 10: SLVERR; 11: DECERR.
pad_biu_rlast	I	-	SYS	Read data last-beat indication signal
pad_biu_rvalid	I	-	SYS	Read data valid signals
biu_pad_rready	O	1’ b1	SYS	Read data channel ready signals
biu_pad_awaddr[39:0]	O	40’ b0	SYS	Write address bus: 40-bit address bus
biu_pad_awburst[1:0]	O	2’ b0	SYS	Burst transfer indication signal: Indication transfer is part of a burst transfer. 01: INCR

Continued on next page

Table 11.5 – continued from previous page

Signal Name	I/O	Initial Value	Clock Domain	Definition
biu_pad_awcache[3:0]	O	4' b0	SYS	The cache attributes corresponding to write requests: [3]: Other Allocate; [2]: Allocate; [1]: Modifiable; [0]: Bufferable
biu_pad_awid[7:0]	O	8' b0	SYS	Write data ID
biu_pad_awlen[7:0]	O	8' b0	SYS	Burst transfer length: 00000000: 1 beat; 00000011: 4 beats.
biu_pad_awlock	O	1' b0	SYS	Access modes corresponding to write requests: 0: normal access; 1: exclusive access.
biu_pad_awprot[2:0]	O	3' b0	SYS	Protection types of write requests: 0 | 1 [2]: Data | Instruction; [1]: Secure | Non-secure; [0]: User | Privileged.
biu_pad_awsize[2:0]	O	3' b0	SYS	Data width per beat for write requests: 000: 8bits; 001: 16bits; 010: 32bits; 011: 64bits; 100: 128bits.
biu_pad_awvalid	O	1' b0	SYS	Write address valid signals
Pad_biu_awready	I	-	SYS	Write address channel ready signals
biu_pad_wvalid	O	1' b1	SYS	Write data valid signals
pad_biu_wready	I	-	SYS	Write data channel ready signals
biu_pad_wdata[127:0]	O	128' b0	SYS	Write data bus: 128-bit write data bus
biu_pad_wstrb[15:0]	O	16' b0	SYS	Write data byte valid signals
biu_pad_wlast	O	1' b0	SYS	Write data last-beat indication signal
pad_biu_bid[7:0]	I	-	SYS	Write response ID
pad_biu_bresp[1:0]	I	-	SYS	Write response signals: 00: OKAY; 01: EXOKAY; 10: SLVERR; 11: DECERR.
pad_biu_bvalid	I	-	SYS	Write response valid signals
biu_pad_bready	O	1' b1	SYS	Write response channel ready signals

Continued on next page

Table 11.5 – continued from previous page

Signal Name	I/O	Initial Value	Clock Domain	Definition
biu_pad_cactive	O	1’ b1	-	Fixed at 1
biu_pad_csysack	O	1’ b1	SYS	Low-power channel request response signal
pad_biu_csysreq	I	-	SYS	Low-power channel request signal
AXI/ACE Bus Master Interface Signal (ACE Mode Only)				
biu_pad_arbar[1:0]	O	2’ b0	SYS	AR channel expanded signal
biu_pad_ardomain[1:0]	O	2’ b0	SYS	
biu_pad_arsnoop[3:0]	O	4’ b0	SYS	
biu_pad_awbar[1:0]	O	2’ b0	SYS	AW channel expanded signal
biu_pad_awdomain[1:0]	O	2’ b0	SYS	
biu_pad_awsnoop[2:0]	O	3’ b0	SYS	
biu_pad_rack	O	1’ b0	SYS	RACK signal
biu_pad_wack	O	1’ b0	SYS	WACK signal
pad_biu_acaddr[39:0]	I	-	SYS	AC channel address
pad_biu_acprot[2:0]	I	-	SYS	AC channel signal
pad_biu_acsnoop[3:0]	I	-	SYS	
pad_biu_acvalid	I	-	SYS	
biu_pad_acready	O	1’ b0	SYS	
biu_pad_crresp[4:0]	O	5’ b0	SYS	CR channel signal
biu_pad_crvalid	O	1’ b0	SYS	
pad_biu_crready	I	-	SYS	
biu_pad_cddata[127:0]	O	128’ b0	SYS	CD channel signal
biu_pad_cdlast	O	1’ b0	SYS	
biu_pad_cdvalid	O	1’ b0	SYS	
pad_biu_cdready	I	-	SYS	

11.1.7 Supported Coherency Transaction Types

As ACE bus is chosen as master device interface, the supported Coherency Transaction types are shown in Table 11.6.

Table 11.6: **Coherency Transaction Types Supported by ACE Protocol**

Request Type	Source
ARSNOOP	
ReadNoSnoop (ARDOMAIN=2’ b11, ARLEN=0)	Non-cacheable weak-ordered read request; Non-cacheable strong-ordered read request
ReadOnce (ARDOMAIN=2’ b01, ARLEN=0/3)	No-allocate read requests of instruction, data, and TLB; DCP no-allocate read request when the corresponding cacheline in L2 is in a miss state.

Continued on next page

Table 11.6 – continued from previous page

Request Type	Source
ReadNotSharedDirty (ARDOMAIN=2' b01, ARLEN=3)	Allocate read requests of instruction, data, and TLB; Prefetch requests of instruction, data, and TLB; prefetch.r*/prefetch.i* instruction; DCP allocate read request when the corresponding cacheline in L2 is in a miss state.
ReadUnique (ARDOMAIN=2' b01, ARLEN=3)	The write requests when there is a cache line miss in L2 cache. Allocate partial write request; prefetch.w* instruction; DCP allocate partial write request when the corresponding cacheline in L2 is in a miss state; DCP no-allocate partial write request when the corresponding cacheline in L2 is in a miss or shared state.
CleanUnique (ARDOMAIN=2' b01, ARLEN=3)	Allocate partial write request when The corresponding cacheline in L2 cache is in a shared state. DCP allocate write request; when the corresponding cacheline in L2 is in a shared state.
MakeUnique (ARDOMAIN=2' b01, ARLEN=3)	Full cache line write request when L2 cacheline miss/shared; DCP full cacheline write request when the corresponding cacheline in L2 is in a miss state; DCP no allocate full cacheline write request when the corresponding cacheline in L2 is in a shared state.
CleanShared (ARDOMAIN=2' b01, ARLEN=3)	cbo.clean*/dcache.cva/dcache.cpa instruction
CleanInvalid (ARDOMAIN=2' b01, ARLEN=3)	cbo.flush*/dcache.civa/dcache.cipa instruction
MakeInvalid (ARDOMAIN=2' b01, ARLEN=3)	cbo.inval* /dcache.iva/dcache.ipa instruction
DVM Message (ARDOMAIN=2' b01, ARLEN=0)	icache.ialls/icache.ipa/icache.iva/fence.i/sfence/sync instruction, in which Sync instruction is only used to transfer DVM.
DVM Complete (ARDOMAIN=2' b01, ARLEN=0)	DVM handshake
AWSNOOP	
WriteNoSnoop (AWDOMAIN=2' b11, AWLEN=0, WSTRB is variable)	Non-cacheable weak-order write request; Non-cacheable strong-order write request
WriteClean (AWDOMAIN=2' b01, AWLEN=3, WSTRB all set to 1)	Execute cbo.clean*/dcache.cva/dcache.cpa instruction when the corresponding cacheline in L1/L2 is in a dirty state.

Continued on next page

Table 11.6 – continued from previous page

Request Type	Source
WriteBack (AWDOMAIN=2' b01, AWLEN=3, WSTRB all set to 1)	Execute cbo.flush*/dcache.civa/dcache.cipa instruction when the corresponding cacheline in L1/L2 is in a dirty state; No allocate write request; The replacement behavior of the L2 cache line when the corresponding cacheline in L1/L2 is in a dirty state; DCP no allocate write request when the corresponding cacheline in l2 is in a miss state.
Evict (AWDOMAIN=2' b01, AWLEN=3)	The replacement behavior of the L2 cache line when the corresponding cacheline in L1/L2 is in a clean state.

Note:

* : The instruction is CMO extended instruction. For specific information, please refer to *RISC-V Base Cache Management Operation ISA Extensions*.

DVM message types supported by C920 are as follow:

- TLB Invalidate
- Physical Instruction Cache Invalidate
- Synchronization

And the corresponding ARADDR is in the following table:

Table 11.7: **TLB Invalidate First Part**

ARADDR Bits	Name	Function
[43][1]	Reserved	1' b0
[42:40][1]	Virtual Address	VA[47:45]
[39:32]	ASID upper byte	ASID[15:8]
[31:24]	ZoneID	ZoneID[7:0]
[23:16]	ASID lower byte	ASID[7:0]
[15]	Completion	1' b0: DVM Completion is not required
[14:12]	Message type	3' b0: TLB Invalidate
[11:10]	Exception Level	2' b10: Applies to Guest OS
[9:8]	Security	2' b11: Applies to Non-secure only
[7]	SBZ or Range	1' b0: Message does not include address range information
[6:5]	VA or ASID valid	2' b00: Invalidate all TLB entries, ADDR[0] must be 0. 2' b01: Reserved 2' b10: Invalidate TLB entries by VA, ADDR[0] must be 1. 2' b11: - ADDR[0]=1' b0, Invalidate TLB entries by ASID - ADDR[0]=1' b1, Invalidate TLB entries by ASID and VA.
[4]	Leaf	1' b0: Invalidate all associated translations.
[3:2]	Stage	2' b0: Stage 1 and Stage 2 invalidation

Continued on next page

Table 11.7 – continued from previous page

ARADDR Bits	Name	Function
[1]	Reserved	1’ b0
[0]	Addr	1’ b0: The message does not require an address and has one part 1’ b1: The message includes an address and has two parts

Note:

1: Present when Sv48 selected.

Table 11.8: **TLB Invalidate Second Part**

ARADDR Bits	Description
[43:40][1]	VA[44:41]
[39:4]	VA[39:4]
[3]	Sv48: VA[40] Sv39: 1’ b0
[2:0]	3’ b0

Note:

1: Present when Sv48 selected.

Table 11.9: **Physical Instruction Cache Invalidate First Part**

ARADDR Bits	Name	Function
[43:40][1]	Reserved	4’ b0
[39:16]	Reserved	24’ b0
[15]	Completion	1’ b0: DVM Completion is not required
[14:12]	Message type	3’ b010: Physical Instruction Cache Invalidate
[11:10]	Exception Level	2’ b00: Applies to Hypervisor and all Guest OS
[9:8]	Security	2’ b11: Applies to Non-secure only
[7]	SBZ or Range	1’ b0: Message does not include address range information
[6:5]	VA or ASID valid	2’ b00: Invalidate all cache lines/ Invalidate cache line by PA.
[4:1]	Reserved	4’ b0
[0]	Addr	1’ b0: The message does not require an address and has one part 1’ b1: The message includes an address and has two parts

Note:

1:Present when Sv48 selected.

Table 11.10: **Physical Instruction Cache Invalidate Second Part**

ARADDR Bits	Description
[43:40][1]	4' b0
[39:4]	PA[39:4]
[3:0]	4' b0

Note:

1: Present when Sv48 selected.

Table 11.11: **Synchronization**

ARADDR Bits	Name	Function
[43:40][1]	Reserved	4' b0
[39:16]	Reserved	24' b0
[15]	Completion	1' b1: Completion required
[14:12]	Message type	3' b100: Synchronization
[11:10]	Exception Level	2' b00: Applies to Hypervisor and all Guest OS
[9:8]	Security	2' b00: Applies to Secure and Non-secure
[7]	SBZ or Range	1' b0: Message does not include address range information
[6:5]	VA or ASID valid	2' b00
[4:1]	Reserved	4' b0
[0]	Addr	1' b0: The message does not require an address and has one part

Note:

1: Present when Sv48 selected.

11.2 DCP

C920MP Device Coherence Port (DCP), is a user-configurable interface that enables peripherals to access the L2 cache and L1 D-Cache, ensuring data consistency between the peripherals and the processor' s on-chip data. DCP supports AMBA AXI4 Protocol (Please refer to *AMBA® AXI™ and ACE™ Protocol Specification*)

11.2.1 Features of DCP

The basic characteristics of a device consistency interface are as follow:

- Supports for the AMBA 4.0 AXI bus protocol.
- Supports for a 128-bit bus width.
- Supports for different frequency ratios between the system clock and the CPU main clock.
- All output signals are flopped out, and input signals are flopped in to achieve better timing.
- Supports for up to 8 concurrent transfers for both read and write operations.

11.2.2 Supported Transfer Types

The transfer characteristics supported by the device consistency interface are as follows:

- Only supports INCR transfer mode, and LEN only supports 0 and 3;
- Requires CACHE[3:0] to be 4' b1111, 4' b1011, 4' b0111, otherwise return SLVERR;
- Requires SIZE[2:0] to be 3' b100, otherwise returns SLVERR;
- Exclusive access is not supported;
- WSTRB: When LEN is 0, it supports any byte enablement; when LEN is 3, all bits must be set to 1.;
- AxADDR: When LEN is 0, the address is 16-byte aligned; when LEN is 3, the address is 64-byte aligned;
- Supports 32-bit AxID;
- Supports read and write operations.

the cache line where the data is located is neither in the L1 D CACHE nor in the L2 CACHE.

11.2.3 L2 cache Allocation Behavior under Different Transfers

For read requests, when ARCACHE[3:0] = 4' b1111 or 4' b0111, and the cache line where the data is located is neither in the L1 D CACHE nor in the L2 CACHE upon the triggering of the read request, the cache line will be allocated to L2 after the read request is completed. This allocation may trigger L2 replacement. If ARCACHE[3:0] = 4' b1011, it will not have any impact on L1 and L2.

For a write request, if AWCACHE[3:0] = 4'b1111 or 4'b1011, the updated cache line data will be allocated to L2 upon completion of the write request, regardless of whether the cache line in which the data locates is in the L2 CACHE upon the triggering of the write request. This allocation may trigger L2 replacement; While if AWCACHE[3:0] = 4' b0111, when the cache line where the data locates is in the L2 CACHE upon the triggering of the write request, then the write request will update the data of the cacheline and write it back to the L2, otherwise it will be written out directly through the external bus.

11.2.4 Supported Response Types

The response types supported by DCP are as follows:

- OKAY;
- SLVERR。

11.2.5 Responses under Different Behaviors

The resopnse types returened from slave devices are illustrated in Table 11.12.

Table 11.12: **Response Types of Slave Devices**

RRESP/BRESP	Result
OKAY	The transfer access is successful, and the received request is appropriately processed.
SLVERR	An access error occurred or an unsupported transmission type was received;

11.2.6 DCP Signals

Table 11.13: **DCP Signals**

Signal	I/O	Reset	Definition
The Interfaces Related to Reading Address Channels			
pad_slvif_araddr[39:0]	I	-	Read address bus: 40-bit address bus
pad_slvif_arburst[1:0]	I	-	Burst transfer indication signal: Indication transfer is part of a burst transfer. 01: INCR
pad_slvif_arcache[3:0]	I	-	The cache attributes corresponding to read requests: [3]: Other Allocate [2]: Allocate [1]: Modifiable [0]: Bufferable
pad_slvif_arid[4:0]	I	-	Read address ID
pad_slvif_arlen[7:0]	I	-	Burst transfer length: 00000000: 1 beat; 00000011: 4 beats
pad_slvif_arlock	I	-	Access modes corresponding to read requests: 0: normal access 1: exclusive access
pad_slvif_arprot[2:0]	I	-	Protection types of read requests: 0 | 1 [2]: Data | Instruction [1]: Secure | Non-Secure [0]: User | Privileged
pad_slvif_arsize[2:0]	I	-	Data width per beat for read requests: 100: 128bits.
pad_slvif_arvalid	I	-	Read address valid signals
slvif_pad_arready	O	1’ b1	Read address channel ready signals
The Interfaces Related to Reading Data Channels			
slvif_pad_rdata[127:0]	O	128’ b0	Read data bus: 128-bit data bus
slvif_pad_rid[4:0]	O	5’ b0	Read data ID
slvif_pad_rresp[3:0]	O	4’ b0	Read response signals: [1:0]: 00: OKAY 10: SLVERR [2]: 1: IsShared [3]: undefined
slvif_pad_rlast	O	1’ b0	Read data last-beat indication signal
slvif_pad_rvalid	O	1’ b0	Read data valid signals

Continued on next page

Table 11.13 – continued from previous page

Signal	I/O	Reset	Definition
pad_slvif_rready	I	-	Read data channel ready signals
The Interfaces Related to Writing Address Channels			
pad_slvif_awaddr[39:0]	I	-	Write address bus: 40-bit address bus
pad_slvif_awburst[1:0]	I	-	Burst transfer indication signal Indication transfer is part of a burst transfer 01: INCR
pad_slvif_awcache[3:0]	I	-	The cache attributes corresponding to write requests [3]: Other Allocate [2]: Allocate [1]: Modifiable [0]: Bufferable
pad_slvif_awid[4:0]	I	-	Write data ID
pad_slvif_awlen[7:0]	I	-	Burst transfer length: 00000000: 1 beat; 00000011: 4 beats
pad_slvif_awlock	I	-	Access modes corresponding to write requests: 0: normal access 1: exclusive access
pad_slvif_awprot[2:0]	I	-	Protection types of write requests: 0 | 1 [2]: Data | Instruction [1]: Secure | Non-Secure [0]: User | Privileged
pad_slvif_awsize[2:0]	I	-	Data width per beat for write requests: 100: 128bits
pad_slvif_awvalid	I	-	Write address valid signals
slvif_pad_awready	O	1’ b1	Write address channel ready signalas
The Interfaces Related to Writing Data Channels			
pad_slvif_wdata[127:0]	I	-	Write data bus: 128-bit write data bus
pad_slvif_wstrb[15:0]	I	-	Write data byte valid signals
pad_slvif_wlast	I	-	Write data last-beat indication signal
pad_slvif_wvalid	I	-	Write data valid signals
slvif_pad_wready	O	1’ b1	Write data channel ready signals
The Signals Related to Writing Response Channels			
slvif_pad_bid[4:0]	O	5’ b0	Write response ID
slvif_pad_bresp[1:0]	O	2’ b0	Write response signals: 00: OKAY; 10: SLVERR
slvif_pad_bvalid	O	1’ b0	Write response valid signals
pad_slvif_bready	I	-	Write response channel ready signals

11.3 LLP

C920MP Low Latency Port (LLP), is a user-configurable master interface that enables to access system peripherals. LLPP supports AMBA AXI4 Protocol (Please refer to AMBA Specification——*AMBA® AXI™ and ACE™ Protocol Specification*)

11.3.1 The Features of LLP

- Supports AX14.0 protocol;
- Supports 128-bit data bus width and 40-bit address bus width;
- Supports up to 7 read outstanding and 12 write outstanding operations per core and a maximum of 28 read outstanding and 32 write outstanding operations for four cores.
- Supports integer multipliers for CPU and LLP with a ratio of 1:N (N <= 8).
- Supports all bus responses.
- Supports unaligned memory accesses.

11.3.2 The Outstanding Capability of LLP

The outstanding capability of C920 LLP is listed in this section, and the details are as follow.

Table 11.14: **The Outstanding Capability of LLP**

Parameter	**Value**	**Description**
Read Issuing Capability	7n, n = the number of cores	Up to 7 read requests per core, and a maximum of 28 for all four cores
Write Issuing Capability	12n, n = the number of cores	Up to 12 read requests per core, and a maximum of 32 for all four cores

Table 11.15: **AXI LLP ARID Encoding**

ARID[7:0]	**Applicable Scenarios**	**Outstanding for Each ID**
{1' b0, 2' b(coreid), 5' b00???}	Non-cacheable weak-ordered read requests	The total number of outstanding non-cacheable read requests is 28.
{1' b0, 2' b(coreid), 5' h10}	Non-cacheable weak-ordered instruction fetch requests	
{1' b0, 2' b(coreid), 5' h1e}	Non-cacheable weak-ordered lock contention requests	
{1' b0, 2' b(coreid), 5' h1d}	Non-cacheable strong-ordered read requests	

- In the AR channel, the same ID will not appear simultaneously at the master device interface and LLP.

Table 11.16: **AXI LLP AWID Encoding**

AWID[7:0]	Applicable Scenarios	Outstanding for Each ID
{2' b00, 1' b?, 1' b0, 4' h????}	Non-cacheable weak-ordered write requests	There is no outstanding for each ID. The total number of outstanding non-cacheable write requests is 32.
{1' b0, 2' b(coreid), 5' h1e}	Non-cacheable weak-ordered lock contention requests	There is no outstanding for each ID.
{1' b0, 2' b(coreid), 5' h1e}	Non-cacheable strong-ordered lock contention requests	The total number of outstanding non-cacheable strong-ordered write requests is 32.

- In the AR channel, the same ID will not appear simultaneously at the master device interface and LLP.

Note:

The ARID/AWID encoding may evolve with successive processor versions; therefore, SoC integration should not rely on specific ID values but rather adhere to the general rules of the AXI protocol.

11.3.3 Supported Transfer Types

The LLP supports the following transfer characteristics:

- Only INCR transfers are supported; FIXED and WRAP transfers are not supported.
- LEN supports only the value 8' b0.
- Supports exclusive access.
- Supports accesses to noncacheable normal memory and devices.
- Supports unaligned accesses.
- The size supports a range from 3' b000 to 3' b100, corresponding to sizes from 1B to 16B.
- Write merging is supported for normal memory non-cacheable transfers; wstrb can issue any value, and Axsize is fixed to 3' b100.
- Axsize may take values from 3' b000 to 3' b011 for device transfers

Note:

The LLP in C920 implements only a subset of all AXI transfers. However, SoC integration should not depend on specific transfer types but instead comply with the general rules of the AXI protocol.

11.3.4 Supported Response Types

The LLP supports the following response types:

- OKAY
- EXOKAY
- SLVERR
- DECERR

CHAPTER 12

Debug

12.1 Features of Debug Unit

C920 is compatible with RISC-V Debug V0.13.2 Protocol standard. And the external debugging interface supports for 5-wire JTAG (Standard JTAG5) mode. The debug interface provides an interaction channel between software and the CPU. You can obtain information about the CPU registers and memory contents, including other on-chip device information. In addition, program download and other operations can also be done through the debug interface.

The debug interface provides the following key features:

- Supports 5-wire JTAG mode;
- Supports multi-cluster debugging;
- Supports synchronous and asynchronous debug, enabling the CPU to enter the debug mode in extreme scenarios.
- Supports software breakpoints.
- Supports setting multiple memory breakpoints.
- Enables to check and set the values of CPU registers.
- Enables to check and modify memory values.
- Enables single-step and multi-step execution of instructions.
- Enables to quickly download programs.
- Enables to enter the debug mode after the reset of CPU.

Debug of C920 is coordinatedly implented by the debug software, debug proxy program, debugger, and debug interface. The location of the debug interface in CPU debug environment is shown in Fig. 12.1. The debug software

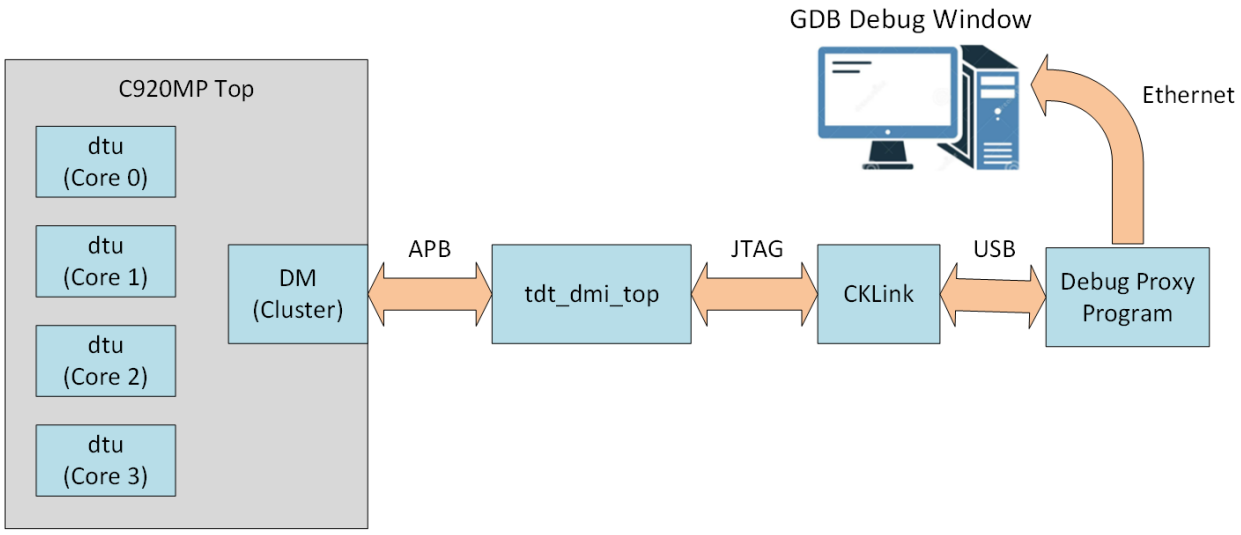


Fig. 12.1: **The Location of the Debug Interface in CPU Debug Environment**

is connected to the debug proxy program over network. The debug proxy program is connected to the debugger through USB. The debugger communicates with the debug interface of CPU in JTAG mode.

If a customer configures the SBA (System Bus Access) feature, an additional set of AXI buses will be present on the cluster side, enabling debug tools to bypass the CPU and directly access memory. The characteristics of this bus are as follows:

- Supports narrow transfers with a minimum size of 32 bits;
- Burst type is limited to increment only, with a fixed length of 0;
- resp only supports OKAY and SLVERR;
- Supports read-write outstanding capability of 1;
- Out-of-order execution is not supported;
- Interleaving is not supported;
- Unaligned accesses is not supported.

12.2 Configuration of Debug Resources

C920 provides 3 different options of debug resources for user convenience:

- Minimum configuration: 1 hardware breakpoint;
- Typical configuration: 3 hardware breakpoints;
- Maximum configurable: 8 hardware breakpoints and can form a trigger chain.

RISC-V Debug Protocol defines multi-function triggers to implement both breakpoints and watchpoints. The four types of triggers that can be configured are as follow:

- Instruction Address Type: Matches the instruction address, i.e., PC. This is similar to the traditional breakpoint functionality.
- Instruction Data Type: Matches the instruction code.
- Memory Access Address Type: Matches the memory access address of memory instruction. This type is similar to the traditional watchpoint feature.
- Memory Access Data Type: Matches the memory access data of memory instruction.

RISC-V Debug Protocol also defines 6 matching modes, which are orthogonal to the types mentioned above:

- Exact match: The trigger is triggered when the actual value of the CPU is equal to the trigger value.
- Low-bit mask match: It allows not comparing the lower bits. The trigger is triggered when the actual CPU value is equal to the trigger value.
- Greater than or equal to: The trigger is triggered when the actual CPU value is greater than or equal to the trigger value.
- Less than: The trigger is triggered when the actual CPU value is less than the trigger value.
- Low-bit mask match: The trigger value is divided into a high half [63:32] and a low half [31:0]. The high half of the trigger value is used as a mask, and the low half is used as a template. The trigger is triggered when trigger value[31:0] = CPU actual value bit[31:0] & trigger value[63:32].
- High-bit mask match: The trigger value is divided into a high half [63:32] and a low half [31:0]. The high half of the trigger value is used as a mask, and the low half is used as a template. The trigger is triggered when trigger value[31:0] = CPU actual value bit[63:32] & trigger value[63:32].

For detailed information, please refer to section 5.2.9 "Match Control" in the *RISC-V External Debug Support version 0.13.2* Documentation.

In addition to the above descriptions, each configuration combination supports debugging resources and methods such as software breakpoints, abstract command registers, asynchronous debugging, debugging after reset, and single-step instruction.

CHAPTER 13

Power Management

Since R1S4 version (also called as "1.4.x version"), C90 has improved significantly in power management features, supporting multiple power domains, power-down of a single core, cluster power-down, and clearing of the L2 cache by external hardware interface. This chapter describes the power management feature of C920 in detail.

13.1 Power Domain

C920 can be divided into up to 6 Power Domain as shown in Fig. 13.1.

- PD_CORE(x): Each core has its own separate power domain, and each core can be powered down independently.
- PD_TDT: Includes tdt_mp_top module and external tdt_dmi_top module, and PD_TDT and CPU power supplies are independent of each other.
  - Supports PD_TDT to remain powered up when cluster is powered down;
  - Supports powering down the PD_TDT to save power when the core is working normally;
- PD_CPU: Covers all parts inside the cluster except the above power domains.

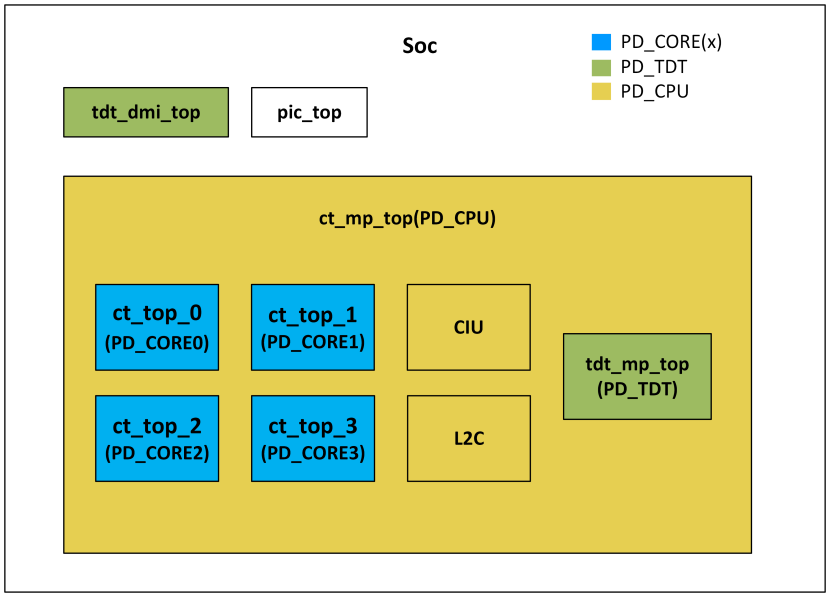


Fig. 13.1: **C920 Power Domain**

13.2 Overview of Low-power Modes

C920 supports the following low-power modes:

- Normal mode: All cores and L2 are running properly.
- Core WFI mode: Some cores are in the wait for interrupt (WFI) mode.
- Single-core power down: Some cores are powered down.
- Cluster power down: The entire cluster, including 4 cores and L2, are all powered down.

13.3 Core WFI Process

By executing the WFI low power instruction, a core enters WFI mode and outputs signal core(x)_pad_lpmd_b[1:0]=2' b00, which indicates that the core has entered WFI mode. At this moment, the L2 subsystem will disable the global Integrated Clock Gating (ICG) of this core inside the cluster.

The core will be woken up and exit WFI mode upon the occurrence of the following events:

- Reset;
- Interrupt requests: external interrupt, software interrupt, or timer interrupt requests sent by the Platform-Level Interrupt Controller (PLIC) or Core Local Interrupter (CLINT) submodules;
- Debug requests.

  When the following event occurs, the core is temporarily woken up to process the event. It reenters low power mode after the event is processed. But the core does not exit WFI mode during the entire process.
- Snoop request: Snoop requests sent by other cores.

13.4 Single-Core Power-Down Process

System can completely terminate the static power of the core by shutting down the core power. And the corresponding terminate process is as follows:

The power-down C920 core performs the following operations:

1. Notifies SoC that the single-core power-down process is to be executed. And the implementation of this step is subject to the SoC design.
2. Masks all interrupt requests of the core, including external interrupts, software interrupts, and timer interrupts, and then disables the interrupt enable bit (mie、sie) of the MSTATUS/SSTATUS register and the interrupt enable bit of the MIE/SIE register. If the power-down process is executed in Machine Mode (M-mode), disable the interrupt enable bits of the MSTATUS and MIE registers. If the power-down process is executed in Supervisor Mode (S-mode), disable the interrupt enable bits of the SSTATUS and SIE registers.
3. Disables data prefetch.
4. Executes D-Cache INV&CLR ALL and writes the dirty line back to the L2 cache.
5. Disables D-Cache (**Notes:** No store instruction is allowed between clearing the cache and disabling the cache).
6. Disables the SMPEN bit and mask snoop requests of the core.
7. Executes the sync.is instruction.
8. Executes the WFI instruction and enters WFI mode.

The system performs the following operations:

1. Detects a valid low-power output signal core(x)_pad_lpmd_b from the core.
2. Asserts pad_tdt_dm_core_unavail[x] to mask debug requests of the core to be powered down.
3. Activates the output signal clamp bit of the core to be powered down.
4. Deasserts the reset signal pad_core(x)_rst_b of the core to be powered down.
5. Powers down the core.

When the core is in a powered-off state, it can only be restarted by resetting it. The process of powering up the core is as follows:

1. The system detects a specific event and decides to power up (also referred to as "waking up") the core.
2. The system sets the reset address for the awakened core.
3. Asserts reset signal of the core.
4. Powers up and maintains the reset signal asserted.
5. Release the output signal clamp of the core.
6. Release the reset signal of the core.
7. The awakened core executes an initialization program, enables SMPEN bit and performs initialization operations such as enabling the MMU and DCACHE.

13.5 Cluster Power-Down Process (Hardware Clearing of the L2 Cache)

First, make sure that the power is shut down for all cores except the main core in the cluster. In this scenario, the "main core" refers to the last core to be powered down, which can be any of the 4 cores.

The main core performs the following operations:

1. Notifies SoC that the cluster power-down process is to be executed. The implementation is subject to the SoC design.
2. Masks all interrupt requests, including external interrupts, software interrupts, and timer interrupts, and then disables the interrupt enable bit (mie、sie) of the MSTATUS/SSTATUS register and the interrupt enable bit of the MIE/SIE register.
3. Disables data prefetch.
4. Executes the D-Cache INV&CLR ALL operation.
5. Disables D-Cache (**Notes:** No store instruction are allowed between clearing the cache and disabling the cache).
6. Disable the SMPEN bit for the core.
7. Execute the sync.is instruction.
8. Execute the low-power instruction WFI and enter low-power mode.

The system performs the following operations:

1. The system detects that the low power output signal core(x)_pad_lpmd_b of the main core is valid.
2. The system asserts pad_tdt_dm_core_unavail[x] to mask debug requests for the main core.
3. Activates the output signal clamp of the main core.
4. Deasserts the reset signal pad_core(x)_rst_b of the main core.
5. Powers down the main core.
6. Asserts pad_cpu_l2cache_flush_req to start the process of clearing the L2 cache.
7. Waits for C920 to return cpu_pad_l2cache_flush_done=1.
8. Deasserts pad_cpu_l2cache_flush_req. (Then C920 will deassert cpu_pad_l2cache_flush_done)
9. Ensures DCP (if configured) has no new requests.
10. Waits for C920 to return cpu_pad_no_op=1.
11. Activates the output signal clamp of the top-level.
12. Deasserts the L2 reset signal pad_cpu_rst_b.
13. Powers down the top-level.

The Cluster is powered up again through a reset process with the following steps:

1. Deasserts the reset signal of all cores and top-level in the Cluster.
2. Powers up, and maintains the reset signal asserted and pll stable.
3. Releases the output signal clamps of all cores and top-level.
4. Releases the reset signals of all cores and top-level.
5. Executes the reset exception service routine to restore the CPU state.

13.6 Simplified Scenario: Overall Cluster Power-Down Process (Hardware Clearing of the L2 cache)

In some systems, SoC designers may take a simplified way to divide power domains. That is, take the entire C920 cluster (4 cores + L2) as a power domain and power down the cluster as a whole, instead of distinguishing each single core. In this scenario, the power-down procedure (hardware clearing of the L2 cache) performs the following steps:

The system performs the following operations:

1. Notifies SoC that the overall cluster power-down process is to be executed. The implementation is subject to the SoC design.
2. Ensure that all existing transfers on the DCP (if any) are completed and no new read and write requests are sent to the DCP.

The core performs the following operations (There is no need to distinguish the main core and secondary core in this scenario, as the process is the same for them):

1. Masks all interrupt requests of the core, including external interrupts, software interrupts, and timer interrupts, and disables the interrupt enable bit (mie、sie) of the MSTATUS/SSTATUS register, as well as the interrupt enable bit of the MIE/SIE register.
2. Disables data prefetch.
3. Executes INV&CLR D-Cache ALL and writes dirty lines back to the L2 cache.
4. Disables D-Cache (No store instruction is allowed between clearing and disabling cache operations).
5. Disables the SMPEN bit and masks snoop requests for the core.
6. Executes the sync.is instruction.
7. Executes the WFI instruction.

The system performs the following operations:

1. Waits for all core(x)_pad_lpmd_b[1:0] == 2’ b00, indicating that all CPUs have entered low power state.
2. Asserts all pad_tdt_dm_core_unavail[x] to block debug requests.
3. Asserts pad_cpu_l2cache_flush_req to initiate the process of hardware clearing the L2 cache.
4. Waits for C920 to return cpu_pad_l2cache_flush_done=1, indicating that L2 cache has been cleared.
5. Deasserts pad_cpu_l2cache_flush_req. (Subsequently, C920 will deassert cpu_pad_l2cache_flush_done.)
6. Waits for cpu_pad_no_op==1’ b1, indicating that L2 is in idle state. (At this point, all CPUs are still in low power mode.)
7. Activates the output signal clamps of the cluster.
8. Asserts all reset signals.
9. Powers down the entire cluster.

13.7 Low-power Related Programming Models and Interface Signals

13.7.1 Changes in the Programming Model

Machine Mode (M-mode) Reset Registe (MRMR)

This register has been deleted. If this register is accessed, the read value will be zero and the write will be invalid, without reporting any exceptions. The impact of this change is that the reset signals of each core are no longer controlled by MRMR. The SoC can independently control the reset and de-reset of each core through pad_core(x)_rst_b.

M-mode Snoop Enable Register (MSMPR)

It is a newly added register with a width of 64 bits. Only bit[0] (SMPEN) is defined with a default value of 0. The feature of this register is to control whether the core can accept snoop requests.

- MSMPR.SMPEN = 0, the core cannot process snoop requests, and the top-level masks sending snoop requests to the core.
- MSMPR.SMPEN = 1, the core can process snoop requests, and the top-level sends snoop requests to the core.

Before powering down the core, it is required to set SMPEN=0 for the corresponding core. After powering up the core, software needs to set SMPEN=1 before enabling D-Cache and MMU. The core must keep SMPEN=1 in normal working mode.

M-mode Reset Vector Base Register (MRVBR)

The programming model has been modified. The implementation has changed from "4-core shared" to "core-private". The access permission has changed from "MRW" to "MRO". The initial values of MRVBR for each core are independent and determined by hardware signal pad_core(x)_rvba[39:1].

13.7.2 Interface Signals

The communication between C920 and the SoC power management unit is mainly achieved through the following signals:

- core(x)_pad_lpmd_b: Determine if a core is in WFI mode. 2' b11 represents normal mode, and 2' b00 represents WFI mode.
- cpu_pad_no_op: L2 Cache idle indication signal. When all cores enter low power mode and L2 Cache completes all transfers, this signal is valid (active high).
- pad_cpu_l2cache_flush_req and cpu_pad_l2cache_flush_done: This signal group is used to clear the L2 cache under the control of SoC, and the application scenario is the power-down of cluster. The "req" signal is driven by SoC, and the "done" signal is driven by C920. And the corresponding operation sequence is as follows: First, SoC asserts and maintains" req" to initiate the L2 clearing process; After C920 completes the L2 clearing, it returns "done" =1; SoC deasserts "req" ; then C920 deasserts "done" .

CHAPTER 14

Performance Monitoring Unit

14.1 PMU Overview

The performance monitoring unit (PMU) of C920 complies with the sstc and sscofpmf extensions of RISC-V standard. And the PMU is designed to collect software and partial hardware information during a program operation for software developers to optimize programs.

The software and hardware information collected by the PMU is classified as follows:

- Number of running clocks and the time (cycle, time)
- Instruction statistics (instret)
- Statistics of key components of the processor (hpmcounter3 to hpmcounter31, but hpmcounter19 to hpmcounter31 do not exist, and their corresponding control registers or control bits have not been implemented)

14.2 PMU Programming Model

14.2.1 Basic Features of PMU

Basic features of the PMU are as follow:

- Prohibits the counting of all events by the mcountinhibit register.
- Resets the current value of each PMU counters to 0, including mcycle, minstret, and mhpmcounter3 to mhpmcounter31.

- Configures the corresponding events for each PMU counter. In C920, each counter can be configured with any event. Write the event index value to the Performance Monitoring Event Select Register, and the counter will count the configured event normally. For example, writing 0x1 to mhpmevent5 indicates that mhpmcounter5 counts the number of L1 ICache accesses for event 0x1; while writing 0x2 to mhpmevent5 indicates that mhpmcounter5 counts the number of L1 ICache misses for event 0x2.
- Access authorization. The mcounteren register determines whether PMU counters can be accessed in Supervisor Mode (S-mode), and scounteren determines whether PMU counters can be accessed in User Mode (U-mode).
- Release disable status by the mcountinhibit register and starts counting.

For specific instances, please refer to *PMU Setting Instance*.

14.2.2 PMU Event Overflow Interrupt

The overflow interrupt initiated by PMU unit has a unified interrupt vector number of 13. The interrupt enablement and handling process are the same as regular private interrupts, and for detailed information, please refer to *Exception and Interrupt*.

14.3 PMU Related Control Register

14.3.1 Mcounteren Register

M-mode Counter Access Enable Register (mcounteren) is designed to authorize whether S-mode can access S-mode/U-mode counter.

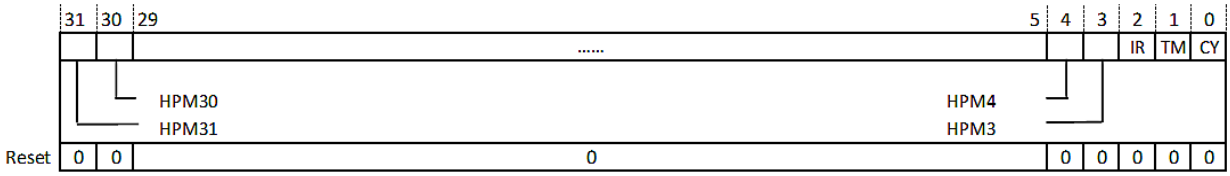


Fig. 14.1: **M-mode Counter Access Enable Register (mcounteren)**

Table 14.1: **Mcounteren Register**

Bit	Read/Write	Name	Description
31:3	Read/Write	HPM*n*	The access bit of the hpmcounter*n* / hpmcounter*n* register in S-mode. 0: An illegal instruction exception will occur for accesses to the hpmcounter*n* register in S-mode. 1: The hpmcounter*n* / hpmcounter*n* register can be normally accessed in S-mode.
2	Read/Write	IR	The access bit of the sintsret/instret register in S-mode. 0: An instruction exception will occur for accesses to the sintsret/instret register in S-mode. 1: The sintsret/instret register can be normally accessed in S-mode.

Continued on next page

Table 14.1 – continued from previous page

Bit	Read/Write	Name	Description
1	Read/Write	TM	The access bit of the time/stimecmp register in S-mode. 0: An illegal instruction exception will occur for accesses to the time/stimecmp register in S-mode. 1: When the corresponding bit of the mcounteren register is 1, the time/stimecmp register can be normally accessed in S-mode. Otherwise, an illegal instruction exception will occur.
0	Read/Write	CY	The access bit of the scycle/cycle register in S-mode. 0: An illegal instruction exception will occur for accesses to the scycle/cycle register in S-mode. 1: The scycle/cycle register can be normally accessed in S-mode.

14.3.2 Mcountinhibit Register

M-mode Count Inhibit Register (mcountinhibit) can prohibit M-mode counter from counting. Disabling counters in scenarios where performance analysis is not required reduces the power consumption of the processor.

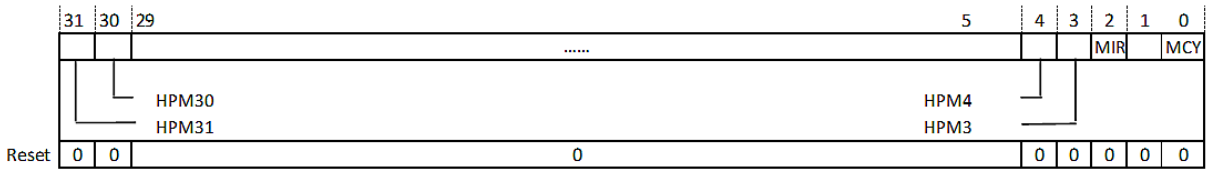


Fig. 14.2: **M-mode Count Inhibit Register (mcountinhibit)**

Table 14.2: **Mcountinhibit Register**

Bit	Read/Write	Name	Description
31:3	Read/Write	MHPMn	The count inhibit bit of the mhpmcounter$n$ register 0: normal counting 1: counting inhibited
2	Read/Write	MIR	The count inhibit bit of the minstret register 0: normal counting 1: counting inhibited
1	-	-	-
0	Read/Write	MCY	The count inhibit bit of the mcycle register 0: normal counting 1: counting inhibited

14.3.3 MHPMCR Register

M-mode Performance Monitor Control Register (MHPMCR) is Xuantie self-extending register.

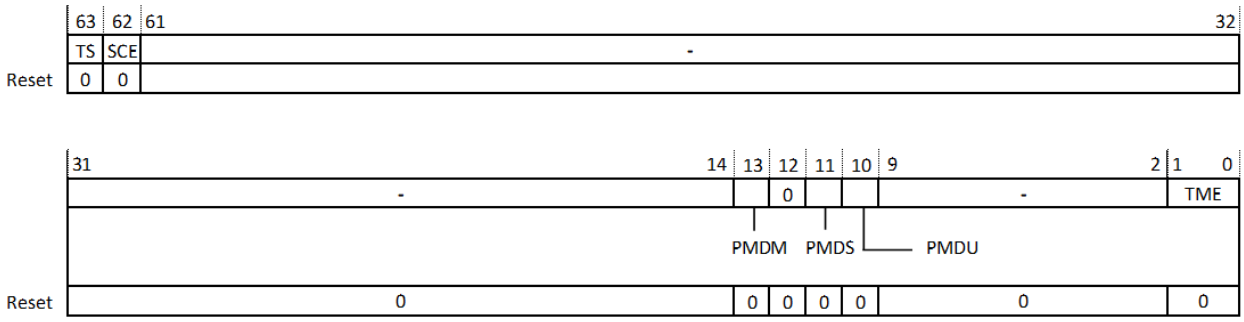


Fig. 14.3: **M-mode Performance Monitor Control Register (MHPMCR)**

Table 14.3: **MHPMCR Register**

Bit	Read/Write	Name	Description
63	Read/Write	TS	Trigger status bit, indicating whether performance monitoring is in trigger state. It is not recommended to manually modify this bit. 1' b0: Not in trigger state 1' b1: In trigger state
62	Read/Write	SCE	S-mode control enable bit: 1' b0: Read and write shpmcr in S-mode, and the trigger register will trigger an illegal instruction 1' b1: Read and write enable shpmcr in S-mode and the trigger register can be read and written normally.
13	Read/Write	PMDM	Performance monitor disable machine mode counting: Stop M-mode counting control bit. And this bit is mapped in mxstatus. 1' b0: M-mode counting is normal 1' b1: Disable M-mode counting
11	Read/Write	PMDS	Performance monitor disable supervisor mode counting: Stop S-mode counting control bit. And this bit is mapped in mxstatus. 1' b0: S-mode counting is normal 1' b1: Disable S-mode counting
1: 0	Read/Write	TME	Trigger Mode enable bit: 2' b00: Trigger mode is not enabled, and normal counting 2' b01: Trigger/Stop trigger mode is enabled. When the program address matches the start trigger (mhpmsp) register and the trigger is enabled, the trigger is activated and the event counter starts counting. When the program address matches the end trigger (mhpmep) register, the trigger is disabled and the counting is stopped. 2' b10: Start/End trigger enabled. When the program address falls within the range of flags indicated by the start trigger register and end trigger, the event counter counts normally. Otherwise, no counting is performed. 2' b11: Reserved state, meaningless, counting will not take place.

14.3.4 Mcounterwen Register

The M-mode Counter Write Enable Register (MCOUNTERWEN) is designed to authorize whether the S-mode can write the S-mode event counters.

Note:

Both the corresponding bits of MCOUNTEREN and MCOUNTERWEN must be enabled at the same time, to achieve the permission to write to the corresponding event counters in M-mode; When the corresponding bits of MCOUNTEREN are not enabled, enabling the corresponding bits of MCOUNTERWEN only still can not achieve write permission.

This register is an extended register in M-mode with a bit length of 64 bits. The read and write permissions of this register are allowed in M-mode only, and accessing it in any mode other than M-mode will occur an illegal instruction exception.

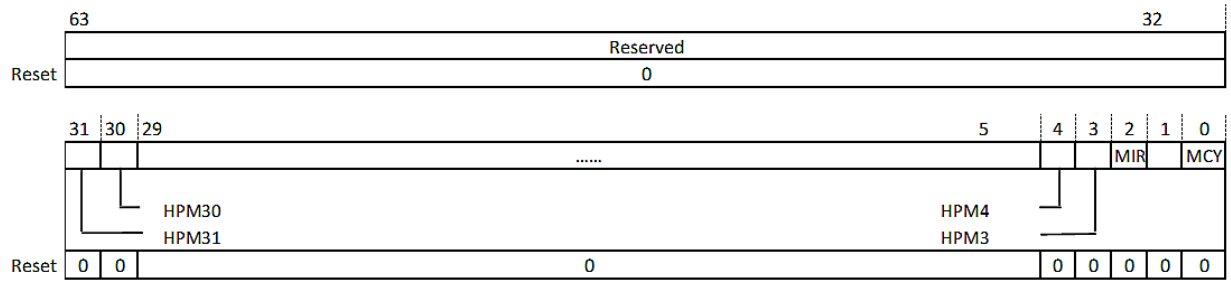


Fig. 14.4: **M-mode Counter Write Enable Register (MCOUNTERWEN)**

Table 14.4: **MCOUNTERWEN Register**

Bit	Read/Write	Name	Description
31:3	Read/Write	HPM*n*	The write enable bit of the scounter*n* register in S-mode: When HPM*n* is set to 1, it indicates that scounter*n* is not writable in S-mode, writing to it will result in an illegal exception. When HPM*n* is set to 0, it indicates that scounter*n* is writable in S-mode .
2	Read/Write	MIR	The write enable bit of the sinst register in S-mode: 0: Instruction counter is not writable in S-mode, and writing to it will cause an illegal exception 0: Instruction counter is writable in S-mode
1	-	-	-
0	Read/Write	MCY	The write enable bit of the scycle register in S-mode: 0: The corresponding counter is not writable in S-mode, and writing to it will cause an illegal exception 0: The corresponding counter is writable in S-mode

14.3.5 Scounteren Register

The S-mode Counter Access Enable Register (scounteren) is designed to authorize whether U-mode can access U-mode couner.

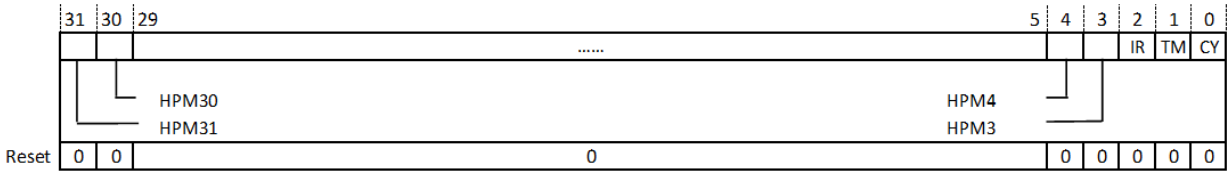


Fig. 14.5: **S-mode Counter Access Enable Register (SCOUNTERE)**

Table 14.5: **SCOUNTERE Register**

Bit	Read/Write	Name	Description
31:3	Read/Write	HPMn	The access bit of the hpmcounter$n$ register in U-mode. 0: An illegal instruction exception will occur for accesses to the hpmcounter$n$ register in U-mode. 1: When the corresponding bit of the mcounteren and scounteren registers both are 1, the hpmcounter$n$ register can be normally accessed in U-mode. Otherwise, an illegal instruction exception will occur.
2	Read/Write	IR	The access bit of the instret register in U-mode. 0: An illegal instruction exception will occur for accesses to the instret register in U-mode. 1: When the corresponding bit of the mcounteren and scounteren registers both are 1, the instret register canbe normally accessed in U-mode. Otherwise, an illegal instruction exception will occur.
1	Read/Write	TM	The access bit of the time register in U-mode. 0: An illegal instruction exception will occur for accesses to the time register in U-mode. 1: When the corresponding bit of the mcounteren and scounteren registers both are 1, the time register can be normally accessed in U-mode. Otherwise, an illegal instruction exception will occur.
0	Read/Write	CY	The access bit of the cycle register in U-mode. 0: An illegal instruction exception will occur for accesses to the cycle register in U-mode. 1: When the corresponding bit of the mcounteren and scounteren registers both are 1, the cycle register can be normally accessed in U-mode. Otherwise, an illegal instruction exception will occur.

14.3.6 SHPMINHIBIT Register

S-mode Count Inhibit Register (SHPMINHIBIT) is applied to inhibit counter counting in S-mode. It disables counters in scenarios where performance analysis is not needed, thereby reducing processor power consumption.

When mcounterwen.bit[n] is set to 1, shpminhibit [n] becomes readable and writable in S-mode.

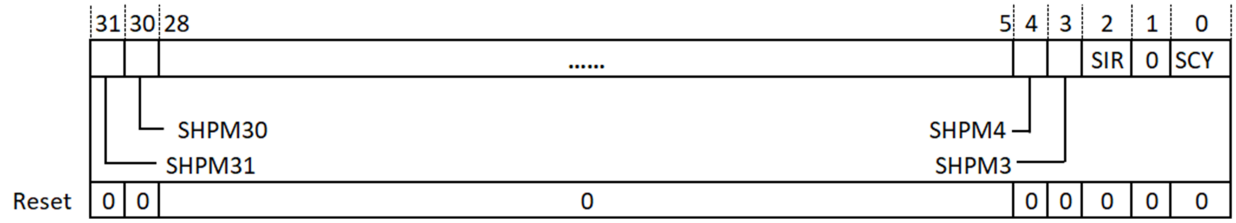


Fig. 14.6: **S-mode Count Inhibit Register (SHPMINHIBIT)**

Table 14.6: **SHPMINHIBIT Register**

Bit	Read/Write	Name	Description
31:3	Read/Write	SHPMn	The count inhibit bit of shpmcountern register: 0: normal counting 1: count inhibited
2	Read/Write	SIR	The count inhibit bit of sinstret register: 0: normal counting 1: count inhibited
1	-	-	-
0	Read/Write	SCY	The count inhibit bit of scycle register: 0: normal counting 1: count inhibited

14.3.7 SHPMCR Register

S-mode Performance Monitor Control Register (SHPMCR) is the Xuantie self-extending register. And it' s the read/write mapping of mhpmcr, excluding sce. When in S32, only TS is mapped to BIT[31].

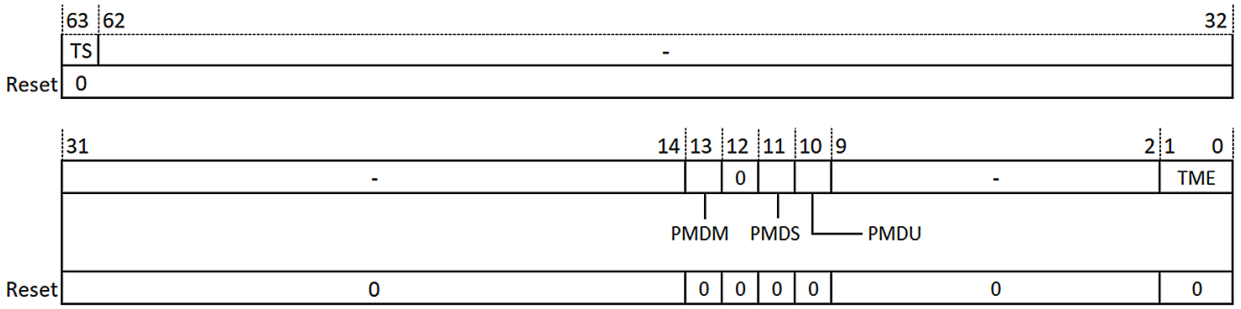


Fig. 14.7: **S-mode Performance Monitor Control Register (SHPMCR)**

14.3.8 STIMECMP Register

S-mode Time Comparison Register (STIMECMP) belongs to ssct extension.

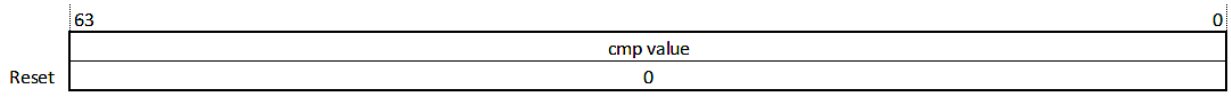


Fig. 14.8: **S-mode Time Comparison Register (STIMECMP)**

STIMECMP Register Description:

- Reset value is 64' hffff_ffff_ffff_ffff.
- When menvcfg.STCE is 0, accessing the stimecmp csr generates an illegal instruction exception, and even if the time value is greater than the stimecmp csr, it will not generate the stimer interrupt.
- When STCE is 1, the stimecmp csr serves as the source of stip generation.
- When STCE is 0, the stimecmp in the memory map serves as the source of stip generation.

14.3.9 SCOUNTOVF Register

S-mode Counter Interrupt Overflow Register belongs to sscofpmf extension.

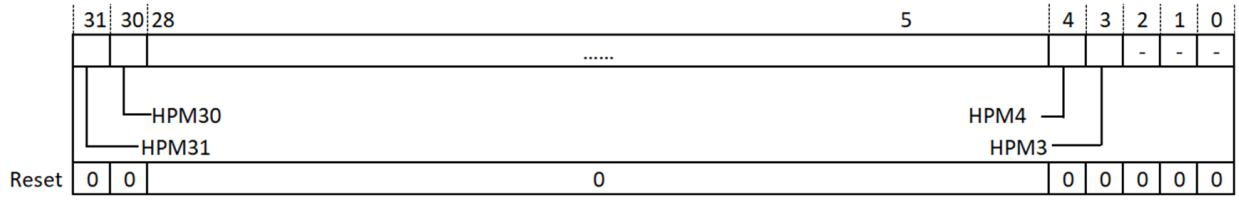


Fig. 14.9: **S-mode Counter Interrupt Overflow Register (SCOUNTOVF)**

Table 14.7: **SCOUNTOVF Register**

Bit	Read/Write	Name	Description
31:3	Read/Write	HPMn	The overflow flg bit of the shpmcounter$n$ register: 1' b0: No overflow occurs in shpmcounter$n$ 1' b0: Overflow occurs in shpmcounter$n$
2	-	-	-
1	-	-	-
0	-	-	-

- Each bit of the scountovf register represents the read-only mapping of each event selector [63] OF bit;
- When mcounteren[i] = 1, scountovf can be read normally in M-mode and S-mode, otherwise, the read value is 0.

14.4 M-mode Performance Monitor Event Select Register

M-mode Performance Monitor Event Select Register (mhpmevent3-31) is designed to selects the counting event corresponding to a counter. In C920, each counter can be configured with any event. The counter can count the configured event normally by writing the event index value into the mhpmevent3-31 Register.

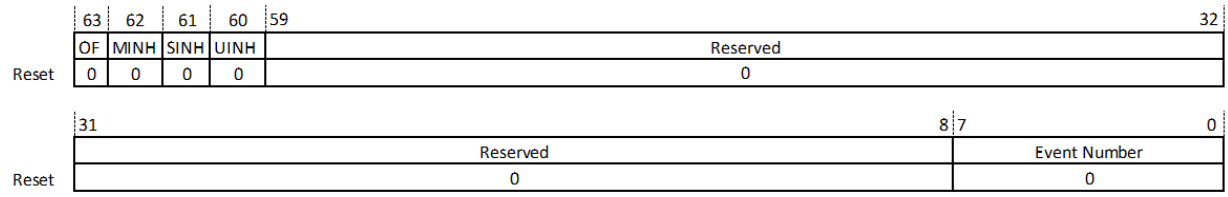


Fig. 14.10: **M-mode Performance Monitor Event Select Register (mhpmevent3-31)**

Table 14.8 is the detailed information for M-mode Performance Monitor Event Select Register.

Table 14.8: **mhpmevent3-31 Register**

Bit	Read/Write	Name	Description
63	Read/Write	OF	The newly extended performance count overflow flg bit of the sscofpmf register: When the CP0 write initial value is 0, the corresponding performance event counter will be set to 1 when it overflows, generating an overflow interrupt; During this time, the counter will continue to wrap and count, but no new overflow interrupts will be generated until CP0 is rewritten by software. When the CP0 write initial value is 1, the value of the corresponding performance event counter will remain unchanged when it overflows, and no overflow interrupt will be generated. The counter will continue to wrap and count until it is rewritten by software.
62	MRW	MINH	The newly extended disable counting bit for newly added performance counter in M-mode.
61	MRW	SINH	The newly extended disable counting bit for newly added performance counter in S-mode.
60	MRW	UINH	The newly extended disable counting bit for newly added performance counter in u-mode.
59~8	MRW	Reserved	Extended Reserved in sscofpmf register
7~0	MRW	Event Number	Event index number: If EVENT_NUMBER is 0, the event counter n is disabled and does not count; If EVENT_NUMBER is non-zero, the event counter n is enabled and counts the events corresponding to the event ID normally.

The correspondence between event selectors, events, and counters in Table 14.9 .

Table 14.9: **Counter Event Mapping List**

Index	Event
0x1	L1 ICache Access Counter
0x2	L1 ICache Miss Counter
0x3	I-UTLB Miss Counter
0x4	D-UTLB Miss Counter
0x5	JTLB Miss Counter
0x6	Conditional Branch Mispredict Counter
0x7	Conditional Branch Counter
0x8	Indirect Branch Mispredict Counter
0x9	Indirect Branch Counter
0xA	LSU Spec Fail Counter
0xB	Store Instruction Counter
0xC	L1 DCache load access Counter
0xD	L1 DCache load miss Counter
0xE	L1 DCache store access Counter
0xF	L1 DCache store miss Counter
0x10	L2 load access Counter
0x11	L2 load miss Counter
0x12	L2 store access Counter
0x13	L2 store miss Counter
0x14	RF Launch Fail Counter
0x15	RF Reg Launch Fail Counter
0x16	RF Instruction Counter
0x17	LSU Cross 4K Stall Counter
0x18	LSU Other Stall Counter
0x19	LSU SQ Discard Counter
0x1A	LSU SQ Data Discard Counter
0x1B	IFU Branch Target Mispred Counter
0x1C	IFU Branch Target Instruction Counter
0x1D	ALU Instruction Counter
0x1E	LDST Instruction Counter
0x1F	Vector SIMD Instruction Counter
0x20	CSR Instruction Counter
0x21	Sync Instruction Counter
0x22	LDST Unaligned Access Counter
0x23	Interupt Number Counter
0x24	Interrupt Off Cycle Counter
0x25	Environment Call Counter
0x26	Long Jump Counter
0x27	Stalled Cycles Frontend Counter
0x28	Stalled Cycles Backend Counter
0x29	Sync Stall Counter

Continued on next page

Table 14.9 – continued from previous page

Index	Event
0x2A	Floating Point Instruction Counter
>= 0x2B	Currently undefined

14.5 Event Counters

Event counters are divided into three groups: M-mode event counters, c920 extended S-mode event counters, and U-mode event counters.

M-mode event counters are shown in Table 14.10.

Table 14.10: **M-mode Event Counter List**

Name	Index	Read/Write	Initial Value	Description
MCYCLE	0xB00	MRW	0x0	cycle counter
MINSTRET	0xB02	MRW	0x0	instructions-retired counter
MHPMCOUNTER3	0xB03	MRW	0x0	performance-monitoring counter
MHPMCOUNTER4	0xB04	MRW	0x0	performance-monitoring counter
...	...	...	...	...
MHPMCOUNTER31	0xB1F	MRW	0x0	performance-monitoring counter

C920 extended s-mode event counters are listed in Table 14.11.

Table 14.11: **C920 Extended S-mode Event Counters List**

Name	Index	Read/Write	Initial Value	Description
SCYCLE	0x5E0	SRO	0x0	cycle counter
SINSTRET	0x5E2	SRO	0x0	instructions-retired counter
SHPMCOUNTER3	0x5E3	SRO	0x0	performance-monitoring counter
SHPMCOUNTER4	0x5E4	SRO	0x0	performance-monitoring counter
...	...	...	...	...
SHPMCOUNTER31	0x5FF	SRO	0x0	performance-monitoring counter

U-mode event counters are listed in Table 14.12.

Table 14.12: **U-mode Event Counters List**

Name	Index	Read/Write	Initial Value	Description
CYCLE	0xC00	URO	0x0	cycle counter
TIME	0xC01	URO	0x0	timer
INSTRET	0xC02	URO	0x0	instructions-retired counter
HPMCOUNTER3	0xC03	URO	0x0	performance-monitoring counter
HPMCOUNTER4	0xC04	URO	0x0	performance-monitoring counter

Continued on next page

Table 14.12 – continued from previous page

Name	Index	Read/Write	Initial Value	Description
...	...	...	...	...
HPMCOUNTER31	0xC1F	URO	0x0	performance-monitoring counter

CYCLE, INSTRET and HPMCOUNTERn counters in U-mode are read-only mappings of the corresponding M-mode event counters. The TIME counter is the read-only mapping of the MTIME register; SCYCLE, SINSTRET, and SHPMCOUNTERn counters in S-mode are mappings of corresponding M-mode event counters.

14.6 Trigger Register

Trigger registers in C920 are shown in the following table:

Table 14.13: **Trigger Register List**

Name	Index	Read/Write	Initial Value	Description
MHPMSR	0x7f1	MRW	0x0	M-mode Start Trigger Register
MHPMER	0x7f2	MRW	0x0	M-mode End Trigger Register
SHPMSR	0x5ca	SRW	0x0	S-mode Start Trigger Register
SHPMER	0x5cb	SRW	0x0	S-mode End Trigger Register

The organization forms of trigger registers are shown in the following pictures:

14.6.1 Start Trigger Register

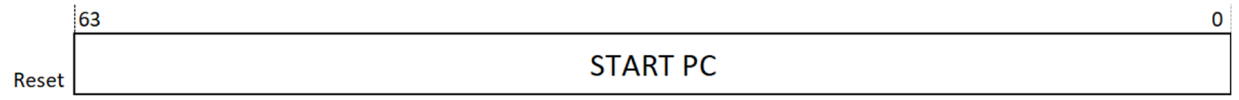


Fig. 14.11: **Start Trigger Register**

14.6.2 End Trigger Register

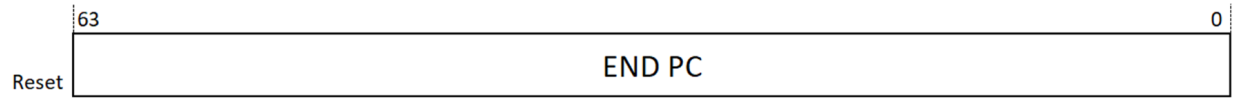


Fig. 14.12: **End Trigger Register**

Based on different trigger modes, Start Trigger Register and End Trigger jointly provide the user-specified trigger function:

TRIGGER/STOP Mode:

The user specifies the starting point of the counter for statistics through the start trigger register, and the endpoint of the counter for statistics through the stop trigger register. When the MHPMCR.TME is enabled, once the program

reaches the PC specified by the start trigger register, the counter starts counting until the program reaches the endpoint specified by the stop trigger register to end the statistics. Users can use the trigger function to achieve statistics on specific threads, features, and so on.

START/END Mode:

When the program' s PC is within the range specified by the start trigger register and the stop trigger register, the counters count normally. However, once the retirement address of an instruction is beyond this range, all counters stop counting.

Compared to the TRIGGER/STOP mode, the START/END mode has stricter conditions for counting. If there is a subroutine call or any other situation that causes the instruction to run beyond the START and END range within the program segment, it will cause the counters to stop.

In addition, the access of S-mode trigger register is controlled by the mhpmcr.SCE bit. When the SCE bit is set to 1, the trigger registers can be read and written normally. Otherwise, it will occur an illegal instruction.

CHAPTER 15

Program Instances

This chapter mainly describes multiple program examples, including Memory Management Unit (MMU) setup, Physical Memory Protection (PMP) setup, cache setup, multi-core startup, synchronization primitive, Platform Level Interrupt Controller (PLIC) setup, PMU setup and Performance Monitoring Unit (PMU) setup.

15.1 Optimal CPU Performance Configuration

The optimal performance of C920 can be achieved by the following configurations:

- MHCR = 0x11FF
- MHINT = 0x31EA32C
- MCCR2 = 0xE249000B (**Note:** Mccr2 contains RAM latency setting, and users need to set the suitable RAM latency based on the actual situation.)
- MXSTATUS = 0x638000
- MSMPR = 0x1

```
# mhcr
  li x3, 0x11ff
  csrs mhcr,x3

#mhint
li x3, 0x31ea32c
csrs mhint,x3
```

(continues on next page)

(continued from previous page)

```
# mxstatus
li   x3, 0x638000
csrs mxstatus,x3

# msmpr
csrsi msmpr,0x1

# mccr2
li x3, 0xe249000b
csrs mccr2,x3
```

15.2 MMU Setting Instance

```
/*****************************************************************************

* Function: An example of setting C920MP MMU.
* Memory space: Virtual address <-> physical address.
*
* Pagesize 4K: vpn: {vpn2,vpn1,vpn0} <-> ppn: {ppn2,ppn1,ppn0}
* Pagesize 2M: vpn: {vpn2,vpn1} <-> ppn:{ppn2,ppn1}
* Pagesize 1G: vpn: {vnp2} <-> ppn: {ppn2}
*
*****************************************************************************/

  /*C920 will invalidate all MMU TLB entries automatically when reset*/
  /*You can use sfence.vma to invalid all MMU TLB entries if necessary*/
  sfence.vma x0, x0

  /* Pagesize 4K: vpn: {vpn2, vpn1, vpn0} <-> ppn: {ppn2, ppn1, ppn0}*/
  /* First-level page addr base: PPN (defined in satp)*/
  /* Second-level page addr base: BASE2 (self define)*/
  /* Third-level page addr base: BASE3 (self define)*/
  /* 1. Get first-level page addr base: PPN and vpn*/
  /* Get PPN*/
  csrr x3, satp
  li x4, 0xfffffffffff
  and x3, x3, x4

  /*2. Config first-level page*/
  /*First-level page addr: {PPN, vpn2, 3'b0}, first-level page pte:{ 44'b BASE2, 10'b1} */
  /*Get first-level page addr*/
```

(continues on next page)

(continued from previous page)

```
slli x3, x3, 12
/*Get vpn2*/
li x4, VPN
li x5, 0x7fc0000
and x4, x4, x5
srli x4, x4, 15
and x5, x3, x4
/*Store pte at first-level page addr*/
li x6, {44'b BASE2, 10'b1}
sd x6, 0(x5)

/*3. Config second-level page*/
/*Second-level page addr: {BASE2, vpn1, 3'b0}, second-level page pte:{ 44'b BASE3, 10'b1} */
/*Get second-level page addr*/
/* VPN1*/
li x4, VPN
li x5, 0x3fe00
and x4, x4, x5
srli x4, x4, 9
/*BASE2*/
li x5, BASE2
srli x5, x5, 12
and x5, x5, x4
/*Store pte at second-level page addr*
li x6, {44'b BASE3, 10'b1}
sd x6, 0(x5)
/*4. Config third-level page*/
/*Third-level page addr: {BASE3, vpn0, 3'b0}, third-level page pte:{
theadflag, ppn2, ppn1, ppn0, 9'b flags,1'b1} */
/*Get second-level page addr*/
/* VPN0*/
li x4, VPN
li x5, 0x1ff
and x4, x4, x5
srli x4, x4, 3
/*BASE3*/
li x5, BASE3
srli x5, x5, 12
and x5, x5, x4
/*Store pte at second-level page addr*/
li x6, { theadflag, ppn2, ppn1, ppn0, 9'b flags, 1'b1}
sd x6, 0(x5)
```

(continues on next page)

(continued from previous page)

```
/* Pagesize 2M: vpn: {vpn2, vpn1} <-> ppn: {ppn2, ppn1}*/
/*First-level page addr base: PPN (defined in satp)*/
/*Second-level page addr base: BASE2 (self define)*/

/*1. Get first-level page addr base: PPN and vpn*/
/* Get PPN*/
csrr x3, satp
li x4, 0xfffffffffff
and x3, x3, x4

/*2. Config first-level page*/
/*First-level page addr: {PPN, vpn2, 3'b0}, first-level page pte:{ 44'b
BASE2, 10'b1}*/
/*Get first-level page addr*/
slli x3, x3, 12
/*Get vpn2*/
li x4, VPN
li x5, 0x7fc0000
and x4, x4, x5
srli x4, x4, 15
and x5, x3, x4
/*Store pte at first-level page addr*/
li x6, {44'b BASE2, 10'b1}
sd x6, 0(x5)

/*3. Config second-level page*/
/*Second-level page addr: {BASE2, vpn1, 3'b0}, second-level page pte:{
theadflag, ppn2, ppn1, 9'b0, 9'b flags,1'b1} */
/*Get second-level page addr*/
/*VPN1*/
li x4, VPN
li x5, 0x3fe00
and x4, x4, x5
srli x4, x4, 9
/*BASE2*/
li x5, BASE2
srli x5, x5, 12
and x5, x5, x4
/*Store pte at second-level page addr*/
li x6, { theadflag, ppn2, ppn1, 9'b0, 9'b flags,1'b1}
sd x6, 0(x5)
```

(continues on next page)

(continued from previous page)

```
  /* Pagesize 1G: vpn: {vpn2} <-> ppn: {ppn2}*/
  /*First-level page addr base: PPN (defined in satp)*/
  /*1. Get first-level page addr base: PPN and vpn*/
  /* Get PPN*/
  csrr x3, satp
  li x4, 0xfffffffffff
  and x3, x3, x4

  /*2. Config first-level page*/
  /*First-level page addr: {PPN, vpn2, 3'b0}, first-level page pte:{
  theadflag, ppn2, 9'b0, 9'b0, 9'b flags,1'b1}*/
  /*Get first-level page addr*/
  slli x3, x3, 12
  /*Get vpn2*/
  li x4, VPN
  li x5, 0x7fc0000
  and x4, x4, x5
  srli x4, x4, 15
  and x5, x3, x4
  /*Store pte at first-level page addr*/
  li x6, { theadflag, ppn2, 9'b0, 9'b0, 9'b flags,1'b1}
  sd x6, 0(x5)
```

15.3 PMP Setting Instance

```
/*****************************************************************************
* Function: An instance of setting C920MP PMP.
* 0x0 ~ 0xf0000000, TOR Mode, RWX
* 0xf0000000 ~ 0xf8000000, NAPOT Mode, RW
*0xfff73000 ~ 0xfff74000, NAPOT Mode, RW
*0xfffc0000 ~ 0xfffc2000, NAPOT Mode, RW
*The above four regions are configured with different execution permissions.
In addition, it is necessary to configure the PMP accordingly,to prevent the CPU from␣
↪speculatively executing into unsupported address regions, especially in the machine mode (M-
↪mode) that has default full execution permissions.
Specifically, after configuring the address regions that require execution permissions, the␣
↪remaining address regions should be configured with no permissions, as shown in the following␣
↪instance.
*****************************************************************************/

  # pmpaddr0,0x0 ~ 0xf0000000, TOR Mode, read and write and execution permissions
```

(continues on next page)

(continued from previous page)

```
  li x3, (0xf0000000 >> 2)
  csrw pmpaddr0, x3
  # pmpaddr1,0xf0000000 ~ 0xf8000000, NAPOT Mode, read and write permissions
  li x3, ( 0xf0000000 >> 2 | (0x8000000-1) >> 3))
  csrw pmpaddr1, x3
  # pmpaddr2,0xfff73000 ~ 0xfff74000, NAPOT Mode, read and write permissions
  li x3, ( 0xfff73000 >> 2 | (0x1000-1) >> 3))
  csrw pmpaddr2, x3
  # pmpaddr3,0xfffc0000 ~ 0xfffc2000, NAPOT Mode, read and write permissions
  li x3, ( 0xfffc0000 >> 2 | (0x2000-1) >> 3))
  csrw pmpaddr3, x3
  # pmpaddr4,0xf0000000 ~ 0x100000000, NAPOT Mode, no permission
  li x3, ( 0xf0000000 >> 2 | (0x10000000-1) >> 3))
  csrw pmpaddr4, x3
  # pmpaddr5,0x100000000 ~ 0xfffffffffff, TOR Mode, no permission
  li x3, (0xfffffffffff >> 2)
  csrw pmpaddr5, x3
  # PMPCFG0, configure each table entry execution permissions/modes/lock bits.
  When the lock bit is set to 1, the table entry is only effective in M-mode.
  li x3,0x88989b9b9b8f
  csrw pmpcfg0, x3
  # pmpaddr5,0x100000000 ~ 0xfffffffffff, TOR Mode, 0x100000000 <= addr < 0xfffffffffff,
  pmpaddr5 will always be hit.
  However, pmpaddr5 cannot be hit in the address range 0xfffffff000 ~ 0xfffffffffff (the minimum␣
↪PMP granularity is 4 KB in C920).
  If it is required to mask the last 4K space of the 1T space, another table entry in NAPOT mode␣
↪needs to be configured.
```

15.4 Cache Instance

15.4.1 Cache Enabling Instance

```
/*C920 will invalidate all I-cache automatically when reset*/
/*You can invalidate I-cache by yourself if necessary*/
/*Invalidate I-cache*/
li x3, 0x33
csrc mcor, x3
li x3, 0x11
csrs mcor, x3
// You can also use icache instrucitons to replace the invalidate sequence
// if theadisaee is enabled.
```

(continues on next page)

(continued from previous page)

```
//icache.iall
//sync.is

/*Enable I-cache*/
li x3, 0x1
csrs mhcr, x3

/*C920 will invalidate all D-cache automatically when reset*/
/*You can invalidate D-cache by yourself if necessary*/
/*Invalidate D-cache*/
li x3, 0x33
csrc mcor, x3
li x3, 0x12
csrs mcor, x3

// You can also use dcache instrucitons to replace the invalidate sequence
// if theadisaee is enabled.
// dcache.iall
// sync.is

/*Enable D-cache*/
li x3, 0x2
csrs mhcr, x3
```

15.4.2 Synchronization Instance between Instruction and Data Caches

CPU0

```
sd x3,0(x4) // a new instruction defined in x3
            // is stored to program memory address defined in x4.
dcache.cval1 r0 // clean the new instrcution to the shared L2 cache.
sync.s          // ensure completion of clean operation.
               // the dcache clean is not necessarily if INSDE is not enabled.
icache.iva r0  // invalid icache according to shareable configuraiton.
sync.s/fence.i  // ensure completion in all CPUs.
sd x5,0(x6)    // set flag to signal operation completion.
sync.is
jr x4 // jmp to new code
```

CPU1~CPU3

```
WAIT_FINISH:
  ld x7,0(x6)
```

(continues on next page)

(continued from previous page)

```
bne x7,x5, WAIT_FINISH // wait CPU0 modification finish.
sync.is
jr x4                  // jmp to new code
```

15.4.3 Synchronization Instance between TLB and Data Cache

CPU0

```
sd x4,0(x3) // update a new translation table entry
sync.is/fence.i // ensure completion of update operation.
sfence.vma x5,x0 // invalid the TLB by va
sync.is/fence.i // ensure completion of TLB invalidation and
                // synchronises context
```

15.4.4 L2 Cache Partitioning Feature Configuration

Step 1: Set MCCR2.PAE=1, to enable partion.

Step 2: Configure the ML2WP register to set the available group placement for each ID. This register is shared among all cores. For example, pid0 is only allowed to be placed in group0, and pid1 is only allowed to be placed in group1, which is shown as follows:

ML2WP[63:56]=8' h80

ML2WP[55:48]=8' h40

...

ML2WP[7:0]=8' h01

Note:

Each id needs to enable at least one group. Otherwise, the register will indicate that the id is enabled for every group after the assignment.

Step 3: Configure ML2PID for each core, which represents the Process ID (PID) of the current core, a simple configuration method is as follows:

For Core 0: Set ML2PID[2:0] = 0

For Core 1: Set ML2PID[2:0] = 1

So forth

The last step: Configure MXSTATUS[9] (i.e., SPCE bit) to determine whether to enable SL2WP and SL2PID.

15.5 Multi-core Startup Instance

Note:

This section is outdated! Version 1.4.x: mrmr has been deleted, and mrvcr has become private, Machine Read Only (MRO).

CPU0

```
......          // CPU1~CPU3 are in reset mode and
                // CPU0 executes system initialize operation.
li x3, RVBA
csrw mrvbr, x3 // Set reset vector base addrress
li x3, 0x2
csrs mrmr, x3 // Release CPU1' s reset signal.
li x3, 0x4
csrs mrmr, x3 // Release CPU2' s reset signal.
li x3, 0x8
csrs mrmr, x3 // Release CPU3' s reset signal.
              // CPU1~CPU3 start to execute reset exception routine.
```

15.6 Synchronization Primitive Instance

CPU0

```
li x1, 0x1
li x6, 0x0

ACQUIRE_LOCK:               // (x3) is the lock address. 0: Free; 1: Busy.
lr x4, 0(x3)                // Read lock
bnez x4, ACQUIRE_LOCK       // Try again if the lock is in use
sc x5, x1, 0(x3)            // Attempt to store new value
bne x6, x5, ACQUIRE_LOCK    // Try again if fail
sync.s

...                         // Critical section code
```

CPU1

```
sync.s/fence.i      // Ensure all operations are observed before clearing the lock.
sd x0, 0(x3)        // Clear the lock.
```


15.7 PLIC Setting Instance

```
//Init id 1 machine mode int for hart 0
/*1.set hart threshold if needed*/
li x3, (plic_base_addr + 0x200000) // h0 mthreshold addr
li x4, 0xa //threshold value
sw x4,0x0(x3) // set hart0 threshold as 0xa

/*2.set priority for int id 1*/
li x3, (plic_base_addr + 0x0) // int id 1 prio addr
li x4, 0x1f // prio value
sw x4,0x4(x3) // init id1 priority as 0x1f

/*3.enable m-mode int id1 to hart*/
li x3, (plic_base_addr + 0x2000) // h0 mie0 addr
li x4, 0x2
sw x4,0x0(x3) // enable int id1 to hart0

/*4.set ip or wait external int*/
/*following code set ip*/
li x3, (plic_base_addr + 0x1000) // h0 mthreshold addr
li x4, 0x2 // id 1 pending
sw x4, 0x0(x3) // set int id1 pending

/*5.core enters interrupt handler, read PLIC_CLAIM and get ID*/

/*6.core takes interrupt*/

/*7.core needs to clear external interrupt source if LEVEL(not PULSE)
configured, then core writes ID to PLIC_CLAIM and exits interrupt*/
```

15.8 PMU Setting Instance

```
/*1.inhibit counters counting*/
li x3, 0xffffffff
csrw mcountinhibit, x3

/*2.C920 will initial all pmu counters when reset*/
/*you can initial pmu counters manually if necessarily*/
csrw mcycle, x0
csrw minstret, x0
csrw mhpmcounter3, x0
```

(continues on next page)

(continued from previous page)

```
……
csrw mhpmcounter31, x0

/*3.configure mhpmevent*/
li x3, 0x1
csrw mhpmevent3, x3 // mhpmcounter3 count event: L1 ICache Access Counter
li x3, 0x2
csrw mhpmevent4, x3 // mhpmcounter4 count event: L1 ICache Miss Counter
……
li x3, 0x13
csrw mhpmevent21, x3 // mhpmcounter21 count event: L2 Cache write miss Counter

/*4. configure mcounteren and scounteren*/
li x3, 0xffffffff
csrw mcounteren, x3 // enable super mode to read hpmcounter
li x3, 0xffffffff
csrw scounteren, x3 // enable user mode to read hpmcounter

/*5. enable counters to count when you want*/
csrw mcountinhibit, x0
```


CHAPTER 16

Appendix A Standard Instructions

C920MP implements the RV64IMAFCV instruction set architecture. And the following sections provide specific descriptions of each instruction according to the different instruction sets.

16.1 Appendix A-1 I Instructions

This section describes the RISC-V I instructions implemented by C920 in detail. And the instructions are listed in alphabetic order.

The instructions are 32-bit wide by default. However, in specific cases, the system assembles some instructions into 16-bit compressed instructions. For more information about compressed instructions, please refer to *Appendix A-6 C Instructions*.

16.1.1 ADD——The Signed Add Instruction

Syntax:

add rd, rs1, rs2

Operation:

rd ← rs1 + rs2

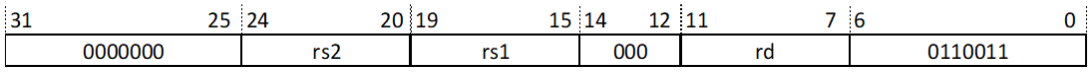
Execute Permission:

Machine Mode (M-mode)/Supervisor Mode (S-mode)/User-mode (U-mode)

Exception:

None

Instruction format:



16.1.2 ADDI——The Signed Immediate Add Instruction

Syntax:

addi rd, rs1, imm12

Operation:

rd ← rs1 + sign_extend(imm12)

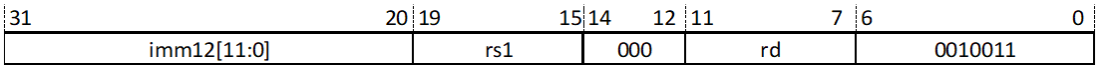
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.3 ADDIW——The Signed Immediate Add Instruction for the Lower 32 Bits

Syntax:

addiw rd, rs1, imm12

Operation:

tmp[31:0] ← rs1[31:0] + sign_extend(imm12)[31:0]

rd ← sign_extend(tmp[31:0])

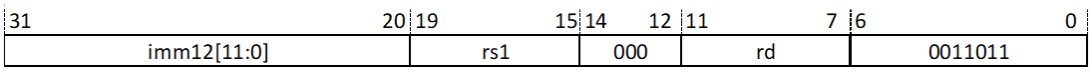
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.4 ADDW——The Signed Add Instruction for the Lower 32 Bits

Syntax:

addw rd, rs1, rs2

Operation:

tmp[31:0] ← rs1[31:0] + rs2[31:0]

rd ← sign_extend(tmp[31:0])

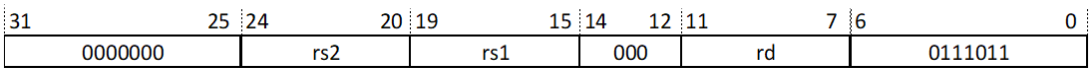
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.5 AND——The Bitwise AND Instruction

Syntax:

and rd, rs1, rs2

Operation:

rd ← rs1 & rs2

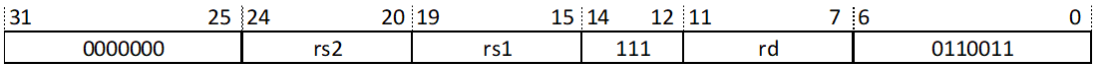
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.6 ANDI——The Immediate Bitwise AND Instruction

Syntax:

andi rd, rs1, imm12

Operation:

rd ← rs1 & sign_extend(imm12)

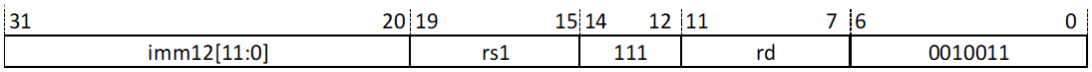
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.7 AUIPC——The Add Upper Immediate to PC Instruction

Syntax:

auipc rd, imm20

Operation:

rd ← current pc + sign_extend(imm20<<12)

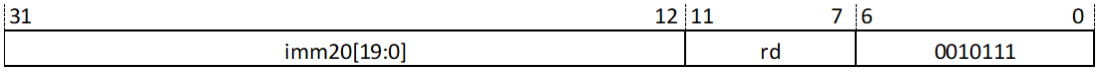
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.8 BEQ——The Branch-if-equal Instruction

Syntax:

beq rs1, rs2, label

Operation:

if (rs1 == rs2)

next pc = current pc +sign_extend(imm12<<1)

else

next pc = current pc + 4

Execute Permission:

M-mode/S-mode/U-mode

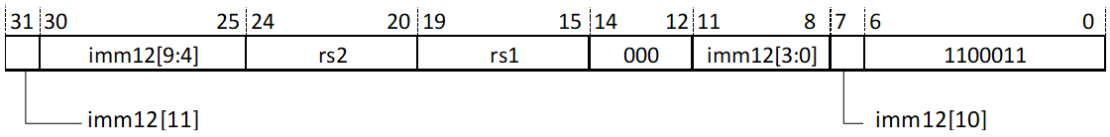
Exception:

None

Note:

- The assembler calculates imm12 based on the label.
- The instruction jump range is ±4KB address space.

Instruction format:



16.1.9 BGE——The Signed Branch-if-greater-than-or-equal Instruction

Syntax:

bge rs1, rs2, label

Operation:

if (rs1 >= rs2)

next pc = current pc + sign_extend(imm12 <<1)

else

next pc = current pc + 4

Execute Permission:

M-mode/S-mode/U-mode

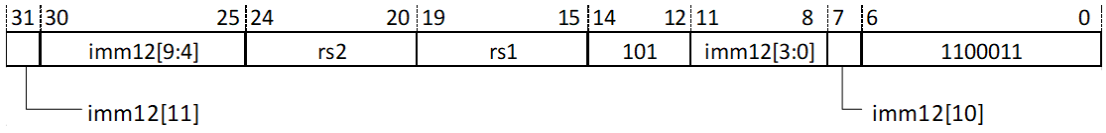
Exception:

None

Note:

- The assembler calculates imm12 based on the label.
- The instruction jump range is ±4KB address space.

Instruction format:



16.1.10 BGEU——The Unsigned Branch-if-greater-than-or-equal instruction

Syntax:

bgeu rs1, rs2, label

Operation:

if (rs1 >= rs2)

next pc = current pc + sign_extend(imm12<<1)

else

next pc = current pc + 4

Execute Permission:

M-mode/S-mode/U-mode

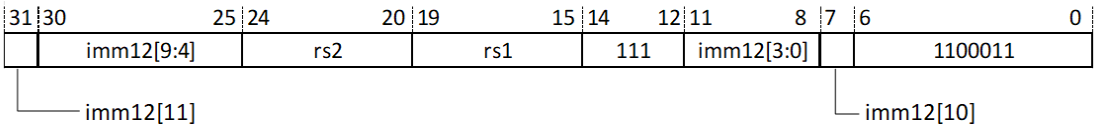
Exception:

None

Note:

- The assembler calculates imm12 based on the label.
- The instruction jump range is ±4KB address space.

Instruction format:



16.1.11 BLT——The Signed Branch-if-less-than Instruction

Syntax:

blt rs1, rs2, label

Operation:

if (rs1 < rs2)

next pc = current pc + sign_extend(imm12<<1)

else

next pc = current pc + 4

Execute Permission:

M-mode/S-mode/U-mode

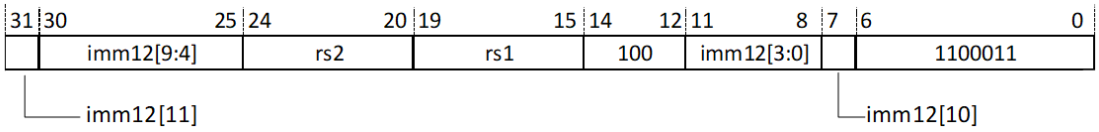
Exception:

None

Note:

- The assembler calculates imm12 based on the label.
- The instruction jump range is ±4KB address space.

Instruction format:



16.1.12 BLTU——The Unsigned Branch-if-less-than Instruction

Syntax:

bltu rs1, rs2, label

Operation:

if (rs1 < rs2)

next pc = current pc + sign_extend(imm12<<1)

else

next pc = current pc + 4

Execute Permission:

M-mode/S-mode/U-mode

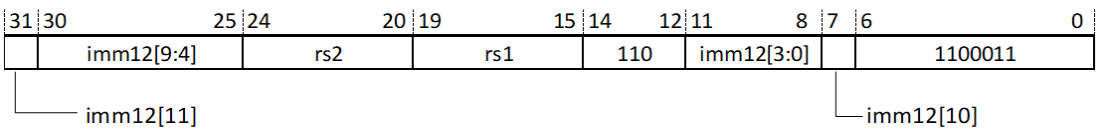
Exception:

None

Note:

- The assembler calculates imm12 based on the label.
- The instruction jump range is ±4KB address space.

Instruction format:



16.1.13 BNE——The Branch-if-not-equal Instruction

Syntax:

bne rs1, rs2, label

Operation:

if (rs1 != rs2)

next pc = current pc + sign_extend(imm12<<1)

else

next pc = current pc + 4

Execute Permission:

M-mode/S-mode/U-mode

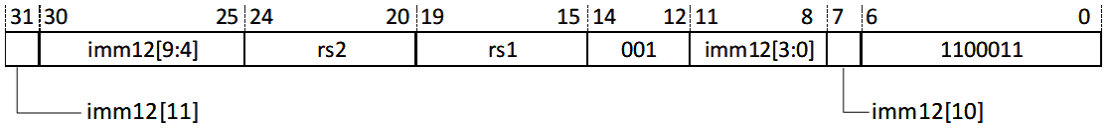
Exception:

None

Note:

- The assembler calculates imm12 based on the label.
- The instruction jump range is ±4KB address space.

Instruction format:



16.1.14 CSRRC——The Control and Status Register Read/Clear Instruction

Syntax:

csrrc rd, csr, rs1

Operation:

rd ← csr

csr ← csr & (~rs1)

Execute Permission:

M-mode/S-mode/U-mode

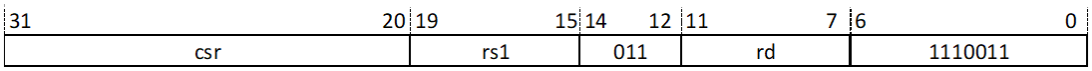
Exception:

The illegal instruction exception

Note:

- The Control and Status Register (CSR) that can be accessed vary depending on the permission levels. Please refer to the CSR chapter for specific details.
- When rs1 = x0, this instruction does not generate a write operation or cause any exceptions related to write behavior.

Instruction format:



16.1.15 CSRRCI——The CSR Read/Clear Immediate Instruction

Syntax:

csrrci rd, csr, imm5

Operation:

rd ← csr

csr ← csr & ~zero_extend(imm5)

Execute Permission:

M-mode/S-mode/U-mode

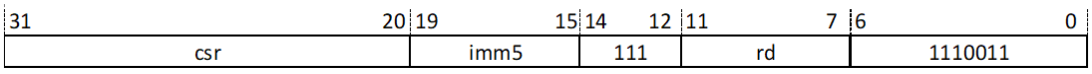
Exception:

The illegal instruction exception

Note:

- The CSRs can be accessed vary depending on the permission levels. Please refer to the CSR chapter for specific details.
- When rs1=x0, this instruction does not generate a write operation or cause any exceptions related to write behavior.

Instruction format:



16.1.16 CSRRS——The CSR Read/Set Instruction

Syntax:

csrrs rd, csr, rs1

Operation:

rd ← csr

csr ← csr | rs1

Execute Permission:

M-mode/S-mode/U-mode

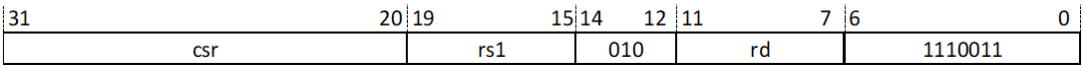
Exception:

The illegal instruction exception

Note:

- The CSRs can be accessed vary depending on the permission levels. Please refer to the CSR chapter for specific details.
- When rs1=x0, this instruction does not generate a write operation or cause any exceptions related to write behavior.

Instruction format:



16.1.17 CSRRSI——The CSR Read/Set Immediate Instruction

Syntax:

csrrsi rd, csr, imm5

Operation:

rd ← csr

csr ← csr | zero_extend(imm5)

Execute Permission:

M-mode/S-mode/U-mode

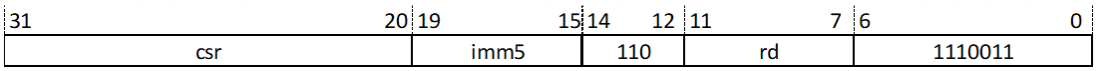
Exception:

The illegal instruction exception

Note:

- The CSRs can be accessed vary depending on the permission levels. Please refer to the CSR chapter for specific details.
- When rs1=x0, this instruction does not generate a write operation or cause any exceptions related to write behavior.

Instruction format:



16.1.18 CSRRW——The CSR Read/Write Instruction

Syntax:

csrrw rd, csr, rs1

Operation:

rd ← csr

csr ← rs1

Execute Permission:

M-mode/S-mode/U-mode

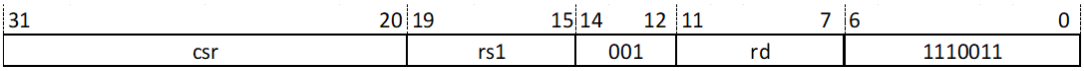
Exception:

The illegal instruction exception

Note:

- The CSRs can be accessed vary depending on the permission levels. Please refer to the CSR chapter for specific details.
- When rs1=x0, this instruction does not generate a write operation or cause any exceptions related to write behavior.

Instruction format:



16.1.19 CSRRWI——The CSR Read/Write Immediate Instruction

Syntax:

csrrwi rd, csr, imm5

Operation:

rd ← csr

csr[4:0] ← imm5

csr[63:5] ← 0

Execute Permission:

M-mode/S-mode/U-mode

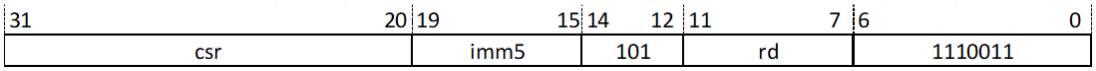
Exception:

The illegal instruction exception

Note:

- The CSRs can be accessed vary depending on the permission levels. Please refer to the CSR section for specific details.
- When rs1=x0, this instruction does not generate a write operation or cause any exceptions related to write behavior.

Instruction format:



16.1.20 EBREAK——The Breakpoint Instruction

Syntax:

ebreak

Operation:

Generates breakpoint exceptions or enters the debug mode.

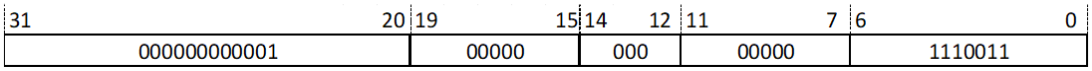
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The breakpoint exception

Instruction format:



16.1.21 ECALL——The Environment Call Instruction

Syntax:

ecall

Operation:

Generates environmental exceptions.

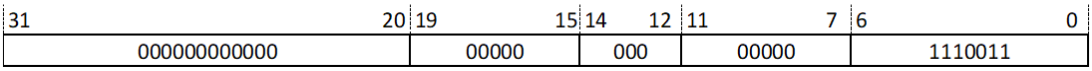
Execute Permission:

M-mode/S-mode/U-mode

Exception:

U-mode, S-mode, M-mode environment call exceptions

Instruction format:



16.1.22 FENCE——The Memory Synchronization Instruction

Syntax:

fence iorw, iorw

Operation:

Ensures that all memory or device read/write instructions before this instruction are observed earlier than those after this instruction.

Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Notes:

When pi=1, so=1, and the instruction syntax is fence i,o, and so forth.

Instruction format:

31 28	27	26	25	24	23	22	21	20	19 15	14 12	11 7	6 0
0000	pi	po	pr	pw	si	so	sr	sw	00000	000	00000	0001111

16.1.23 FENCE.I——The Instruction Stream Synchronization Instruction

Syntax:

fence.i

Operation:

Clears the Instruction Cache (I-Cache) to ensure that all the data access results before this instruction can be accessed by the instruction’ s subsequent fetch operation.

Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:

31 28	27 24	23 20	19 15	14 12	11 7	6 0
0000	0000	0000	00000	001	00000	0001111

16.1.24 JAL——The Instruction for Directly Jumping to a Subroutine

Syntax:

jal rd, label

Operation:

next pc ← current pc + sign_extend(imm20<<1)

rd ← currect pc + 4

Execute Permission:

M-mode/S-mode/U-mode

Exception:

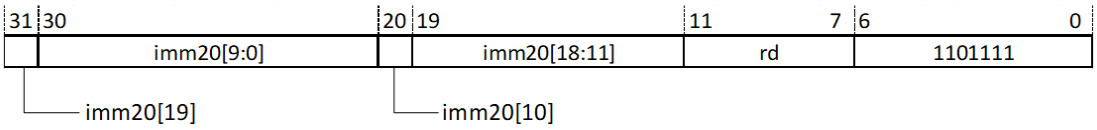
None

Notes:

The assembler calculates imm20 based on the label.

The jump range of the instruction is ±1 MB address space.

Instruction format:



16.1.25 JALR——The Jump and Link Register Instruction

Syntax:

jalr rd, rs1, imm12

Operation:

next pc ← (rs1 + sign_extend(imm12)) & 64' hfffffffffffffe

rd ← currect pc + 4

Execute Permission:

M-mode/S-mode/U-mode

Exception:

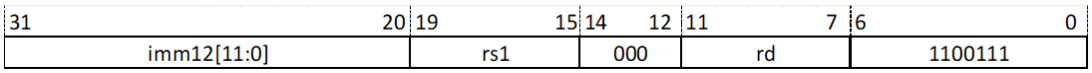
None

Notes:

- The jump range of the instruction is the entire 1 TB address space, when M-mode or the Memory Management Unit (MMU) is disabled.

- The jump range of the instruction is the entire 512 GB address space, when none-machine mode and the MMU are enabled.

Instruction format:



16.1.26 LB——The Signed Extended Byte Load Instruction

Syntax:

lb rd, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

rd ← sign_extend(mem[address])

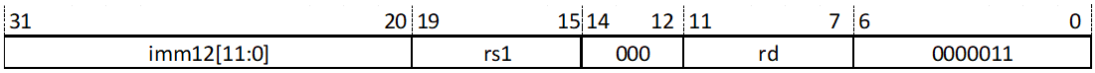
Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



16.1.27 LBU——The unsigned Extended Byte Load Instruction

Syntax:

lbu rd, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

rd ← zero_extend(mem[address])

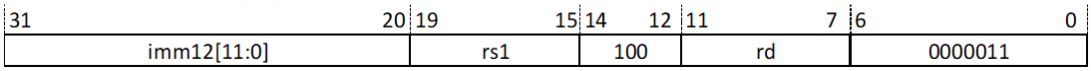
Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



16.1.28 LD——The Doubleword Load Instruction

Syntax:

ld rd, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

rd ← mem[(address+7):address]

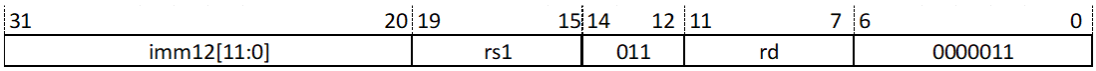
Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



16.1.29 LH——The Signed Extended Halfword Load Instruction

Syntax:

lh rd, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

rd ← sign_extend(mem[(address+1):address])

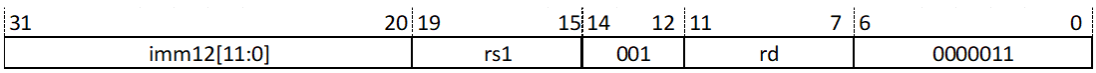
Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



16.1.30 LHU——The Unsigned Extended Halfword Load Instruction

Syntax:

lhu rd, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

rd ← zero_extend(mem[(address+1):address])

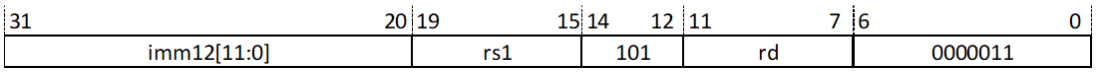
Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



16.1.31 LUI——The Upper Immediate Load Instruction

Syntax:

lui rd, imm20

Operation:

rd←sign_extend(imm20<<12)

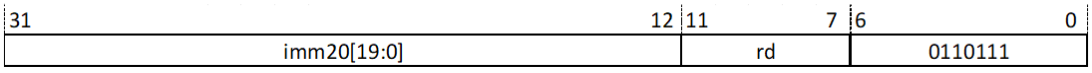
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.32 LW——The Signed Extended Word Load Instruction

Syntax:

lw rd, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

rd ← sign_extend(mem[(address+3):address])

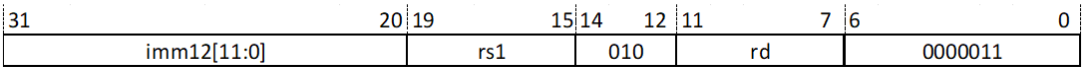
Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



31 20	19 15	14 12	11 7	6 0
imm12[11:0]	rs1	010	rd	0000011

16.1.33 LWU——The Unsigned Extended Word Load Instruction

Syntax:

lwu rd, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

rd ← zero_extend(mem[(address+3):address])

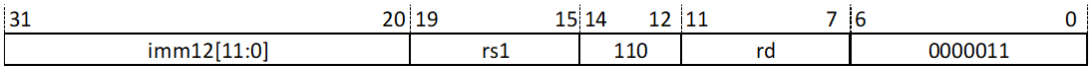
Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



31 20	19 15	14 12	11 7	6 0
imm12[11:0]	rs1	110	rd	0000011

16.1.34 MRET——The Exception Return Instruction in M-mode

Syntax:

mret

Operation:

next pc← mepc

mstatus.mie ←mstatus.mpie

mstatus.mpie ←1

Execute Permission:

M-mode

Exception:

The illegal instruction exception

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0011000	00010	00000	000	00000	1110011

16.1.35 OR——The Bitwise OR Instruction

Syntax:

or rd, rs1, rs2

Operation:

rd ← rs1 | rs2

Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	110	rd	0110011

16.1.36 ORI——The Immediate Bitwise OR Instruction

Syntax:

ori rd, rs1, imm12

Operation:

rd ← rs1 | sign_extend(imm12)

Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:

31 20	19 15	14 12	11 7	6 0
imm12[11:0]	rs1	110	rd	0010011

16.1.37 SB——The Byte Store Instruction

Syntax:

sb rs2, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

mem[:address] ← rs2[7:0]

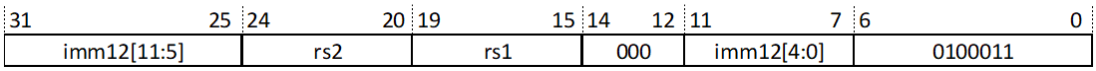
Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
imm12[11:5]	rs2	rs1	000	imm12[4:0]	0100011

16.1.38 SD——The Doubleword Store Instruction

Syntax:

sd rs2, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

mem[(address+7):address] ← rs2

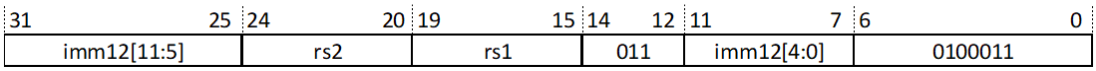
Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
imm12[11:5]	rs2	rs1	011	imm12[4:0]	0100011

16.1.39 SFENCE.VMA——The Virtual Memory Synchronization Instruction

Syntax:

sfence.vma rs1,rs2

Operation:

Invalidation and synchronization operations of virtual memory

Execute Permission:

M-mode/S-mode

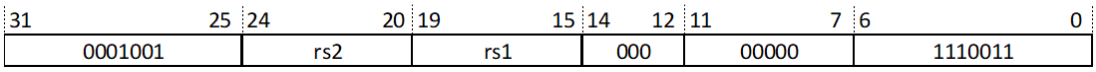
Exception:

The illegal instruction exception

Notes:

- mstatus.tvm=1, running this instruction in S-mode will trigger an illegal instruction exception.
- rs1: the virtual address, rs2: the Address Space Identifier (ASID).
  - rs1=x0, rs2=x0, all TLB entries are invalidated.
  - rs1!=x0, rs2=x0, all TLB entries that hit the virtual address specified by rs1 are invalidated.
  - rs1=x0, rs2!=x0, all TLB entries that hit the process ID specified by rs2 are invalidated.
  - rs1!=x0, rs2!=x0, all TLB entries that hit the virtual address specified by rs1 and the process ID specified by rs2 are invalidated.

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
0001001	rs2	rs1	000	00000	1110011

16.1.40 SH——The Halfword Store Instruction

Syntax:

sh rs2, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

mem[(address+1):address] ← rs2[15:0]

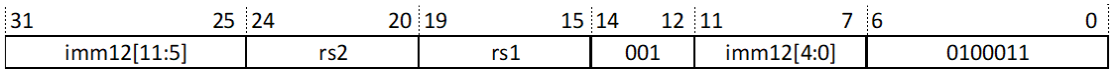
Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
imm12[11:5]	rs2	rs1	001	imm12[4:0]	0100011

16.1.41 SLL——The Logical Left Shift instruction

Syntax:

sll rd, rs1, rs2

Operation:

rd← rs1 << rs2[5:0]

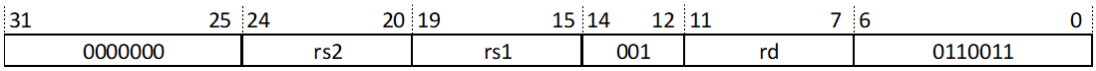
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.42 SLLI——The Immediate Logical Left Shift Instruction

Syntax:

slli rd, rs1, shamt6

Operation:

rd← rs1 << shamt6

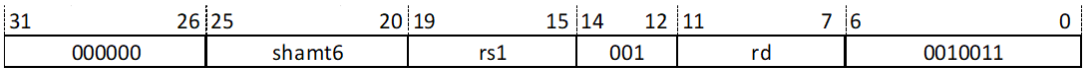
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.43 SLLIW——The Immediate Logical Left Shift Instruction on the Lower 32 Bits

Syntax:

slliw rd, rs1, shamt5

Operation:

tmp[31:0]←(rs1[31:0] << shamt5)[31:0]

rd← sign_extend(tmp[31:0])

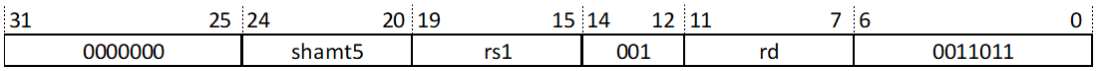
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.44 SLLW——The Logical Left Shift Instruction on the Lower 32 Bits

Syntax:

sllw rd, rs1, rs2

Operation:

tmp[31:0]← (rs1[31:0] << rs2[4:0])[31:0]

rd←sign_extend(tmp[31:0])

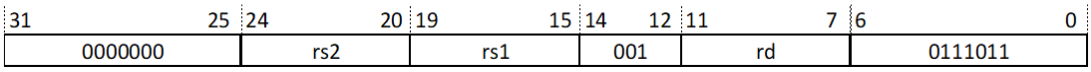
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.45 SLT——The Signed Set-If-Less-than Instruction

Syntax:

slt rd, rs1, rs2

Operation:

if (rs1 < rs2)

rd←1

else

rd←0

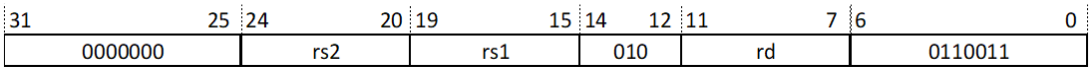
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.46 SLTI——The Signed Set-If-less-than-Immediate Instruction

Syntax:

slti rd, rs1, imm12

Operation:

if (rs1 <sign_extend(imm12))

rd←1

else

rd←0

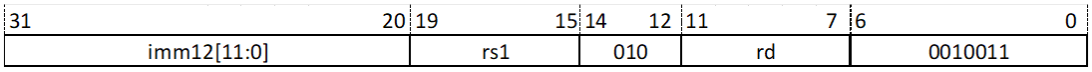
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.47 SLTIU——The Unsigned Set-If-less-than-Immediate Instruction

Syntax:

sltiu rd, rs1, imm12

Operation:

if (rs1 < sign_extend(imm12))

rd←1

else

rd←0

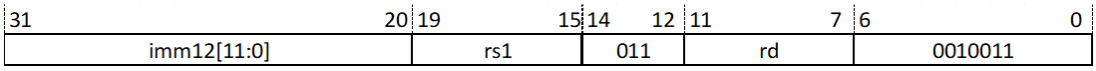
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.48 SLTU——The Unsigned Set-If-less-than Instruction

Syntax:

sltu rd, rs1, rs2

Operation:

if (rs1 < rs2)

rd←1

else

rd←0

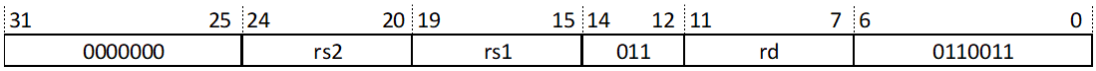
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.49 SRA——The Arithmetic Right Shift Instruction

Syntax:

sra rd, rs1, rs2

Operation:

rd←rs1 >>> rs2[5:0]

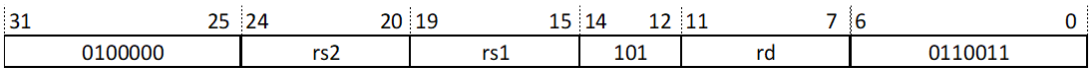
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.50 SRAI——The Immediate Arithmetic Right Shift Instruction

Syntax:

srai rd, rs1, shamt6

Operation:

rd← rs1 >>>shamt6

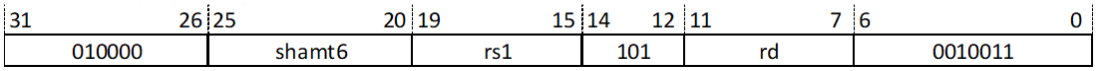
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.51 SRAIW——The Immediate Arithmetic Right Shift Instruction on the Lower 32 Bits

Syntax:

sraiw rd, rs1, shamt5

Operation:

tmp[31:0]←(rs1[31:0] >>> shamt5)[31:0]

rd← sign_extend(tmp[31:0])

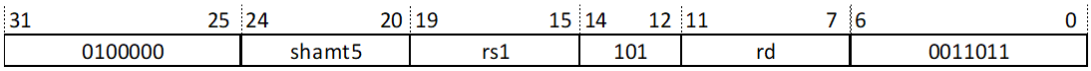
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.52 SRAW——The Arithmetic Right Shift Instruction on the Lower 32 Bits

Syntax:

sraw rd, rs1, rs2

Operation:

tmp←(rs1[31:0] >>> rs2[4:0])[31:0]

rd←sign_extend(tmp)

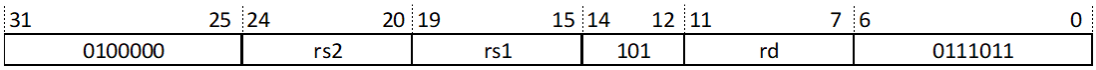
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.53 SRET——The Exception Return Instruction in S-mode

Syntax:

sret

Operation:

next pc← sepc

sstatus.sie ←sstatus.spie

sstatus.spie ←1

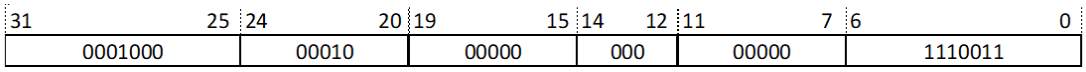
Execute Permission:

S-mode

Exception:

The illegal instruction exception

Instruction format:



16.1.54 SRL——The Logical Right Shift Instruction

Syntax:

srl rd, rs1, rs2

Operation:

rd←rs1 >> rs2[5:0]

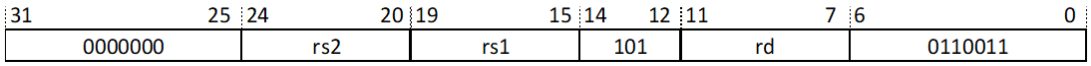
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.55 SRLI——The Immediate Logical Right Shift Instruction

Syntax:

srli rd, rs1, shamt6

Operation:

rd← rs1 >> shamt6

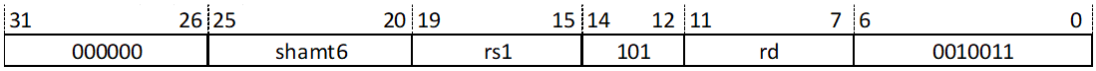
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.56 SRLIW——The Immediate Logical Right Shift Instruction on the Lower 32 Bits

Syntax:

srliw rd, rs1, shamt5

Operation:

tmp[31:0]←(rs1[31:0] >> shamt5)[31:0]

rd← sign_extend(tmp[31:0])

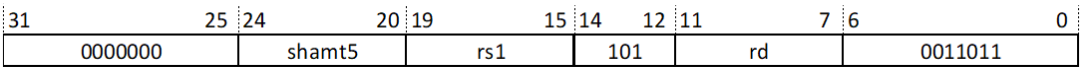
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.57 SRLW——The Logical Right Shift Instruction on the Lower 32 Bits

Syntax:

srlw rd, rs1, rs2

Operation:

tmp←(rs1[31:0] >> rs2[4:0])[31:0]

rd←sign_extend(tmp)

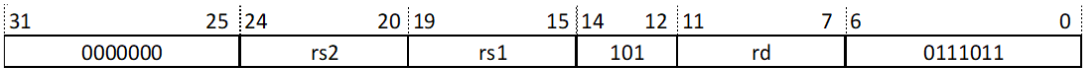
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.58 SUB——The Signed Subtract Instruction

Syntax:

sub rd, rs1, rs2

Operation:

rd ← rs1 - rs2

Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0110011

16.1.59 SUBW——The Signed Subtract Instruction on the Lower 32 Bits

Syntax:

subw rd, rs1, rs2

Operation:

tmp[31:0] ← rs1[31:0] - rs2[31:0]

rd ← sign_extend(tmp[31:0])

Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0111011

16.1.60 SW——The Word Store Instruction

Syntax:

sw rs2, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

mem[(address+3):address] ← rs2[31:0]

Execute Permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
imm12[11:5]	rs2	rs1	010	imm12[4:0]	0100011

16.1.61 WFI——The Instruction for Entering the Low Power Mode

Syntax:

wfi

Operation:

The processor enters a low-power mode, during which the CPU clock is disabled and most peripheral clocks are also disabled.

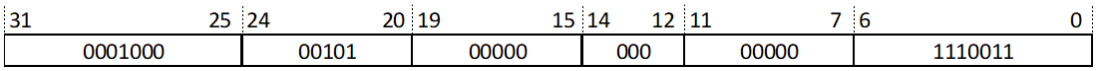
Execute Permission:

M-mode/S-mode

Exception:

None

Instruction format:



16.1.62 XOR——The Bitwise XOR Instruction

Syntax:

xor rd, rs1, rs2

Operation:

rd ← rs1 ⌒ rs2

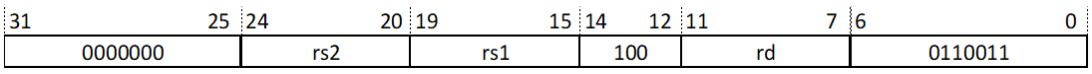
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.1.63 XORI——The Immediate Bitwise XOR Instruction

Syntax:

xori rd, rs1, imm12

Operation:

rd ← rs1 & sign_extend(imm12)

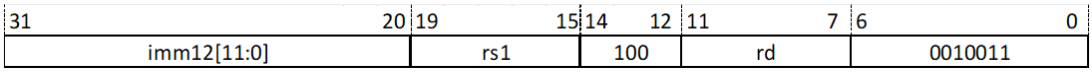
Execute Permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.2 Appendix A-2 M instructions

This section describes the RISC-V M instruction set implemented by C920. And instructions of this section are 32-bit wide and listed in alphabetic order.

16.2.1 DIV——The Signed Divide Instruction

Syntax

div rd, rs1, rs2

Operation:

rd ← rs1 / rs2

Execute permission:

Machine mode (M-mode)/Supervisor mode (S-mode)/User mode (U-mode)

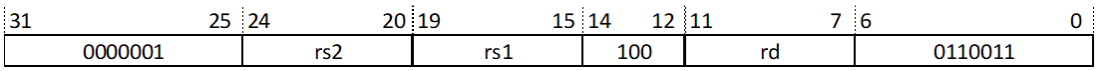
Exception:

None

Note:

- When the divisor is 0, the division result is 0xffffffffffffffff.
- When overflow occurs, the division result is 0x8000000000000000.

Instruction format:



16.2.2 DIVU——The Unsigned Divide Instruction

Syntax

divu rd, rs1, rs2

Operation:

rd ← rs1 / rs2

Execute permission:

M-mode/S-mode/U-mode

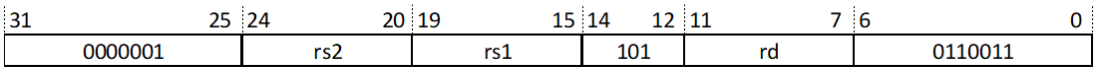
Exception:

None

Note:

When the divisor is 0, the division result is 0xffffffffffffffff.

Instruction format:



16.2.3 DIVUW——The Unsigned Divide Instruction on the Lower 32 Bits

Syntax

divuw rd, rs1, rs2

Operation:

tmp[31:0] ← (rs1[31:0] / rs2[31:0])[31:0]

rd←sign_extend(tmp[31:0])

Execute permission:

M-mode/S-mode/U-mode

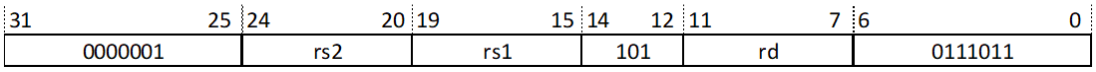
Exception:

None

Note:

When the divisor is 0, the division result is 0xffffffffffffffff.

Instruction format:



16.2.4 DIVW——The Signed Divide Instruction on the Lower 32 Bits

Syntax

divw rd, rs1, rs2

Operation:

tmp[31:0] ← (rs1[31:0] / rs2[31:0])[31:0]

rd←sign_extend(tmp[31:0])

Execute permission:

M-mode/S-mode/U-mode

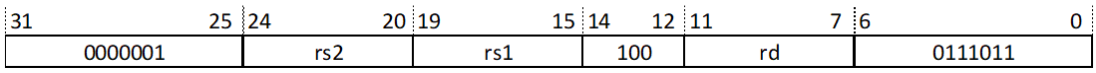
Exception:

None

Note:

- When the divisor is 0, the division result is 0xffffffffffffffff.
- When overflow occurs, the division result is 0x8000000000000000.

Instruction format:



16.2.5 MUL——The Signed Multiply Instruction

Syntax

mul rd, rs1, rs2

Operation:

rd ← (rs1 * rs2)[63:0]

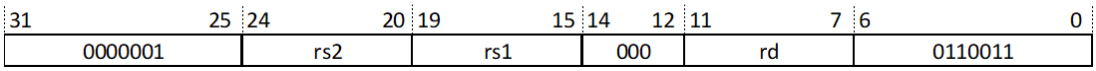
Execute permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.2.6 MULH——The Signed Multiply Upper Bit Extraction Instruction

Syntax

mulh rd, rs1, rs2

Operation:

rd ← (rs1 * rs2)[127:64]

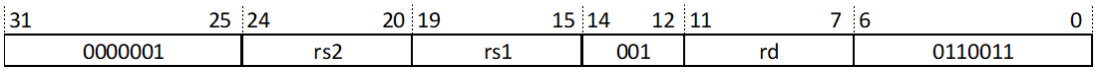
Execute permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.2.7 MULHSU——The Signed and Unsigned Multiply Upper Bit Extraction Instruction

Syntax

mulusu rd, rs1, rs2

Operation:

rd ← (rs1 * rs2)[127:64]

Execute permission:

M-mode/S-mode/U-mode

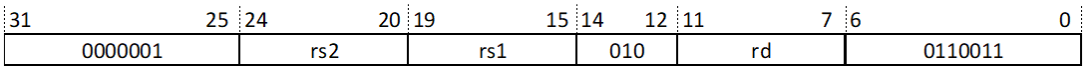
Exception:

None

Note:

rs1: Signed number; rs2: Unsigned number

Instruction format:



16.2.8 MULHU——The Unsigned Multiply Upper Bit Extraction Instruction

Syntax

mulhu rd, rs1, rs2

Operation:

rd ← (rs1 * rs2)[127:64]

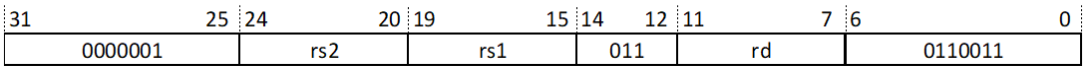
Execute permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.2.9 MULW——The Signed Multiply Instruction on the Lower 32 Bits

Syntax

mulw rd, rs1, rs2

Operation:

tmp ← (rs1[31:0] * rs2[31:0])[31:0]

rd ← sign_extend(tmp[31:0])

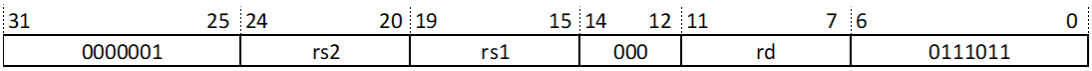
Execute permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.2.10 REM——The Signed Remainder Instruction

Syntax

rem rd, rs1, rs2

Operation:

rd ← rs1 % rs2

Execute permission:

M-mode/S-mode/U-mode

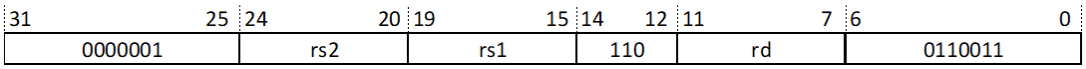
Exception:

None

Note:

- When the divisor is 0, the remainder operation result is the dividend.
- When overflow occurs, the remainder operation result is 0x0.

Instruction format:



16.2.11 REMU——The Unsigned Remainder Divide Instruction

Syntax

remu rd, rs1, rs2

Operation:

rd ← rs1 % rs2

Execute permission:

M-mode/S-mode/U-mode

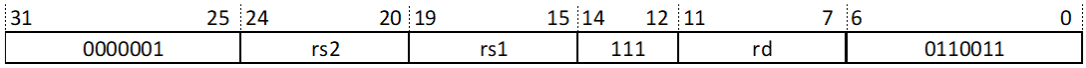
Exception:

None

Note:

When the divisor is 0, the remainder operation result is the dividend.

Instruction format:



16.2.12 REMUW——The Unsigned Remainder Divide Instruction on the Lower 32 Bits

Syntax

remw rd, rs1, rs2

Operation:

tmp ← (rs1[31:0] % rs2[31:0])[31:0]

rd ← sign_extend(tmp)

Execute permission:

M-mode/S-mode/U-mode

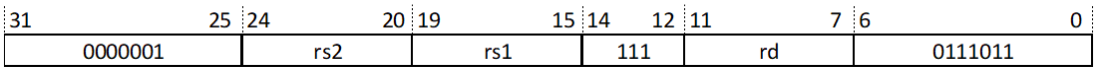
Exception:

None

Note:

When the divisor is 0, the remainder is the result of sign-extending the sign bit of the dividend at bit position [31].

Instruction format:



16.2.13 REMW——The Signed Remainder Divide Instruction on the Lower 32 Bits

Syntax

remw rd, rs1, rs2

Operation:

tmp[31:0] ← (rs1[31:0] % rs2[31:0])[31:0]

rd ← sign_extend(tmp[31:0])

Execute permission:

M-mode/S-mode/U-mode

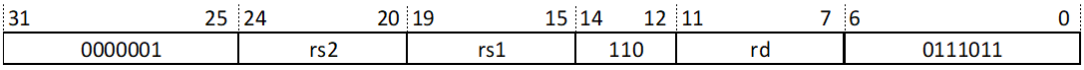
Exception:

None

Note:

- When the divisor is 0, the remainder is the result of sign-extending the sign bit of the dividend at bit position [31].
- When overflow occurs, the remainder operation result is 0x0.

Instruction format:



16.3 Appendix A-3 A Instructions

This section describes the RISC-V A instructions implemented by C920. The instructions of the section are 32-bit wide and listed in alphabetic order.

16.3.1 AMOADD.D——The Atomic Add Instruction

Syntax:

amoadd.d.aqrl rd, rs2, (rs1)

Operation:

rd ← mem[rs1+7: rs1]

mem[rs1+7:rs1] ← mem[rs1+7:rs1] + rs2

Execute permission:

Machine Mode (M-mode)/Supervisor Mode (S-mode)/User Mode (U-mode)

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

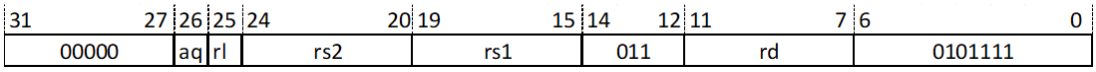
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amoadd.d rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amoadd.d.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amoadd.d.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amoadd.d.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed, and all memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.2 AMOADD.W——The Atomic Add Instruction on the Lower 32 Bits

Syntax:

amoadd.w.aqrl rd, rs2, (rs1)

Operation:

rd ←sign_extend(mem[rs1+3: rs1])

mem[rs1+3:rs1]← mem[rs1+3:rs1] + rs2[31:0]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

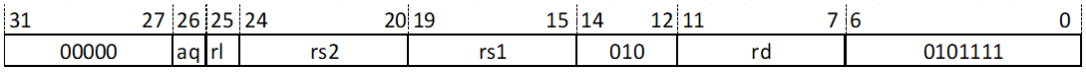
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amoadd.w rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amoadd.w.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amoadd.w.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amoadd.w.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed, and all memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.3 AMOAND.D——The Atomic Bitwise AND Instruction

Syntax:

amoand.d.aqrl rd, rs2, (rs1)

Operation:

rd ← mem[rs1+7: rs1]

mem[rs1+7:rs1] ← mem[rs1+7:rs1] & rs2

Execute permission: M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

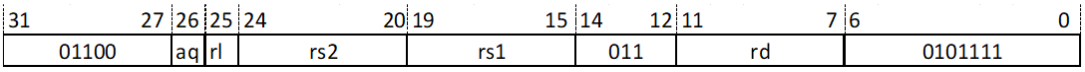
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amoand.d rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amoand.d.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amoand.d.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amoand.d.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed, and all memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.4 AMOAND.W——The Atomic Bitwise AND Instruction on the Lower 32 Bits

Syntax:

amoand.w.aqrl rd, rs2, (rs1)

Operation:

rd ← sign_extend(mem[rs1+3: rs1])

mem[rs1+3:rs1] ← mem[rs1+3:rs1] & rs2[31:0]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

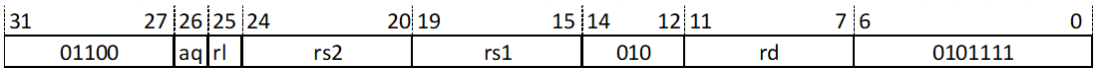
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amoand.w rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amoand.w.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amoand.w.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amoand.w.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed, and all memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.5 AMOMAX.D——The Atomic Signed Maximum Instruction on the Lower 32 Bits

Syntax:

amomax.d.aqrl rd, rs2, (rs1)

Operation:

rd ← mem[rs1+7: rs1]

mem[rs1+7:rs1] ← max(mem[rs1+7:rs1], rs2)

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

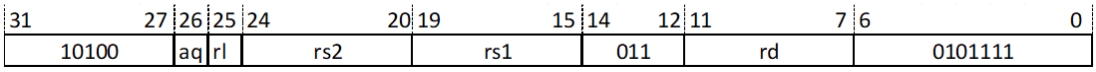
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amomax.d rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amomax.d.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amomax.d.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amomax.d.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed, and all memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.6 AMOMAX.W——The Atomic Signed Maximum Instruction on the Lower 32 Bits

Syntax:

amomax.w.aqrl rd, rs2, (rs1)

Operation:

rd ← sign_extend(mem[rs1+3: rs1])

mem[rs1+3:rs1]← max(mem[rs1+3:rs1], rs2[31:0])

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

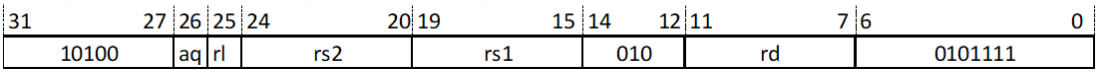
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amomax.w rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amomax.w.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amomax.w.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amomax.w.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed, and all memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.7 AMOMAXU.D——The Atomic Unsigned Maximum Instruction

Syntax:

amomaxu.d.aqrl rd, rs2, (rs1)

Operation:

rd ← mem[rs1+7: rs1]

mem[rs1+7:rs1] ← max(mem[rs1+7:rs1], rs2)

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amomaxu.d rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amomaxu.d.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amomaxu.d.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amomaxu.d.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed, and all memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:

31 27	26	25	24 20	19 15	14 12	11 7	6 0
11100	aq	rl	rs2	rs1	011	rd	0101111

16.3.8 AMOMAXU.W——The Atomic Unsigned Maximum Instruction on the Lower 32 Bits

Syntax:

amomaxu.w.aqrl rd, rs2, (rs1)

Operation:

rd ← zero_extend(mem[rs1+3: rs1])

mem[rs1+3:rs1] ← max(mem[rs1+3:rs1], rs2[31:0])

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

None

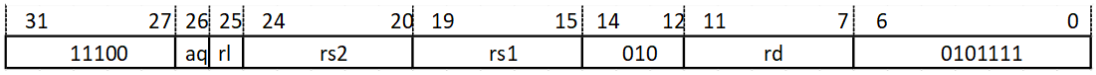
Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amomaxu.w rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amomaxu.w.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amomaxu.w.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

- aq=1,rl=1: The corresponding assembler instruction is amomaxu.w.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.9 AMOMIN.D——The Atomic Signed Minimum Instruction

Syntax:

amomin.d.aqrl rd, rs2, (rs1)

Operation:

rd ← mem[rs1+7: rs1]

mem[rs1+7:rs1] ← min(mem[rs1+7:rs1],rs2)

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

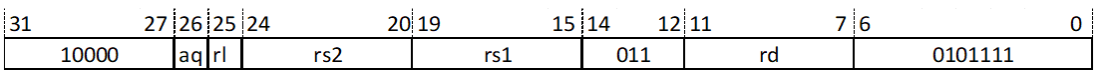
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amomin.d rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amomin.d.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amomin.d.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amomin.d.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.10 AMOMIN.W——The Atomic Signed Minimum Instruction on the Lower 32 Bits

Syntax:

amomin.w.aqrl rd, rs2, (rs1)

Operation:

rd ← sign_extend(mem[rs1+3: rs1])

mem[rs1+3:rs1] ← min(mem[rs1+3:rs1], rs2[31:0])

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

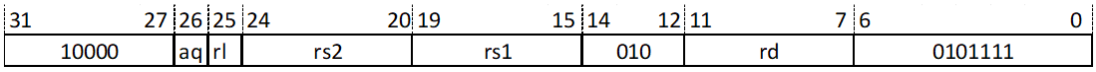
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amomin.w rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amomin.w.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amomin.w.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction isamomin.w.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.11 AMOMINU.D——The Atomic Unsigned Minimum Instruction

Syntax:

amominu.d.aqrl rd, rs2, (rs1)

Operation:

rd ← mem[rs1+7: rs1]

mem[rs1+7:rs1] ← min(mem[rs1+7:rs1], rs2)

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

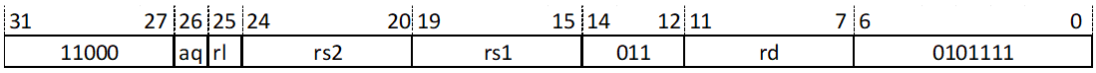
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amominu.d rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amominu.d.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amominu.d.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amominu.d.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.12 AMOMINU.W——The Atomic Unsigned Minimum Instruction on the Lower 32 Bits

Syntax:

amominu.w.aqrl rd, rs2, (rs1)

Operation:

rd ← sign_extend(mem[rs1+3: rs1])

mem[rs1+3:rs1] ← min(mem[rs1+3:rs1], rs2[31:0])

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

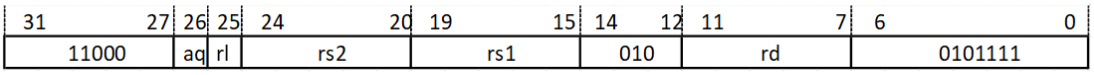
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amominu.w rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amominu.w.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amominu.w.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amominu.w.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.13 AMOOR.D——The Atomic Bitwise OR Instruction

Syntax:

amoor.d.aqrl rd, rs2, (rs1)

Operation:

rd ← mem[rs1+7: rs1]

mem[rs1+7:rs1] ← mem[rs1+7:rs1] | rs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

None

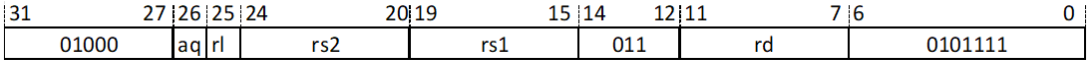
Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amoor.d rd, rs2, (rs1).

- aq=0,rl=1: The corresponding assembler instruction is amoor.d.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amoor.d.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amoor.d.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.14 AMOOR.W——The Atomic Bitwise OR Instruction on the Lower 32 Bits

Syntax:

amoor.w.aqrl rd, rs2, (rs1)

Operation:

rd ← sign_extend(mem[rs1+3: rs1])

mem[rs1+3:rs1] ← mem[rs1+3:rs1] | rs2[31:0]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

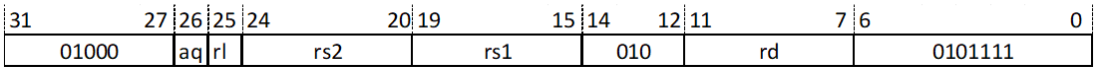
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amoor.w rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amoor.w.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amoor.w.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amoor.w.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.15 AMOSWAP.D——The Atomic Swap Instruction

Syntax:

amoswap.d.aqrl rd, rs2, (rs1)

Operation:

rd ← mem[rs1+7: rs1]

mem[rs1+7:rs1] ←rs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

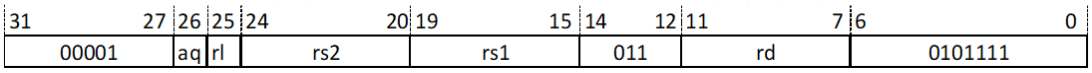
Affected flag: None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amoswap.d rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amoswap.d.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amoswap.d.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amoswap.d.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.16 AMOSWAP.W——The Atomic Swap Instruction on the Lower 32 Bits

Syntax:

amoswap.w.aqrl rd, rs2, (rs1)

Operation:

rd ← sign_extend(mem[rs1+3: rs1])

mem[rs1+3:rs1] ←rs2[31:0]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

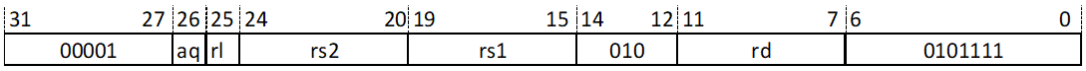
Affected flag: None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amoswap.w rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amoswap.w.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amoswap.w.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amoswap.w.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.17 AMOXOR.D——The Atomic Bitwise XOR Instruction

Syntax:

amoxor.d.aqrl rd, rs2, (rs1)

Operation:

rd ← mem[rs1+7: rs1]

mem[rs1+7:rs1] ← mem[rs1+7:rs1] ^ rs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

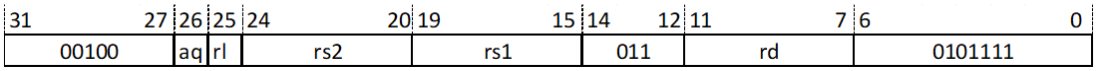
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amoxor.d rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amoxor.d.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is amoxor.d.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amoxor.d.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.18 AMOXOR.W——The Atomic Bitwise XOR Instruction on the Lower 32 Bits

Syntax:

amoxor.w.aqrl rd, rs2, (rs1)

Operation:

rd ← sign_extend(mem[rs1+3: rs1])

mem[rs1+3:rs1] ← mem[rs1+3:rs1] ^ rs2[31:0]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

None

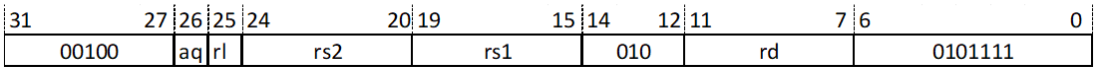
Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is amoxor.w rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is amoxor.w.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.

- aq=1,rl=0: The corresponding assembler instruction is amoxor.w.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is amoxor.w.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.19 LR.D——The Doubleword Load-reserved Instruction

Syntax:

lr.d.aqrl rd, (rs1)

Operation:

rd ← mem[rs1+7: rs1]

mem[rs1+7:rs1] is reserved

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

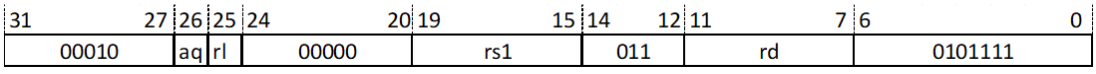
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is lr.d rd, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is lr.d.rl rd, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is lr.d.aq rd, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is lr.d.aqrl rd, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.20 LR.W——The Word Load-reserved Instruction

Syntax:

lr.w.aqrl rd, (rs1)

Operation:

rd ←sign_extend(mem[rs1+3: rs1])

mem[rs1+3:rs1] is reserved

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is lr.w rd, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is lr.w.rl rd, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is lr.w.aq rd, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is lr.w.aqrl rd, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00010	aq	rl	00000	rs1	010	rd	0101111

16.3.21 SC.D——The Doubleword Conditional Store Instruction

Syntax:

sc.d.aqrl rd, rs2, (rs1)

Operation:

If(mem[rs1+7:rs1] is reserved)

mem[rs1+7: rs1] ← rs2

rd ← 0

else

rd ← 1

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

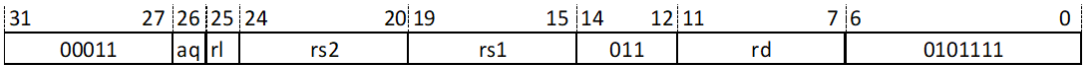
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is sc.d rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is sc.d.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is sc.d.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is sc.d.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.3.22 SC.W——The Word Conditional Store Instruction

Syntax:

sc.w.aqrl rd, rs2, (rs1)

Operation:

if(mem[rs1+3:rs1] is reserved)

mem[rs1+3:rs1] ← rs2[31:0]

rd ← 0

else

rd ← 1

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions.

Affected flag:

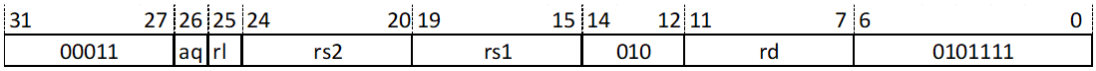
None

Note:

The aq and rl bits determine the execution order of memory access instructions in the pre-order and post-order respectively:

- aq=0,rl=0: The corresponding assembler instruction is sc.w rd, rs2, (rs1).
- aq=0,rl=1: The corresponding assembler instruction is sc.w.rl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed.
- aq=1,rl=0: The corresponding assembler instruction is sc.w.aq rd, rs2, (rs1). All memory access instructions after the instruction can be executed only after execution of the instruction is completed.
- aq=1,rl=1: The corresponding assembler instruction is sc.w.aqrl rd, rs2, (rs1). The Results of all memory access instructions before the instruction must be observed before the instruction is executed. All memory access instructions after the instruction can be executed only after execution of the instruction is completed.

Instruction format:



16.4 Appendix A-4 F instructions

This section describes the RISC-V F instructions implemented by C920. The instructions are 32-bit wide and listed in alphabetic order.

For single-precision floating-point instructions, if the upper 32 bits in the source register are not all 1, the single-precision data is treated as cNaN.

When mstatus.fs==2' b00, executing any instruction listed in this section will trigger an illegal instruction exception; When mstatus.fs != 2' b00, mstatus.fs will be set to 2' b11 after executing any instruction in this section

16.4.1 FADD.S——The Single-precision Floating-point Add Instruction

Syntax:

fadd.s fd, fs1, fs2, rm

Operation:

frd ← fs1 + fs2

Execute permission:

Machine Mode (M-mode)/Supervisor Mode (S-mode)/User Mode (U-mode)

Exception:

The illegal instruction exception

Affected flag:

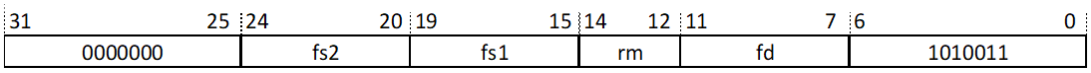
Floating-point status bit NV/OF/NX

Note:

rm determines the round-off mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembler instruction is fadd.s fd, fs1,fs2,rne.
- 3' b001: Rounds to zero. And the corresponding assembler instruction is fadd.s fd, fs1,fs2,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembler instruction is fadd.s fd, fs1,fs2,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembler instruction is fadd.s fd, fs1,fs2,rup.
- 3' b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fadd.s fd, fs1,fs2,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the rm bit of the Floating-point Control and Status Register (fcsr). And the corresponding assembler instruction is fadd.s fd, fs1,fs2.

Instruction format:



16.4.2 FCLASS.S——The Single-precision Floating-point Classification Instruction

Syntax:

fclass.s rd, fs1

Operation:

if (fs1 = -inf)

rd ← 64' h1

if (fs1 = -norm)

rd ← 64' h2

if (fs1 = -subnorm)

rd ← 64’ h4

if (fs1 = -zero)

rd ← 64’ h8

if (fs1 = +zero)

rd ← 64’ h10

if (fs1 = +subnorm)

rd ← 64’ h20

if (fs1 = +norm)

rd ← 64’ h40

if (fs1 = +Inf)

rd ← 64’ h80

if (fs1 = sNaN)

rd ← 64’ h100

if (fs1 = qNaN)

rd ← 64’ h200

Execute permission:

M-mode/S-mode/U-mode

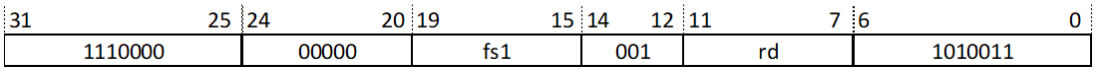
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



16.4.3 FCVT.L.S——The Instruction to Convert a Single-precision Floating-point Number to a Signed Long Integer

Syntax:

fcvt.l.s rd, fs1, rm

Operation:

rd ← single_convert_to_signed_long(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

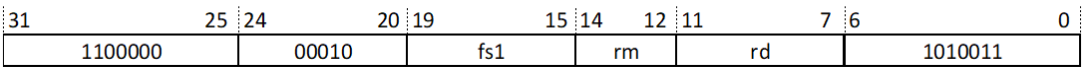
Floating-point status bit NV/NX

Note:

rm determines the round-off mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembler instruction is fcvt.l.s rd,fs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembler instruction is fcvt.l.s rd,fs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembler instruction is fcvt.l.s rd,fs1,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembler instruction is fcvt.l.s rd,fs1,rup.
- 3' b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fcvt.l.s rd,fs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fcvt.l.s rd, fs1.

Instruction format:



16.4.4 FCVT.LU.S——The Instruction to Convert a Single-precision Floating-point Number to a Unsigned Long Integer

Syntax:

fcvt.lu.s rd, fs1, rm

Operation:

rd ← single_convert_to_unsigned_long(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV/NX

Note:

rm determines the round-off mode:

- 3'b000: Rounds to the nearest even number. And the corresponding assembler instruction is fcvt.lu.s rd,fs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembler instruction is fcvt.lu.s rd,fs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembler instruction is fcvt.lu.s rd,fs1,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembler instruction is fcvt.lu.s rd,fs1,rup.
- 3'b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fcvt.lu.s rd,fs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fcvt.lu.s rd, fs1.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00011	fs1	rm	rd	1010011

16.4.5 FCVT.S.L——The Instruction to Convert a Signed Long Integer to a Single-precision Floating-point Number

Syntax:

fcvt.s.l fd, rs1, rm

Operation:

fd ← signed_long_convert_to_single(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NX

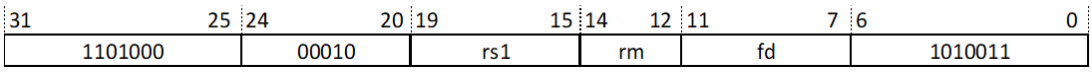
Note:

rm determines the round-off mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembler instruction is fcvt.s.l fd,rs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembler instruction is fcvt.s.l fd,rs1,rtz.

- 3’ b010: Rounds to negative infinity. And the corresponding assembler instruction is fcvt.s.l fd,fs1,rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembler instruction is fcvt.s.l fd,fs1,rup.
- 3’ b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fcvt.s.l fd,fs1,rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fcvt.s.l fd, fs1.

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
1101000	00010	rs1	rm	fd	1010011

16.4.6 FCVT.S.LU——The Instruction to Convert a Unsigned Long Integer to a Single-precision Floating-point Number

Syntax:

fcvt.s.l fd, fs1, rm

Operation:

fd ← unsigned_long_convert_to_single_fp(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NX

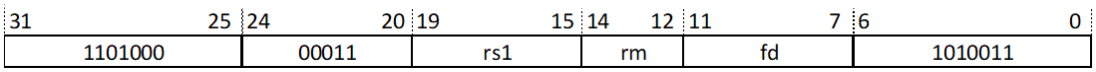
Note:

rm determines the round-off mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembler instruction is fcvt.s.lu fd,fs1,rne.
- 3’ b001: Rounds to zero. And the corresponding assembler instruction is fcvt.s.lu fd, fs1,rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembler instruction is fcvt.s.lu fd, fs1,rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembler instruction is fcvt.s.lu fd, fs1,rup.
- 3’ b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fcvt.s.lu fd, fs1,rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.

- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fcvt.s.lu fd, fs1.

Instruction format:



16.4.7 FCVT.S.W——The Instruction to Convert a Signed Integer to a Single-precision Floating-point Number

Syntax:

fcvt.s.w fd, rs1, rm

Operation:

fd ← signed_int_convert_to_single(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

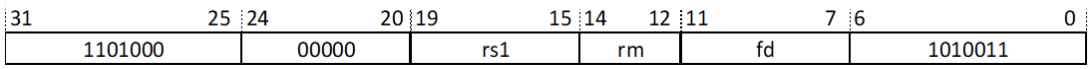
Floating-point status bit NX

Note:

rm determines the round-off mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembler instruction is fcvt.s.w fd,rs1,rne.
- 3’ b001: Rounds to zero. And the corresponding assembler instruction is fcvt.s.w fd,rs1,rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembler instruction is fcvt.s.w fd,rs1,rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembler instruction is fcvt.s.w fd,rs1,rup.
- 3’ b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fcvt.s.w fd,rs1,rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fcvt.s.w fd, rs1.

Instruction format:



16.4.8 FCVT.S.WU——The Instruction to Convert a Unsigned Integer to a Single-precision Floating-point Number

Syntax:

fcvt.s.wu fd, rs1, rm

Operation:

fd ← unsigned_int_convert_to_single_fp(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

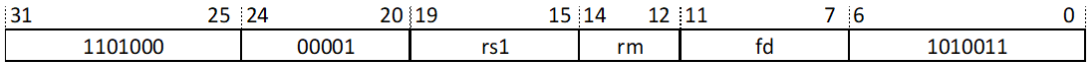
Floating-point status bit NX

Note:

rm determines the round-off mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembler instruction is fcvt.s.wu fd,rs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembler instruction is fcvt.s.wu fd,rs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembler instruction is fcvt.s.wu fd,rs1,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembler instruction is fcvt.s.wu fd,rs1,rup.
- 3' b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fcvt.s.wu fd,rs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fcvt.s.wu fd, rs1.

Instruction format:



16.4.9 FCVT.W.S——The Instruction to Convert a Single-precision Floating-point Number to a Signed Integer

Syntax:

fcvt.w.s rd, fs1, rm

Operation:

tmp ← single_convert_to_signed_int(fs1)

rd←sign_extend(tmp)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

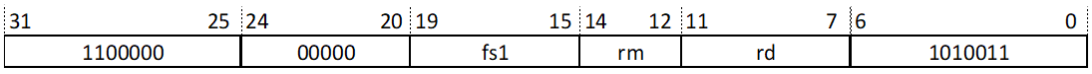
Floating-point status bit NV/NX

Note:

rm determines the round-off mode:

- 3'b000: Rounds to the nearest even number. And the corresponding assembler instruction is fcvt.w.s rd,fs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembler instruction is fcvt.w.s rd,fs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembler instruction is fcvt.w.s rd,fs1,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembler instruction is fcvt.w.s rd,fs1,rup.
- 3'b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fcvt.w.s rd,fs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fcvt.w.s rd, fs1.

Instruction format:



16.4.10 FCVT.WU.S——The Instruction to Convert a Single-precision Floating-point Number to a Unsigned Integer

Syntax:

fcvt.wu.s rd, fs1, rm

Operation:

tmp ← single_convert_to_unsigned_int(fs1)

rd←sign_extend(tmp)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV/NX

Note:

rm determines the round-off mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembler instruction is fcvt.wu.s rd,fs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembler instruction is fcvt.wu.s rd,fs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembler instruction is fcvt.wu.s rd,fs1,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembler instruction is fcvt.wu.s rd,fs1,rup.
- 3' b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fcvt.wu.s rd,fs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fcvt.wu.s rd, fs1.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00001	fs1	rm	rd	1010011

16.4.11 FDIV.S——The Single-precision Floating-point Divide instruction

Syntax:

fdiv.s fd, fs1, fs2, rm

Operation:

fd ← fs1 / fs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

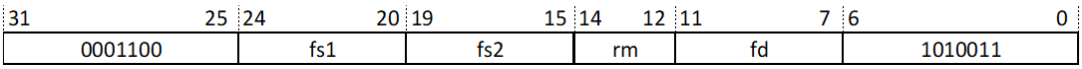
Floating-point status bit NV/DZ/OF/UF/NX

Note:

rm determines the round-off mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembler instruction is fdiv.s fs1,fs2,rne.
- 3’ b001: Rounds to zero. And the corresponding assembler instruction is fdiv.s fd fs1,fs2,rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembler instruction is fdiv.s fd, fs1,fs2,rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembler instruction is fdiv.s fd, fs1,fs2,rup.
- 3’ b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fdiv.s fd, fs1,fs2,rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fdiv.s fd, fs1,fs2.

Instruction format:



16.4.12 FEQ.S——The Single-precision Floating-point Compare Equal Instruction

Syntax:

feq.s rd, fs1, fs2

Operation:

if(fs1 == fs2)

rd ← 1

else

rd ← 0

Execute permission:

M-mode/S-mode/U-mode

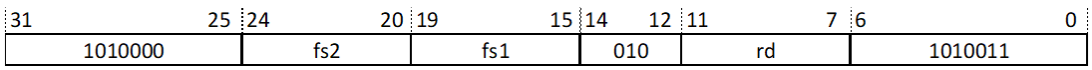
Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV

Instruction format:



16.4.13 FLE.S——The Single-precision Floating-point Compare Less than or Equal to Instruction

Syntax:

fle.s rd, fs1, fs2

Operation:

if(fs1 <= fs2)

rd ← 1

else

rd ← 0

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1010000	fs2	fs1	000	rd	1010011

16.4.14 FLT.S——The Single-precision Floating-point Compare Less than Instruction

Syntax:

flt.s rd, fs1, fs2

Operation:

if(fs1 < fs2)

rd ← 1

else

rd ← 0

Execute permission:

M-mode/S-mode/U-mode

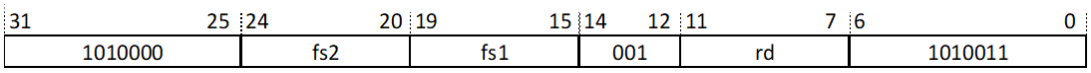
Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV

Instruction format:



16.4.15 FLW——The Single-precision Floating-point Load Instruction

Syntax:

flw fd, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

fd[31:0] ← mem[(address+3):address]

fd[63:32] ← 32’ hffffffff

Execute permission:

M-mode/S-mode/U-mode

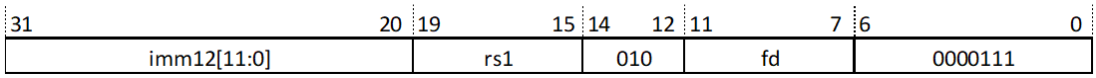
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions, and illegal instruction exceptions.

Affected flag:

None

Instruction format:



16.4.16 FMADD.S——The Single-precision Floating-point Multiply-add Instruction

Syntax:

fmadd.s fd, fs1, fs2, fs3, rm

Operation:

rd ← fs1*fs2 + fs3

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

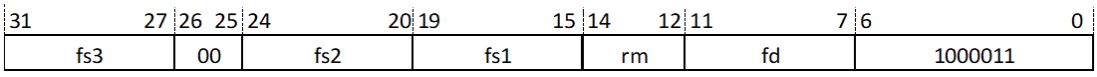
Floating-point status bit NV/OF/UF/IX

Note:

rm determines the round-off mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembler instruction is fmadd.s fd,fs1, fs2, fs3, rne.
- 3' b001: Rounds to zero. And the corresponding assembler instruction is fmadd.s fd,fs1, fs2, fs3, rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembler instruction is fmadd.s fd,fs1, fs2, fs3, rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembler instruction is fmadd.s fd,fs1, fs2, fs3, rup.
- 3' b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fmadd.s fd,fs1, fs2, fs3, rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fmadd.s fd,fs1, fs2, fs3.

Instruction format:



16.4.17 FMAX.S——The Single-Precision Floating-Point Maxmum Instruction

Syntax:

fmax.s fd, fs1, fs2

Operation:

if(fs1 >= fs2)

fd ← fs1

else

fd ← fs2

Execute permission:

M-mode/S-mode/U-mode

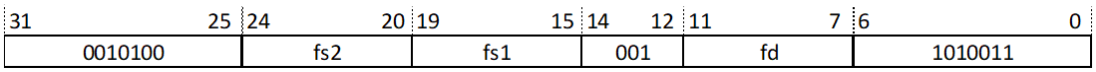
Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV

Instruction format:



16.4.18 FMIN.S——The Single-Precision Floating-Point Minimum Instruction

Syntax:

fmin.s fd, fs1, fs2

Operation:

if(fs1 >= fs2)
    fd ← fs2
else
    fd ← fs1

Execute permission:

M-mode/S-mode/U-mode

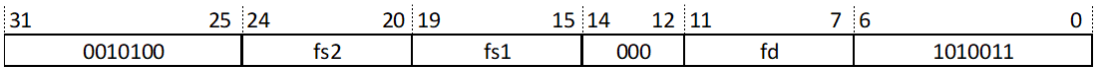
Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV

Instruction format:



16.4.19 FMSUB.S——The Single-precision Floating-point Multiply-subtract Instruction

Syntax:

fmsub.s fd, fs1, fs2, fs3, rm

Operation:

fd ← fs1*fs2 - fs3

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

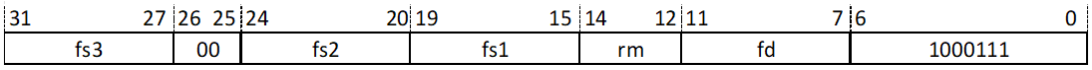
Floating-point status bit NV/OF/UF/IX

Note:

rm determines the round-off mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembler instruction is fmsub.s fd,fs1, fs2, fs3, rne.
- 3’ b001: Rounds to zero. And the corresponding assembler instruction is fmsub.s fd,fs1, fs2, fs3, rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembler instruction is fmsub.s fd,fs1, fs2, fs3, rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembler instruction is fmsub.s fd,fs1, fs2, fs3, rup.
- 3’ b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fmsub.s fd, fs1, fs2, fs3,rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fmsub.s fd,fs1, fs2, fs3.

Instruction format:



16.4.20 FMUL.S——The Single-precision Floating-point Multiply Instruction

Syntax:

fmul.s fd, fs1, fs2, rm

Operation:

fd ← fs1 * fs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV/OF/UF/NX

Note:

rm determines the round-off mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembler instruction is fmul.s fd, fs1, fs2, rne.
- 3’ b001: Rounds to zero. And the corresponding assembler instruction is fmul.s fd, fs1, fs2, rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembler instruction is fmul.s fd, fs1, fs2, rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembler instruction is fmul.s fd, fs1, fs2, rup.
- 3’ b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fmul.s fd, fs1,fs2, rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fmul.s fs1,fs2.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0001000	fs2	fs1	rm	fd	1010011

16.4.21 FMV.W.X——The Single-precision Floating-point Write Transfer Instruction

Syntax:

fmv.w.x fd, rs1

Operation:

fd[31:0] ← rs[31:0]

fd[63:32] ← 32’ hffffffff

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1111000	00000	rs1	000	fd	1010011

16.4.22 FMV.X.W——The Single-precision Floating-point Register Read Transfer Instruction

Syntax:

fmv.x.w rd, fs1

Operation:

tmp[31:0] ← fs1[31:0]

rd ← sign_extend(tmp[31:0])

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1110000	00000	fs1	000	rd	1010011

16.4.23 FNMADD.S——The Single-precision Floating-point Negate-(Multiply-add) Instruction

Syntax:

fnmadd.s fd, fs1, fs2, fs3, rm

Operation:

fd ←-(fs1*fs2 + fs3)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

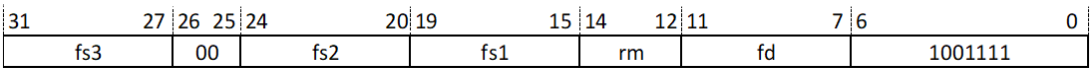
Floating-point status bit NV/OF/UF/IX

Note:

rm determines the round-off mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembler instruction is fnmadd.s fd,fs1, fs2, fs3, rne.
- 3’ b001: Rounds to zero. And the corresponding assembler instruction is fnmadd.s fd,fs1, fs2, fs3, rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembler instruction is fnmadd.s fd,fs1, fs2, fs3, rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembler instruction is fnmadd.s fd,fs1, fs2, fs3, rup.
- 3’ b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fnmadd.s fd,fs1, fs2, fs3, rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fnmadd.s fd,fs1, fs2, fs3.

Instruction format:



16.4.24 FNMSUB.S——The Single-precision Floating-point Negate-(Multiply-subtract) Instruction

Syntax:

fnmsub.s fd, fs1, fs2, fs3, rm

Operation:

fd ← -(fs1*fs2 - fs3)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV/OF/UF/IX

Note:

rm determines the round-off mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembler instruction is fnmsub.s fd,fs1, fs2, fs3, rne.
- 3’ b001: Rounds to zero. And the corresponding assembler instruction is fnmsub.s fd,fs1, fs2, fs3, rtz.

- 3’ b010: Rounds to negative infinity. And the corresponding assembler instruction is fnmsub.s fd,fs1, fs2, fs3, rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembler instruction is fnmsub.s fd,fs1, fs2, fs3, rup.
- 3’ b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fnmsub.s fd,fs1, fs2, fs3, rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fnmsub.s fd,fs1, fs2, fs3.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
fs3	00	fs2	fs1	rm	fd	1001011

16.4.25 FSGNJ.S——The Single-precision Floating-point Sign-injection Instruction

Syntax:

fsgnj.s fd, fs1, fs2

Operation:

fd[30:0] ← fs1[30:0]

fd[31] ← fs2[31]

fd[63:32] ← 32’ hffffffff

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0010000	fs2	fs1	000	fd	1010011

16.4.26 FSGNJN.S——The Single-precision Floating-point Negate Sign-injection Instruction

Syntax:

fsgnjn.s fd, fs1, fs2

Operation:

fd[30:0] ← fs1[30:0]

fd[31] ← ! fs2[31]

fd[63:32] ← 32’ hffffffff

Execute permission:

M-mode/S-mode/U-mode

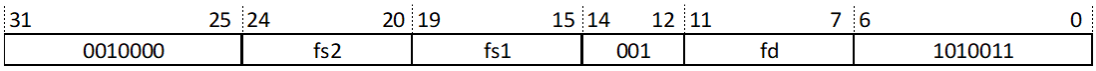
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



16.4.27 FSGNJX.S——The Single-precision Floating-point XOR Sign-injection Instruction

Syntax:

fsgnjx.s fd, fs1, fs2

Operation:

fd[30:0] ← fs1[30:0]

fd[31] ← fs1[31] ^ fs2[31]

fd[63:32] ← 32’ hffffffff

Execute permission:

M-mode/S-mode/U-mode

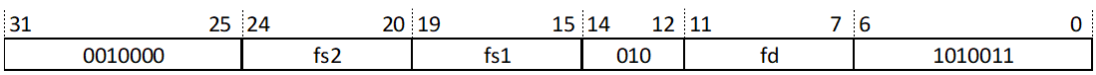
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



16.4.28 FSQRT.S——The Single-precision Floating-point Square-root Instruction

Syntax:

fsqrt.s fd, fs1, rm

Operation:

fd ← sqrt(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

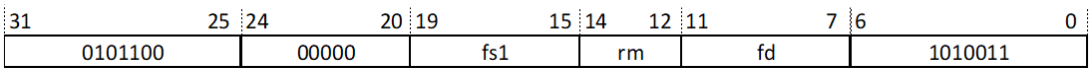
Floating-point status bit NV/NX

Note:

rm determines the round-off mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembler instruction is fsqrt.s fd, fs1,rne
- 3' b001: Rounds to zero. And the corresponding assembler instruction is fsqrt.s fd, fs1,rtz
- 3' b010: Rounds to negative infinity. And the corresponding assembler instruction is fsqrt.s fd, fs1,rdn
- 3' b011: Rounds to positive infinity. And the corresponding assembler instruction is fsqrt.s fd, fs1,rup
- 3' b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fsqrt.s fd, fs1,rmm
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fsqrt.s fd, fs1.

Instruction format:



16.4.29 FSUB.S——The Single-precision Floating-point Subtract Instruction

Syntax:

fsub.s fd, fs1, fs2, rm

Operation:

fd ← fs1 - fs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

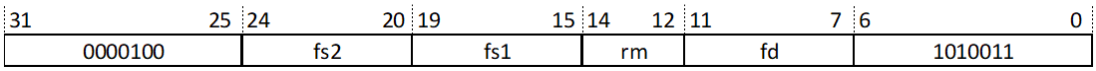
Floating-point status bit NV/OF/NX

Note:

rm determines the round-off mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembler instruction is fsub.fd, fs1,fs2,rne
- 3' b001: Rounds to zero. And the corresponding assembler instruction is fsub.s fd, fs1,fs2,rtz
- 3' b010: Rounds to negative infinity. And the corresponding assembler instruction is fsub.s fd, fs1,fs2,rdn
- 3' b011: Rounds to positive infinity. And the corresponding assembler instruction is fsub.s fd, fs1,fs2,rup
- 3' b100: Rounds to the nearest larger value. And the corresponding assembler instruction is fsub.s fd, fs1,fs2,rmm
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembler instruction is fsub.s fd, fs1,fs2.

Instruction format:



16.4.30 FSW——The Single-precision Floating-point Store Instruction

Syntax:

fsw fs2, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)
mem[(address+31):address] ← fs2[31:0]

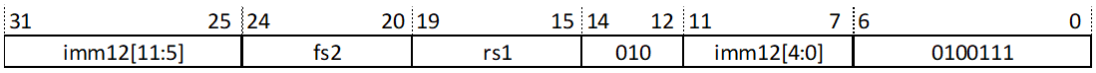
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions.

Instruction format:



16.5 Appendix A-5 D Instructions

This section describes the RISC-V D instructions implemented by C920. And the instructions are 32-bit wide, listed in alphabetic order.

When mstatus.fs==2' b00, execution of the instructions in this section will occur an illegal instruction exception; When mstatus.fs != 2' b00, mstatus.fs is set to 2' b11 after the execution of any instruction in this section.

16.5.1 FADD.D——Double-Precision Floating-Point Add Instruction

Syntax:

fadd.d fd, fs1, fs2, rm

Operation:

fd ← fs1 + fs2

Execute permission:

Machine Mode (M-mode)/Supervisor Mode (S-mode)/User-mode (U-mode)

Exception:

The illegal instruction exception

Affected flag:

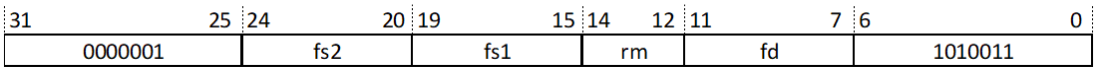
Floating-point state: Invalid Operation (NV)/Overflow (OF)/Inexact (NX)

Note:

rm determins the rounding mode:

- 3' b000: Rounding to the nearest number, corresponding to the assembly instruction: fadd.d fd, fs1,fs2,rne
- 3' b001: Rounding to zero, corresponding to the assembly instruction: fadd.d fd, fs1,fs2,rtz
- 3' b010: Rounding to negative infinity. And the corresponding assembly instruction is: fadd.d fd, fs1,fs2,rdn
- 3' b011: Rounding to positive infinity. And the corresponding assembly instruction is: fadd.d fd, fs1,fs2,rup
- 3' b100: Rounding to the nearest large value. And the corresponding assembly instruction is: fadd.d fd, fs1,fs2,rmm
- 3' b101: This code is reserved and not used..
- 3' b110: This code is reserved and not used..
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fadd.d fd, fs1,fs2.

Instruction format:



16.5.2 FCLASS.D——Double-Precision Floating-Point Classification Instructions

Syntax:

fclass.d rd, fs1

Operation:

if (fs1 = -Inf)

rd ← 64’ h1

if (fs1 = -norm)

rd ← 64’ h2

if (fs1 = -subnorm)

rd ← 64’ h4

if (fs1 = -zero)

fd ← 64’ h8

if (fs1 = +Zero)

rd ← 64’ h10

if (fs1 = +subnorm)

rd ← 64’ h20

if (fs1 = +norm)

rd ← 64’ h40

if (fs1 = +Inf)

rd ← 64’ h80

if (fs1 = sNaN)

rd ← 64’ h100

if (fs1 = qNaN)

rd ← 64’ h200

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1110001	00000	fs1	001	rd	1010011

16.5.3 FCVT.D.L——The Instruction to Convert a Signed Long Integer to a Double Precision Floating Point Number

Syntax:

fcvt.d.l fd, rs1, rm

Operation:

fd ← signed_long_convert_to_double(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point state: NX

Note:

rm determins the rounding mode:

- 3' b000: Rounding to the nearest number, corresponding to the assembly instruction: fcvt.d.l fd,rs1,rne.
- 3' b001: Rounding to zero, corresponding to the assembly instruction: fcvt.d.l fd,rs1,rtz.
- 3' b010: Rounding to negative infinity. And the corresponding assembly instruction is: fcvt.d.l fd,rs1,rdn.
- 3' b011: Rounding to positive infinity. And the corresponding assembly instruction is: fcvt.d.l fd,rs1,rup.
- 3'b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fcvt.d.l fd,rs1,rmm.
- 3' b101: This code is reserved and not used..
- 3' b110: This code is reserved and not used..
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit of the fcsr register. And the corresponding assembly instruction isfcvt.d.l fd, rs1.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00010	rs1	rm	fd	1010011

16.5.4 FCVT.D.LU——The Instruction to Convert an Unsigned Long Integer to a Double-Precision Floating-Point Number

Syntax:

fcvt.d.lu fd, rs1, rm

Operation:

fd ← unsigned_long_convert_to_double(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

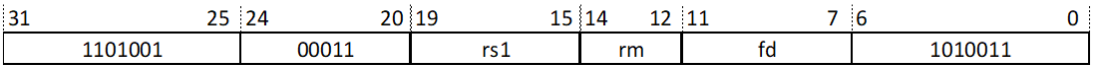
Floating-point state: NX

Note:

rm determins the rounding mode:

- 3' b000: Rounding to the nearest number, corresponding to the assembly instruction: fcvt.d.lu fd,rs1,rne.
- 3' b001: Rounding to zero, corresponding to the assembly instruction: fcvt.d.lu fd,rs1,rtz.
- 3' b010: Rounding to negative infinity. And the corresponding assembly instruction is: fcvt.d.lu fd,rs1,rdn.
- 3' b011: Rounding to positive infinity. And the corresponding assembly instruction is: fcvt.d.lu fd,rs1,rup.
- 3' b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fcvt.d.lu fd,rs1,rmm.
- 3' b101: This code is reserved and not used..
- 3' b110: This code is reserved and not used..
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit of the fcsr register. And the corresponding assembly instruction isfcvt.d.lu fd, rs1.

Instruction format:



16.5.5 FCVT.D.S——The Instruction to Convert a Single-Precision Floating-Point Number to a Double-Precision Floating-Point Number

Syntax:

fcvt.d.s fd, fs1

Operation:

fd ← single_convert_to_double(fs1)

Execute permission:

M-mode/S-mode/U-mode

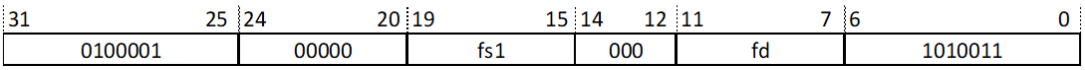
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
0100001	00000	fs1	000	fd	1010011

16.5.6 FCVT.D.W——The Instruction to Convert a Signed Integer to a Double-Precision Floating-Point Number

Syntax:

fcvt.d.w fd, rs1

Operation:

fd ← signed_int_convert_to_double(fs1)

Execute permission:

M-mode/S-mode/U-mode

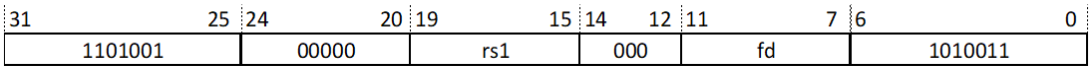
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
1101001	00000	rs1	000	fd	1010011

16.5.7 FCVT.D.WU——The Instruction to Convert an Unsigned Integer to a Double-Precision Floating-Point Number

Syntax:

fcvt.d.wu fd, rs1

Operation:

fd ← unsigned_int_convert_to_double(fs1)

Execute permission:

M-mode/S-mode/U-mode

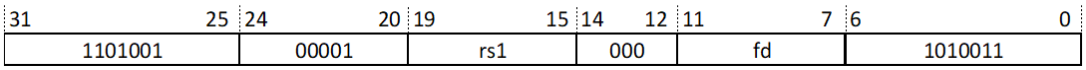
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
1101001	00001	rs1	000	fd	1010011

16.5.8 FCVT.L.D——The Instruction to Convert a Double-Precision Floating-Point Number to a Signed Long Integer

Syntax:

fcvt.l.d rd, fs1, rm

Operation:

rd ← double_convert_to_signed_long(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point state: NV/NX

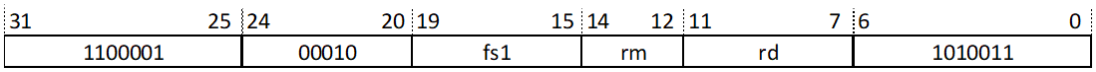
Note:

rm determins the rounding mode:

- 3' b000: Rounding to the nearest number, corresponding to the assembly instruction: fcvt.l.d rd,fs1,rne.
- 3' b001: Rounding to zero, corresponding to the assembly instruction: fcvt.l.d rd,fs1,rtz.
- 3' b010: Rounding to negative infinity. And the corresponding assembly instruction is: fcvt.l.d rd,fs1,rdn.
- 3' b011: Rounding to positive infinity. And the corresponding assembly instruction is: fcvt.l.d rd,fs1,rup.
- 3'b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fcvt.l.d rd,fs1,rmm.
- 3' b101: This code is reserved and not used..
- 3' b110: This code is reserved and not used..

- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fcvt.l.d rd, fs1.

Instruction format:



16.5.9 FCVT.LU.D——The Instruction to Convert a Double-Precision Floating-Point Number to an Unsigned Long Integer

Syntax:

fcvt.lu.d rd, fs1, rm

Operation:

rd ← double_convert_to_unsigned_long(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

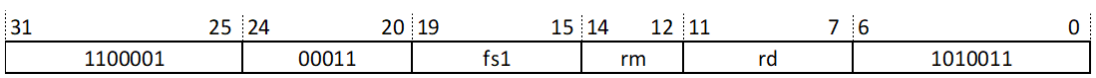
Floating-point state: NV/NX

Note:

rm determins the rounding mode:

- 3’ b000: Rounding to the nearest number, corresponding to the assembly instruction: fcvt.lu.d rd,fs1,rne.
- 3’ b001: Rounding to zero, corresponding to the assembly instruction: fcvt.lu.d rd,fs1,rtz.
- 3’ b010: Rounding to negative infinity. And the corresponding assembly instruction is: fcvt.lu.d rd,fs1,rdn.
- 3’ b011: Rounding to positive infinity. And the corresponding assembly instruction is: fcvt.lu.d rd,fs1,rup.
- 3’ b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fcvt.lu.d rd,fs1,rmm.
- 3’ b101: This code is reserved and not used..
- 3’ b110: This code is reserved and not used..
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fcvt.lu.d rd, fs1.

Instruction format:



16.5.10 FCVT.S.D——The Instruction to Convert a Double-Precision Floating-Point Number to a Single-Precision Floating-Point Number

Syntax:

fcvt.s.d fd, fs1, rm

Operation:

fd ← double_convert_to_single(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

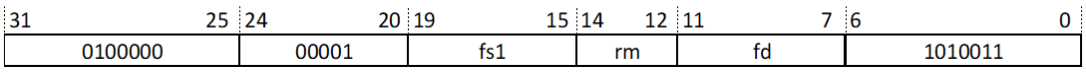
Floating-point state: NV/OF/UF/NX

Note:

rm determins the rounding mode:

- 3’ b000: Rounding to the nearest number, corresponding to the assembly instruction: fcvt.s.d fd,fs1,rne.
- 3’ b001: Rounding to zero, corresponding to the assembly instruction: fcvt.s.d fd,fs1,rtz.
- 3’ b010: Rounding to negative infinity. And the corresponding assembly instruction is: fcvt.s.d fd,fs1,rdn.
- 3’ b011: Rounding to positive infinity. And the corresponding assembly instruction is: fcvt.s.d fd,fs1,rup.
- 3’b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fcvt.s.d fd,fs1,rmm.
- 3’ b101: This code is reserved and not used..
- 3’ b110: This code is reserved and not used..
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fcvt.s.d fd, fs1.

Instruction format:



16.5.11 FCVT.W.D——The Instruction to Convert a Double-Precision Floating-Point Number to a Signed Integer

Syntax:

fcvt.w.d rd, fs1, rm

Operation:

tmp ← double_convert_to_signed_int(fs1)

rd←sign_extend(tmp)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

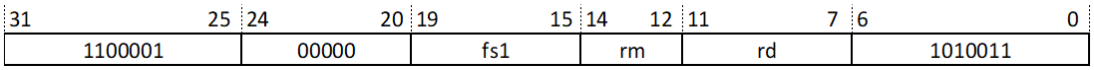
Floating-point state: NV/NX

Note:

rm determins the rounding mode:

- 3' b000: Rounding to the nearest number, corresponding to the assembly instruction: fcvt.w.d rd,fs1,rne.
- 3' b001: Rounding to zero, corresponding to the assembly instruction: fcvt.w.d rd,fs1,rtz.
- 3' b010: Rounding to negative infinity. And the corresponding assembly instruction is: fcvt.w.d rd,fs1,rdn.
- 3' b011: Rounding to positive infinity. And the corresponding assembly instruction is: fcvt.w.d rd,fs1,rup.
- 3' b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fcvt.w.d rd,fs1,rmm.
- 3' b101: This code is reserved and not used..
- 3' b110: This code is reserved and not used..
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fcvt.w.d rd, fs1.

Instruction format:



16.5.12 FCVT.WU.D——The Instruction to Convert a Double-Precision Floating-Point Number to an Unsigned Integer

Syntax:

fcvt.wu.d rd, fs1, rm

Operation:

tmp ← double_convert_to_unsigned_int(fs1)

rd←sign_extend(tmp)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

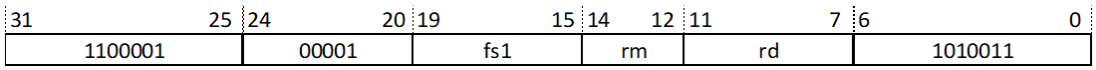
Floating-point state: NV/NX

Note:

rm determins the rounding mode:

- 3’ b000: Rounding to the nearest number, corresponding to the assembly instruction: fcvt.wu.d rd,fs1,rne.
- 3’ b001: Rounding to zero, corresponding to the assembly instruction: fcvt.wu.d rd,fs1,rtz.
- 3’ b010: Rounding to negative infinity. And the corresponding assembly instruction is: fcvt.wu.d rd,fs1,rdn.
- 3’ b011: Rounding to positive infinity. And the corresponding assembly instruction is: fcvt.wu.d rd,fs1,rup.
- 3’ b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fcvt.wu.d rd,fs1,rmm.
- 3’ b101: This code is reserved and not used..
- 3’ b110: This code is reserved and not used..
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fcvt.wu.d rd, fs1.

Instruction format:



16.5.13 FDIV.D——Double-Precision Floating-Point Division Instruction

Syntax:

fdiv.d fd, fs1, fs2, rm

Operation:

fd ← fs1 / fs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point state: NV/DZ/OF/UF/NX

Note:

rm determins the rounding mode:

- 3’ b000: Rounding to the nearest number, corresponding to the assembly instruction: fdiv.d fd, fs1,fs2,rne.

- 3’ b001: Rounding to zero, corresponding to the assembly instruction: fdiv.d fd, fs1,fs2,rtz.
- 3’ b010: Rounding to negative infinity. And the corresponding assembly instruction is: fdiv.d fd, fs1,fs2,rdn.
- 3’ b011: Rounding to positive infinity. And the corresponding assembly instruction is: fdiv.d fd, fs1,fs2,rup.
- 3’ b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fdiv.d fd, fs1,fs2,rmm.
- 3’ b101: This code is reserved and not used..
- 3’ b110: This code is reserved and not used..
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fdiv.d fd, fs1,fs2.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0001101	fs2	fs1	rm	fd	1010011

16.5.14 FEQ.D——The Compare-if-equal-to Instruction of Double-Precision Floating-Point Numbers

Syntax:

feq.d rd, fs1, fs2

Operation:

if(fs1 == fs2)

rd ← 1

else

rd ← 0

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point state: NV

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1010001	fs2	fs1	010	rd	1010011

16.5.15 FLD——The Double-Precision Floating-Point Load Instruction

Syntax:

fld fd, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

fd[63:0] ← mem[(address+7):address]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Affected flag:

None

Instruction format:

31 20	19 15	14 12	11 7	6 0
imm12[11:0]	rs1	011	fd	0000111

16.5.16 FLE.D——The Compare-if-less-than-or-equal-to Instruction of Double-Precision Floating-Point Numbers

Syntax:

fle.d rd, fs1, fs2

Operation:

if(fs1 <= fs2)

rd ← 1

else

rd ← 0

Execute permission:

M-mode/S-mode/U-mode

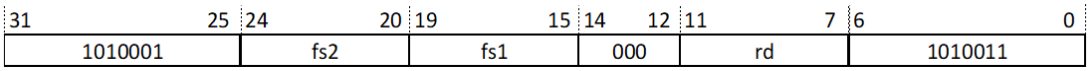
Exception:

The illegal instruction exception

Affected flag:

Floating-point state: NV

Instruction format:



16.5.17 FLT.D——The Compare-if-less-than Instruction of Double-Precision Floating-Point Numbers

Syntax:

flt.d rd, fs1, fs2

Operation:

if(fs1 < fs2)

rd ← 1

else

rd ← 0

Execute permission:

M-mode/S-mode/U-mode

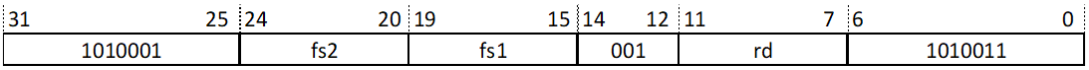
Exception:

The illegal instruction exception

Affected flag:

Floating-point state: NV

Instruction format:



16.5.18 FMADD.D——The Double-Precision Floating-Point Multiply-add Instruction

Syntax:

fmadd.d fd, fs1, fs2, fs3, rm

Operation:

fd ← fs1*fs2 + fs3

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

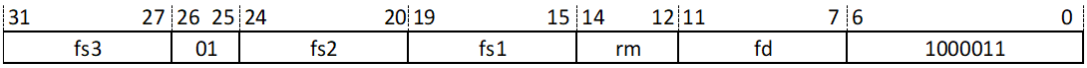
Floating-point state: NV/OF/UF/IX

Note:

rm determins the rounding mode:

- 3’ b000: Rounding to the nearest number, corresponding to the assembly instruction: fmadd.d fd,fs1, fs2, fs3, rne.
- 3’ b001: Rounding to zero, corresponding to the assembly instruction: fmadd.d fd,fs1, fs2, fs3, rtz.
- 3’ b010: Rounding to negative infinity. And the corresponding assembly instruction is: fmadd.d fd,fs1, fs2, fs3, rdn.
- 3’ b011: Rounding to positive infinity. And the corresponding assembly instruction is: fmadd.d fd,fs1, fs2, fs3, rup.
- 3’ b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fmadd.d fd,fs1, fs2, fs3, rmm.
- 3’ b101: This code is reserved and not used..
- 3’ b110: This code is reserved and not used..
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fmadd.d fd,fs1, fs2, fs3.

Instruction format:



16.5.19 FMAX.D——The Double-Precision Floating-Point Maximum Instruction

Syntax:

fmax.d fd, fs1, fs2

Operation:

if(fs1 >= fs2)

fd ← fs1

else

fd ← fs2

Execute permission:

M-mode/S-mode/U-mode

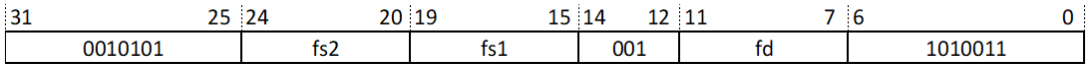
Exception:

The illegal instruction exception

Affected flag:

Floating-point state: NV

Instruction format:



16.5.20 FMIN.D——The Double-Precision Floating-Point Minimum Instruction

Syntax:

fmin.d fd, fs1, fs2

Operation:

if(fs1 >= fs2)

fd ← fs2

else

fd ← fs1

Execute permission:

M-mode/S-mode/U-mode

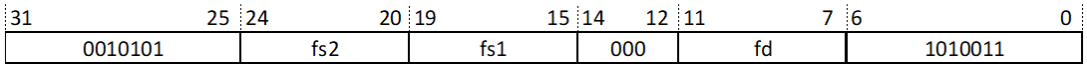
Exception:

The illegal instruction exception

Affected flag:

Floating-point state: NV

Instruction format:



16.5.21 FMSUB.D——The Double-Precision Floating-Point Multiply-subtract Instruction

Syntax:

fmsub.d fd, fs1, fs2, fs3, rm

Operation:

fd ← fs1*fs2 - fs3

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

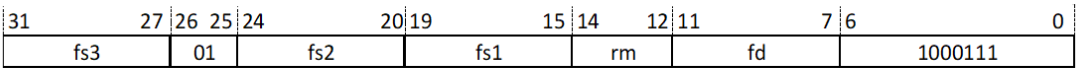
Floating-point state: NV/OF/UF/IX

Note:

rm determins the rounding mode:

- 3' b000: Rounding to the nearest number, corresponding to the assembly instruction: fmsub.d fd, fs1, fs2, fs3, rne.
- 3' b001: Rounding to zero, corresponding to the assembly instruction: fmsub.d fd, fs1, fs2, fs3, rtz.
- 3' b010: Rounding to negative infinity. And the corresponding assembly instruction is: fmsub.d fd, fs1, fs2, fs3, rdn.
- 3' b011: Rounding to positive infinity. And the corresponding assembly instruction is: fmsub.d fd, fs1, fs2, fs3, rup.
- 3' b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fmsub.d fd, fs1, fs2, fs3, rmm.
- 3' b101: This code is reserved and not used..
- 3' b110: This code is reserved and not used..
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fmsub.d fd, fs1, fs2, fs3.

Instruction format:



16.5.22 FMUL.D——The Double-Precision Floating-Point Multiply Instruction

Syntax:

fmul.d fd, fs1, fs2, rm

Operation:

fd ← fs1 * fs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

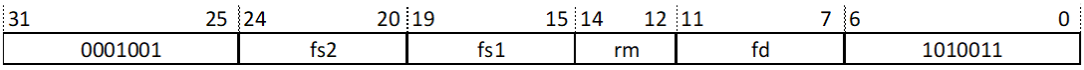
Floating-point state: NV/OF/UF/NX

Note:

rm determins the rounding mode:

- 3’ b000: Rounding to the nearest number, corresponding to the assembly instruction: fmul.d fd, fs1, fs2, rne.
- 3’ b001: Rounding to zero, corresponding to the assembly instruction: fmul.d fd, fs1, fs2, rtz.
- 3’ b010: Rounding to negative infinity. And the corresponding assembly instruction is: fmul.d fd, fs1, fs2, rdn.
- 3’ b011: Rounding to positive infinity. And the corresponding assembly instruction is: fmul.d fd, fs1, fs2, rup.
- 3’ b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fmul.d fd, fs1, fs2, rmm.
- 3’ b101: This code is reserved and not used..
- 3’ b110: This code is reserved and not used..
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fmul. fd, fs1,fs2.

Instruction format:



16.5.23 FMV.D.X——The Double-Precision Floating-Point Write Transfer Instruction

Syntax:

fmv.d.x fd, rs1

Operation:

fd← rs1

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

None

Note:

Transfer from integer register to floating-point register

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1111001	00000	rs1	000	fd	1010011

16.5.24 FMV.X.D——Double-Precision Floating-point Read Transfer Registers

Syntax:

fmv.x.d rd, fs1

Operation:

rd← fs1

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

None

Note:

Transfer from floating-point registers to integer registers

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1110001	00000	fs1	000	rd	1010011

16.5.25 FNMADD.D——The Double-Precision Floating-point Negate-(Multiply-add) Instruction

Syntax:

fnmadd.d fd, fs1, fs2, fs3, rm

Operation:

fd ←-(fs1*fs2 + fs3)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

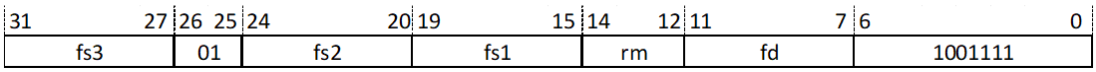
Floating-point state: NV/OF/UF/IX

Note:

rm determins the rounding mode:

- 3’ b000: Rounding to the nearest number, corresponding to the assembly instruction: fnmadd.d fd, fs1, fs2, fs3, rne.
- 3’ b001: Rounding to zero, corresponding to the assembly instruction: fnmadd.d fd, fs1, fs2, fs3, rtz.
- 3’ b010: Rounding to negative infinity. And the corresponding assembly instruction is: fnmadd.d fd, fs1, fs2, fs3, rdn.
- 3’ b011: Rounding to positive infinity. And the corresponding assembly instruction is: fnmadd.d fd, fs1, fs2, fs3, rup.
- 3’ b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fnmadd.d fd, fs1, fs2, fs3, rmm.
- 3’ b101: This code is reserved and not used..
- 3’ b110: This code is reserved and not used..
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fnmadd.d fd,fs1, fs2, fs3.

Instruction format:



16.5.26 FNMSUB.D——The Double-Precision Floating-point Negate-(Multiply-subtract) Instruction

Syntax:

fnmsub.d fd, fs1, fs2, fs3, rm

Operation:

fd ← -(fs1*fs2 - fs3)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

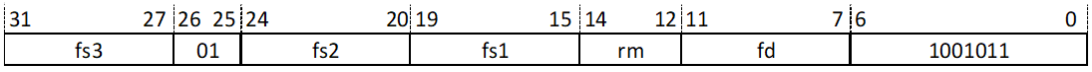
Floating-point state: NV/OF/UF/IX

Note:

rm determins the rounding mode:

- 3’ b000: Rounding to the nearest number, corresponding to the assembly instruction: fnmsub.d fd, fs1, fs2, fs3, rne.
- 3’ b001: Rounding to zero, corresponding to the assembly instruction: fnmsub.d fd, fs1, fs2, fs3, rtz.
- 3’ b010: Rounding to negative infinity. And the corresponding assembly instruction is: fnmsub.d fd, fs1, fs2, fs3, rdn.
- 3’ b011: Rounding to positive infinity. And the corresponding assembly instruction is: fnmsub.d fd, fs1, fs2, fs3, rup.
- 3’ b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fnmsub.d fd, fs1, fs2, fs3, rmm.
- 3’ b101: This code is reserved and not used..
- 3’ b110: This code is reserved and not used..
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fnmsub.d fd,fs1, fs2, fs3.

Instruction format:



16.5.27 FSD——The Double-Precision Floating-Point Store Instruction

Syntax:

fsd fs2, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

mem[(address+63):address] ← fs2[63:0]

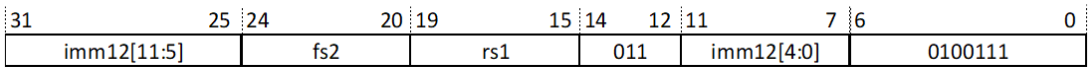
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



16.5.28 FSGNJ.D——The Double-Precision Floating-point Sign-injection Instruction

Syntax:

fsgnj.d fd, fs1, fs2

Operation:

fd[62:0] ← fs1[62:0]

fd[63] ← fs2[63]

Execute permission:

M-mode/S-mode/U-mode

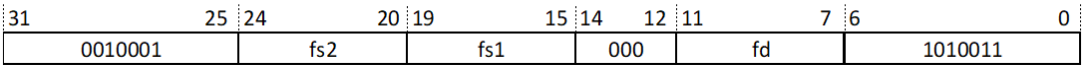
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



16.5.29 FSGNJN.D——The Double-Precision Floating-point Sign-injection Negate Instruction

Syntax:

fsgnjn.d fd, fs1, fs2

Operation:

fd[62:0] ← fs1[62:0]

fd[63] ← !fs2[63]

Execute permission:

M-mode/S-mode/U-mode

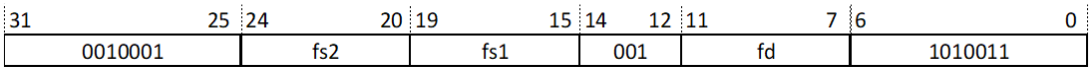
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



16.5.30 FSGNJX.D——The Double-Precision Floating-point Sign XOR Injection Instruction

Syntax:

fsgnjx.d fd, fs1, fs2

Operation:

fd[62:0] ← fs1[62:0]

fd[63] ← fs1[63] ^ fs2[63]

Execute permission:

M-mode/S-mode/U-mode

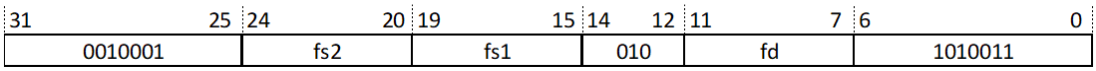
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



16.5.31 FSQRT.D——The Square Root Instruction of Double-Precision Floating-point

Syntax:

fsqrt.d fd, fs1, rm

Operation:

fd ← sqrt(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point state: NV/NX

Note:

rm determins the rounding mode:

- 3’ b000: Rounding to the nearest number, corresponding to the assembly instruction: fsqrt.d fd, fs1, rne.
- 3’ b001: Rounding to zero, corresponding to the assembly instruction: fsqrt.d fd, fs1, rtz.

- 3’ b010: Rounding to negative infinity. And the corresponding assembly instruction is: fsqrt.d fd, fs1, rdn.
- 3’ b011: Rounding to positive infinity. And the corresponding assembly instruction is: fsqrt.d fd, fs1, rup.
- 3’ b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fsqrt.d fd, fs1, rmm.
- 3’ b101: This code is reserved and not used..
- 3’ b110: This code is reserved and not used..
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fsqrt.d fd, fs1.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0101101	00000	fs1	rm	fd	1010011

16.5.32 FSUB.D——The Double-Precision Floating-point Subtract Instruction

Syntax:

fsub.d fd, fs1, fs2, rm

Operation:

fd ← fs1 - fs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point state: NV/OF/NX

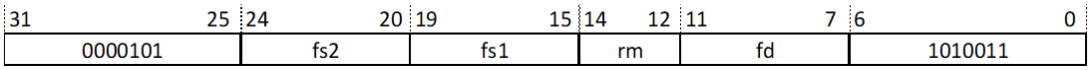
Note:

rm determins the rounding mode:

- 3’ b000: Rounding to the nearest number, corresponding to the assembly instruction: fsub.fd, fs1, fs2, rne.
- 3’ b001: Rounding to zero, corresponding to the assembly instruction: fsub.d fd, fs1, fs2, rtz.
- 3’ b010: Rounding to negative infinity. And the corresponding assembly instruction is: fsub.d fd, fs1, fs2, rdn.
- 3’ b011: Rounding to positive infinity. And the corresponding assembly instruction is: fsub.d fd, fs1, fs2, rup.
- 3’ b100: Rounding to the nearest large value. And the corresponding assembly instruction is:fsub.d fd, fs1, fs2, rmm.
- 3’ b101: This code is reserved and not used..
- 3’ b110: This code is reserved and not used..

- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is: fsub.dfd, fs1, fs2.

Instruction format:



16.6 Appendix A-6 C Instructions

This section describes the RISC-V C instructions implemented by C920. And the instructions are 16-bit wide, listed in alphabetic order.

16.6.1 C.ADD——The Signed Add Instruction

Syntax:

c.add rd, rs2

Operation:

rd ← rs1 + rs2

Execute permission:

Machine Mode (M-mode)/Supervisor Mode (S-mode)/User-mode (U-mode)

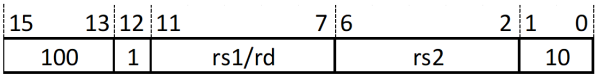
Exception:

None

Note:

- rs1 = rd != 0
- rs2 ! = 0

Instruction format:



16.6.2 C.ADDI——The Signed Immediate Add Instruction

Syntax:

c.addi rd, nzimm6

Operation:

rd ← rs1 + sign_extend(nzimm6)

Execute permission:

M-mode/S-mode/U-mode

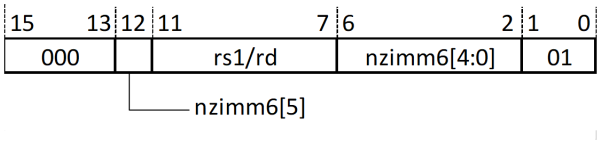
Exception:

None

Note:

- rs1 = rd != 0
- nzimm6!=0

Instruction format:



16.6.3 C.ADDIW——The Signed Immediate Add Instruction on the Lower 32 Bits

Syntax:

c.addiw rd, imm6

Operation:

tmp[31:0] ← rs1[31:0] + sign_extend(imm6)

rd ←sign_extend(tmp[31:0])

Execute permission:

M-mode/S-mode/U-mode

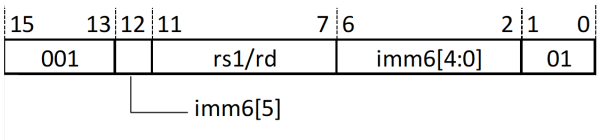
Exception:

None

Note:

rs1 = rd != 0

Instruction format:



16.6.4 C.ADDI4SPN——The Instruction to Add Immediate Scaled by 4 to Stack Pointer

Syntax:

c.addi4spn rd, sp, nzuimm8<<2

Operation:

rd ← sp + zero_extend(nzuimm8<<2)

Execute permission:

M-mode/S-mode/U-mode

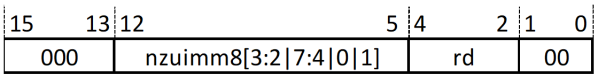
Exception:

None

Note:

- nzuimm8 != 0
- rd code represents the following registers:
  - 000 x8
  - 001 x9
  - 010 x10
  - 011 x11
  - 100 x12
  - 101 x13
  - 110 x14
  - 111 x15

Instruction format:



16.6.5 C.ADDI16SP——The Instruction to Add Immediate Scaled by 16 to Stack Pointer

Syntax:

c.addi16sp sp, nzuimm6<<4

Operation:

sp ← sp + sign_extend(nzuimm6<<4)

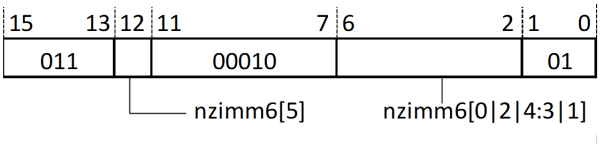
Execute permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:



16.6.6 C.ADDW——The Signed Add Instruction on the Lower 32 Bits

Syntax:

c.addw rd, rs2

Operation:

tmp[31:0] ← rs1[31:0] + rs2[31:0]

rd ←sign_extend(tmp[31:0])

Execute permission:

M-mode/S-mode/U-mode

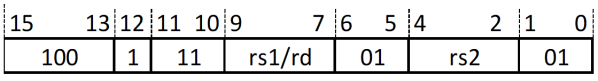
Exception:

None

Note:

- rs1 = rd
- rd/rs1, rs2 code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15

Instruction format:



16.6.7 C.AND——The Bitwise AND Instruction

Syntax:

c.and rd, rs2

Operation:

rd ← rs1 & rs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

None

Note:

- rs1 = rd
- rd/rs1, rs2 code represent the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15

Instruction format:

15 13	12	11 10	9 7	6 5	4 2	1 0
100	0	11	rs1/rd	11	rs2	01

16.6.8 C.ANDI——The Immediate Bitwise AND Instruction

Syntax:

c.andi rd, imm6

Operation:

rd ← rs1 & sign_extend(imm6)

Execute permission:

M-mode/S-mode/U-mode

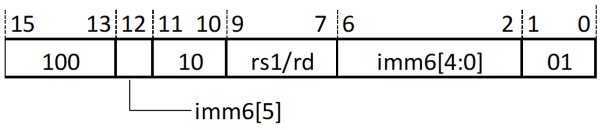
Exception:

None

Note:

- rs1 = rd
- rd/rs1 code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15

Instruction format:



16.6.9 C.BEQZ——The Branch-if-equal-to-zero Instruction

Syntax:

c.beqz rs1, label

Operation:

if (rs1 == 0)

next pc = current pc + imm8<<1;

else

next pc = current pc + 2;

Execute permission:

M-mode/S-mode/U-mode

Exception:

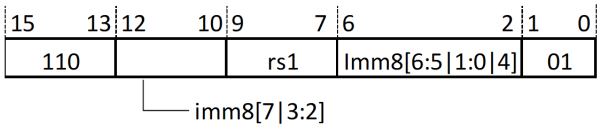
None

Note:

- rs1 code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10

  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15
- The assembler calculates imm8 based on the label.
- The instruction jump range is ±256B address space.

Instruction format:



16.6.10 C.BNEZ——The Branch-if-not-equal-to-zero Instruction

Syntax:

c.bnez rs1, label

Operation:

if (rs1 != 0)

    next pc = current pc + imm8<<1;

else

    next pc = current pc + 2;

Execute permission:

M-mode/S-mode/U-mode

Exception:

None

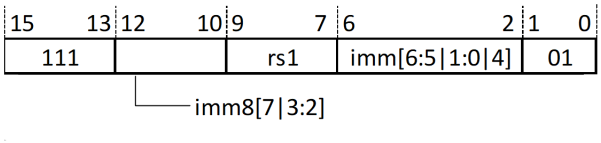
Note:

- rs1 code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14

– 111: x15

- The assembler calculates imm12 based on the label.
- The instruction jump range is ±256B address space.

Instruction format:



16.6.11 C.EBREAK——The Breakpoint Instruction

Syntax:

c.ebreak

Operation:

Generates breakpoint exceptions or enters the debug mode.

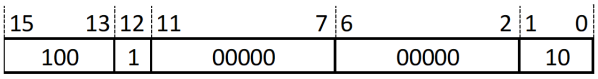
Execute permission:

M-mode/S-mode/U-mode

Exception:

Breakpoint exceptions

Instruction format:



16.6.12 C.FLD——The Floating-point Doubleword Load Instruction

Syntax:

c.fld fd, uimm5<<3(rs1)

Operation:

address ← rs1+ zero_extend(uimm5<<3)

fd ←mem[address+7:address]

Execute permission:

M-mode/S-mode/U-mode

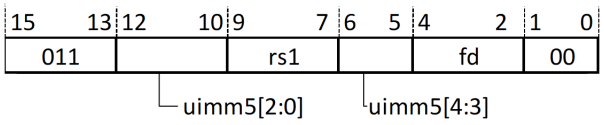
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions.

Note:

- rs1 code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15
- fd code represents the following registers:
  - 000: f8
  - 001: f9
  - 010: f10
  - 011: f11
  - 100: f12
  - 101: f13
  - 110: f14
  - 111: f15

Instruction format:



16.6.13 C.FLDSP——The Instruction to Load Floating-point Doubleword from a Stack

Syntax:

c.fldsp fd, uimm6<<3(sp)

Operation:

address ← sp+ zero_extend(uimm6<<3)

fd ←mem[address+7:address]

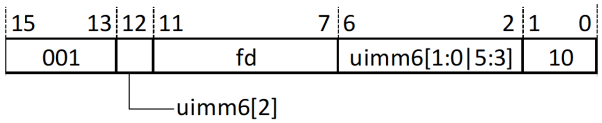
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions.

Instruction format:



16.6.14 C.FSD——The Instruction to Store Doubleword into a Stack

Syntax:

c.fsd fs2, uimm5<<3(rs1)

Operation:

address ← rs1+ zero_extend(uimm5<<3)

mem[address+7:address] ←fs2

Execute permission:

M-mode/S-mode/U-mode

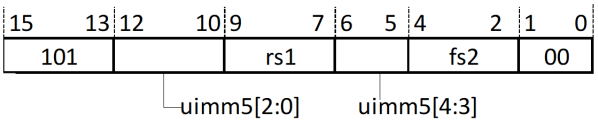
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions.

Note:

- fs1 code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15
- rs2 code represents the following registers:
  - 000: f8
  - 001: f9
  - 010: f10
  - 011: f11
  - 100: f12
  - 101: f13
  - 110: f14
  - 111: f15

Instruction format:



16.6.15 C.FSDSP——The Instruction to Store Floating-point Doubleword into a Stack

Syntax:

c.fsdsp fs2, uimm6<<3(sp)

Operation:

address ← sp+ zero_extend(uimm6<<3)

mem[address+7:address] ←fs2

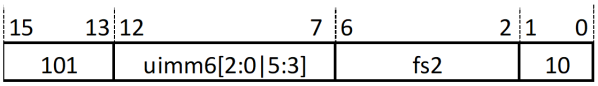
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions.

Instruction format:



16.6.16 C.J——The Unconditional Jump Instruction

Syntax:

c.j label

Operation:

next pc ← current pc + sign_extend(imm<<1);

Execute permission:

M-mode/S-mode/U-mode

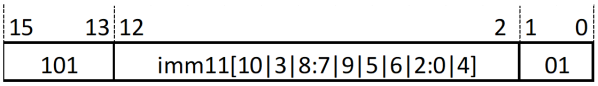
Exception:

None

Note:

- The assembler calculates imm11 based on the label.
- The instruction jump range is ±2KB address space.

Instruction format:



16.6.17 C.JALR——The Jump and Link Register Instruction

Syntax:

c.jalr rs1

Operation:

next pc ← rs1;

x1←current pc + 2;

Execute permission:

M-mode/S-mode/U-mode

Exception:

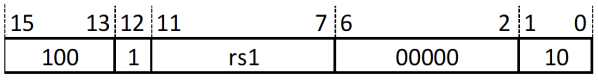
None

Note:

rs1 != 0

- When MMU is enabled, the jump range is the entire 512 GB address space.
- When MMU is disabled, the jump range is the entire 1 TB address space.

Instruction format:



16.6.18 C.JR——The Jump to Register Instruction

Syntax:

c.jr rs1

Operation:

next pc = rs1;

Execute permission:

M-mode/S-mode/U-mode

Exception:

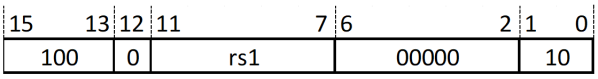
None

Note:

rs1 != 0

- When MMU is enabled, the jump range is the entire 512 GB address space.
- When MMU is disabled, the jump range is the entire 1 TB address space.

Instruction format:



16.6.19 C.LD——The Doubleword Load Instruction

Syntax:

c.ld rd, uimm5<<3(rs1)

Operation:

address ← rs1+ zero_extend(uimm5<<3)

rd ←mem[address+7:address]

Execute permission:

M-mode/S-mode/U-mode

Exception:

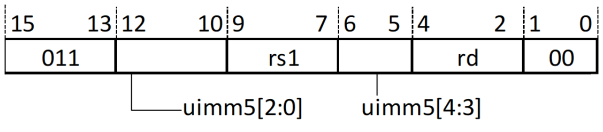
Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions.

Note:

rs1/rd code represents the following registers:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



16.6.20 C.LDSP——The Instruction to Load Doubleword from Stack

Syntax:

c.ldsp rd, uimm6<<3(sp)

Operation:

address ← sp+ zero_extend(uimm6<<3)

rd ←mem[address+7:address]

Execute permission:

M-mode/S-mode/U-mode

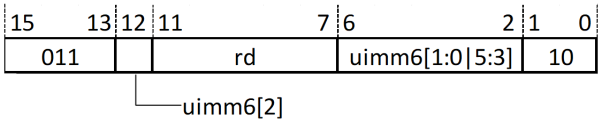
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions.

Note:

rd != 0

Instruction format:



16.6.21 C.LI——The Immediate Transfer Instruction

Syntax:

c.li rd, imm6

Operation:

rd ←sign_extend(imm6)

Execute permission:

M-mode/S-mode/U-mode

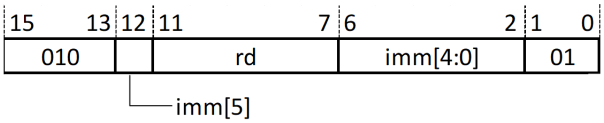
Exception:

None

Note:

rd != 0

Instruction format:



16.6.22 C.LUI——The Upper Bit Immediate Transfer Instruction

Syntax:

c.lui rd, nzimm6

Operation:

rd ←sign_extend(nzimm6<<12)

Execute permission:

M-mode/S-mode/U-mode

Exception:

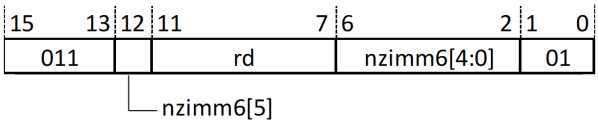
None

Note:

rd != 0

Nzimm6 != 0

Instruction format:



16.6.23 C.LW——The Word Load Instruction

Syntax:

c.lw rd, uimm5<<2(rs1)

Operation:

address ← rs1+ zero_extend(uimm5<<2)

tmp[31:0] ←mem[address+3:address]

rd ←sign_extend(tmp[31:0])

Execute permission:

M-mode/S-mode/U-mode

Exception:

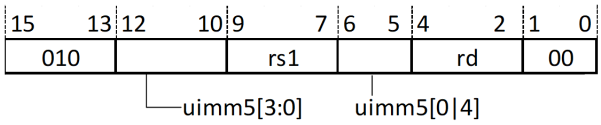
Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions.

Note:

rs1/rd code represents the following registers:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



16.6.24 C.LWSP——The Load Word from Stack Pointer Instruction

Syntax:

c.lwsp rd, uimm6<<2(sp)

Operation:

address ← sp+ zero_extend(uimm6<<2)

tmp[31:0] ←mem[address+3:address]

rd ←sign_extend(tmp[31:0])

Execute permission:

M-mode/S-mode/U-mode

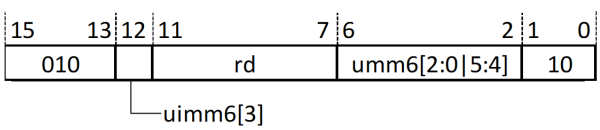
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions.

Note:

rd != 0

Instruction format:



16.6.25 C.MV——The Data Transfer Instruction

Syntax:

c.mv rd, rs2

Operation:

rd ← rs2;

Execute permission:

M-mode/S-mode/U-mode

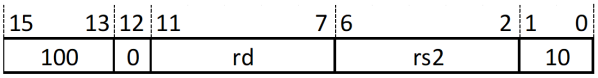
Exception:

None

Note:

rs2 != 0, rd !=0

Instruction format:



16.6.26 C.NOP——The No-operation Instruction

Syntax:

c.nop

Operation:

No operation

Execute permission:

M-mode/S-mode/U-mode

Exception:

None

Instruction format:

16.6.27 C.OR——The Bitwise OR Instruction

Syntax:

c.or rd, rs2

Operation:

rd ← rs1 | rs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

None

Note:

- rs1 = rd
- rd/rs1 code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15

Instruction format:

15 13	12	11 10	9 7	6 5	4 2	1 0
100	0	11	rs1/rd	10	rs2	01

16.6.28 C.SD——The Doubleword Store Instruction

Syntax:

c.sd rs2, uimm5<<3(rs1)

Operation:

address ← rs1+ zero_extend(uimm5<<3)

mem[address+7:address] ←rs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions.

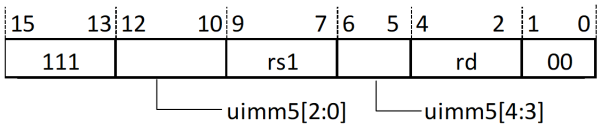
Note:

rs1/rd code represents the following registers:

- 000: x8

- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



16.6.29 C.SDSP——The Instruction to Store Doubleword into a Stack

Syntax:

c.fsdsp rs2, uimm6<<3(sp)

Operation:

address ← sp+ zero_extend(uimm6<<3)

mem[address+7:address] ←rs2

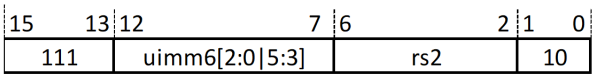
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions.

Instruction format:



16.6.30 C.SLLI——The Immediate Logical Left Shift Instruction

Syntax:

c.slli rd, nzuimm6

Operation:

rd ←rs1 << nzuimm6

Execute permission:

M-mode/S-mode/U-mode

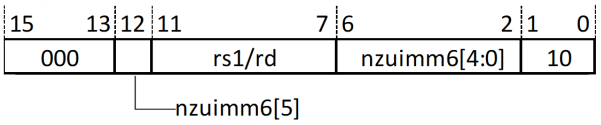
Exception:

None

Note:

- rs1==rd
- rd/rs1 != 0, nzuimm6 != 0

Instruction format:



16.6.31 C.SRAI——The Immediate Arithmetic Right Shift Instruction

Syntax:

c.srli rd, nzuimm6

Operation:

rd ←rs1 >>>nzuimm6

Execute permission:

M-mode/S-mode/U-mode

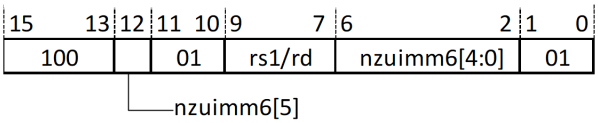
Exception:

None

Note:

- nzuimm6 != 0
- rs1 == rd
- rs1/rd code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15

Instruction format:



16.6.32 C.SRLI——The Immediate Logical Right Shift Instruction

Syntax:

c.srli rd, nzuimm6

Operation:

rd ←rs1 >> nzuimm6

Execute permission:

M-mode/S-mode/U-mode

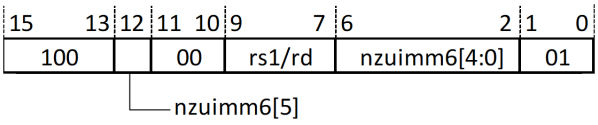
Exception:

None

Note:

- nzuimm6 != 0
- rs1 == rd
- rs1/rd code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15

Instruction format:



16.6.33 C.SW——The Word Store Instruction

Syntax:

c.sw rs2, uimm5<<2(rs1)

Operation:

address ← rs1+ zero_extend(uimm5<<2)

mem[address+3:address] ←rs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

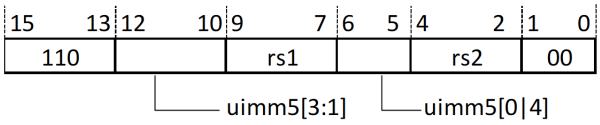
Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions.

Note:

rs1/rs2 code represents the following registers:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



16.6.34 C.SWSP——The Word Stack Store Instruction

Syntax:

c.swsp rs2, uimm6<<2(sp)

Operation:

address ← sp+ zero_extend(uimm6<<2)

mem[address+3:address] ←rs2

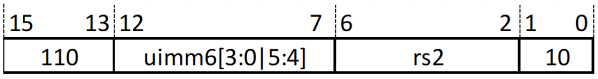
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions.

Instruction format:



16.6.35 C.SUB——The Signed Subtract Instruction

Syntax:

c.sub rd, rs2

Operation:

rd ← rs1 - rs2

Execute permission:

M-mode/S-mode/U-mode

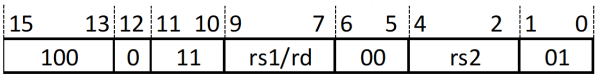
Exception:

None

Note:

- rs1 == rd
- rs1/rd code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15

Instruction format:



16.6.36 C.SUBW——The Signed Subtract Instruction on the Lower 32 Bits

Syntax:

c.subw rd, rs2

Operation:

tmp[31:0] ← rs1[31:0] - rs2[31:0]

rd ←sign_extend(tmp)

Execute permission:

M-mode/S-mode/U-mode

Exception:

None

Note:

- rs1 == rd
- rs1/rd code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10
  - 011: x11
  - 100: x12
  - 101: x13
  - 110: x14
  - 111: x15

Instruction format:

15 13	12	11 10	9 7	6 5	4 2	1 0
100	1	11	rs1/rd	00	rs2	01

16.6.37 C.XOR——The Bitwise XOR Instruction

Syntax:

c.xor rd, rs2

Operation:

rd ← rs1 ^ rs2

Execute permission:

M-mode/S-mode/U-mode

Exception:

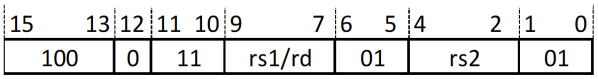
None

Note:

- rs1 == rd
- rs1/rd code represents the following registers:
  - 000: x8
  - 001: x9
  - 010: x10

- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



16.7 Appendix A-8 Pseudo Instruction List

RISC-V implements a series of pseudo instructions, listed in this section for reference only and sorted in alphabetic order.

Table 16.1: **RISC-V Pseudo Instruction List**

Pseudo Instruction	Base Instruction	Description
beqz rs, offset	beq rs, x0, offset	Take the branch jump if the value in the rs register is zero.
bnez rs, offset	bne rs, x0, offset	Take the branch jump if the value in the rs register is not zero.
blez rs, offset	bge x0,rs, offset	Take the branch jump if the value in the rs register is less than or equal to zero.
bgez rs, offset	bge rs, x0, offset	Take the branch jump if the value in the rs register is greater than or equal to zero.
bltz rs, offset	blt rs, x0, offset	Take the branch jump if the value in the rs register is less than zero.
bgtz rs, offset	blt x0, xs, offset	Take the branch jump if the value in the rs the register is greater than zero.
bgt rs, rt, offset	blt rt, rs, offset	Take the branch jump if the value in the rs register is greater than that of the rt register.
ble rs, rt, offset	bge rt, rs, offset	Take the branch jump if the value in the rs register is less than or equal to that of the rt register.
bgtu rs, rt, offset	bltu rt, rs, offset	Takes the branch jump if the value in the rs register is greater than that of the rt register, using unsigned comparison.
bleu rs, rt, offset	bgeu rt, rs, offset	Takes the branch jump if the value in the rs register is less than or equal to that of the rt register, using unsigned comparison.
call offset	auipc x6, offset[31:12] jalr x1, x6, offset[11:0]	Function Jump within a 4-Kilobyte to 4-Gigabyte Address Range

Continued on next page

Table 16.1 – continued from previous page

Pseudo Instruction	Base Instruction	Description
csrc csr, rs	csrrc x0, csr, rs	Clear the corresponding bits in the control/status register (CSR)
csrci csr, imm	csrrci x0, csr, imm	Clear the corresponding bits in the lower 5 bits of the CSR
csrs csr, rs	csrrs x0, csr, rs	Set the corresponding bits in the CSR
csrsi csr, imm	csrrsi x0, csr, imm	Set the corresponding bits in the lower 5 bits of the CSR
csrw csr, rs	csrrw x0, csr, rs	Write the corresponding bits in the CSR
csrwi csr, imm	csrrwi x0, csr, imm	Write the corresponding bits in the lower 5 bits of the CSR
fabs.d rd , rs	fsgnjx.d rd, rs,rs	Take the absolute value of a double-precision Floating-point (FP) number.
fabs.s rd , rs	fsgnjx.s rd, rs,rs	Take the absolute value of a single-precision FP number.
fence	fence iorw, iorw	The synchronization instruction between memory and device
fl{w|d} rd, symbol, rt	auipc rt, symbol[31:12] fl{w|d} rd, symbol[11:0](rt)	The FP load instruction for a 4GB address space
fmv.d rd, rs	fsgnj.d rd, rs,rs	The copy instruction of a double-precision FP
fmv.s rd, rs	fsgnj.s rd, rs, rs	The copy instruction of a single-precision FP
fneg.d rd, rs	fsgnjn.d rd, rs,rs	The negate instruction of a double-precision FP
fneg.s rd, rs	fsgnjn.s rd, rs, rs	The negate instruction of a single-precision FP
frcsr rd	csrrs x0, fcsr, x0	Read the instruction from the FP CSR
frflags rd	csrrs rd, fflags, x0	Read the instruction in the FP exception flag
frrm rd	csrrs rd, frm, x0	Read the instruction in the FP rounding bit
fscsr rs	csrrw x0, fcsr, rs	Write the instruction from FP CSR
fscsr rd, rs	csrrs rd, fcsr, rs	Read and write the instruction from the FP CSR
fsflags rs	csrrw x0, fcsr, rs	Write the instruction in the FP exception flag
fsflags rd, rs	csrrs rd, fcsr, rs	Read and write the instruction in the FP exception flag
fsflagsi imm	csrrwi x0, fflags, imm	The instruction to write a specified immediate number into the FP exception flag
fsflagsi rd, imm	csrrwi rd, fflags, imm	The instruction to read and write the value of the FP exception flag by the immediate
fsrm rs	csrrw x0, frm, rs	The instruction to write a specific value into the FP rounding mode
fsrm rd, rs	csrrs rd, frm, rs	The instruction to read and write FP rounding mode

Continued on next page

Table 16.1 – continued from previous page

Pseudo Instruction	Base Instruction	Description
fsrmi imm	csrrwi x0, frm, imm	The instruction to write an immediate value to the FP rounding mode
fsrmi rd, imm	csrrwi rd, frm, imm	The instruction to read and write the value of the FP rounding mode by the immediate
fs{w|d} rd, symbol,rt	auipc rt,symbol[31:12] fs{w|d} rd, symbol[11:0](rt)	The inistruction to store FP numbers in a 4GB address space
j offset	jal x0, offset	The direct jump instruction
jal offset	jal x1, offset	The subroutine jump and link instruction
jalr rs	jalr x1, rs, 0	The instructions of subroutine jump register and link register
jr rs	jalr x0, rs, 0	The jump to register instruction
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	The instruction to load address
li rd, immediate	Split into multiple instructions based on the size of the immediate	The instruction to load immediate
l{b|h|w|d} rd,symbol, rt	auipc rt, symbol[31:12] l{b|h|w|d} rd, symbol[11:0](rt)	The load instruction in 4GB address space
mv rd, rs	addi rd, rs, 0	The data transfer instruction
neg rd, rs	sub rd, x0, rs	The negate value from register and store
negw rd, rs	subw rd, x0, rs	The instuction to take negative of the lower 32 bits of a register
nop	addi x0,x0,0	No operation
not rd, rs	xori rd, rs, -1	The bitwise complement register instruction
rdcycle[h] rd	csrrs rd, cycle[h], x0	The instruction to read the cycle counter
rdinstret[h] rd	csrrs rd, instret[h], x0	The instruction count retrieval instruction
rdtime[h] rd	csrrs rd, time[h], x0	The Real-time clock retrieval instruction
ret	jalr x0, x1,0	Return from the subroutine
s{b|h|w|d} rd, symbol, rt	auipc rt,symbol[31:12] s{b|h|w|d} rd,symbol[11:0](rt)	Store the 4 GiB address space
seqz rd, rs	sltiu rd, rs, 1	Set the register value 0 to 1
sextw rd, rs	addiw rd, rs, 0	The sign extension instruction
sgtz rd, rs	slt rd, rs, x0, rs	The instruction to set the if-greater-than-zero register value to 1
sltz rd, rs	slt rd, rs, rs, x0	The instruction to set the if-smaller-than-zero register value to 1
snez rd, rs	sltu rd, rs, x0, rs	The instruction to set the non-zero register value to 1
tail offset	auipc x6,offset[31:12] jalr x0, x6,offset[11:0]	Not link or jump to the subroutine

CHAPTER 17

Appendix B Xuantie Extended Instructions

Apart from the standard defined GCV instruction set, C920 implements custom instruction sets, including the subsets of Cache instructions, synchronization instructions, arithmetic operation instructions, bitwise operation instructions, store instructions, and floating-point half-precision instructions.

Among these instruction subsets, the subsets of Cache instructions, synchronization instructions, arithmetic operation instructions, bitwise operation instructions, and store instructions can be executed normally only when mxstatus.theadisaee == 1. Otherwise, instruction exceptions will occur; The floating-point half-precision instruction subset can be executed normally only when mstatus.fs != 2' b00. Otherwise, illegal instruction exceptions will occur. This section specifically describes each instruction according to the different instruction subset extensions.

17.1 Appendix B-1 Cache Instructions

Cache instruction subset has implemented the cache operation with 32-bit width for each instruction.

The following instructions are listed in alphabetic order.

17.1.1 DCACHE.CALL——The Instruction that Clears All Dirty Table Entries in the D-Cache

Syntax:

dcache.call

Operation:

Clears all table entries in the L1 Data Cache (D-Cache) and writes all dirty table entries back into the next-level store, operating only on the current core.

Execute permission:

Machine Mode (M-mode)/Supervisor Mode (S-mode)

Exception:

The illegal instruction exception

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in User Mode (U-mode) causes an illegal instruction exception.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0000000	00001	00000	000	00000	0001011

17.1.2 DCACHE.CIALL——The Instruction to Clear All Dirty Table Entries in the D-Cache and Invalidates the D-Cache

Syntax:

dcache.ciall

Operation:

Writes all dirty table entries in the L1 D-Cache back into the next-level store and invalidate all these table entries.

Execute permission:

M-mode/S-mode

Exception:

The illegal instruction exception

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0000000	00011	00000	000	00000	0001011

17.1.3 DCACHE.CIPA——The Instruction to Clear Dirty Table Entries by Physical Addresses in the D-Cache and Invalidates the D-Cache

Syntax:

dcache.cipa rs1

Operation:

Writes the D-Cache/L2 Cache table entries corresponding to the physical addresses in rs1 back into the next-level store and invalidate these table entries, operating on all cores and the L2 Cache.

Execute permission:

M-mode/S-mode

Exception:

The illegal instruction exception

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0000001	01011	rs1	000	00000	0001011

17.1.4 DCACHE.CISW——The Instruction to Clear Dirty Table Entries in the D-Cache by the Specified Way/Set and Invalidates the D-Cache

Syntax:

dcache.cisw rs1

Operation:

Writes the dirty L1 D-Cache table entry that matches the specified way/set in rs1 back into the next-level store and invalidate this table entry, operating only on the current core.

Execute permission:

M-mode/S-mode

Exception:

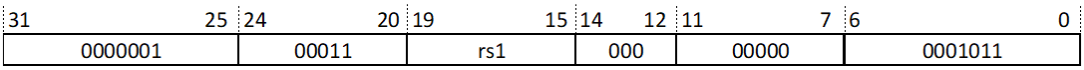
The illegal instruction exception

Note:

C920 D-Cache is a 2-way set-associative cache, where rs1[31] is the way encoding and rs1[w:6] is the set encoding. When the size of the dcache is 32K, the value of w is 13, and when the dcache size is 64K, the value of w is 14.

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



17.1.5 DCACHE.CIVA——The Instruction to Clear Dirty Table Entries by Virtual Addresses in the D-Cache and Invalidates the D-Cache

Syntax:

dcache.civa rs1

Operation:

Writes the dcache/L2 cache table entry belonging to the specified virtual address in rs1 back into the next-level store and invalidate this table entry, operating on the current core and the L2 Cache. And the sharing attribute of the virtual address determines whether to broadcast to other cores.

Execute permission:

M-mode/S-mode/U-mode

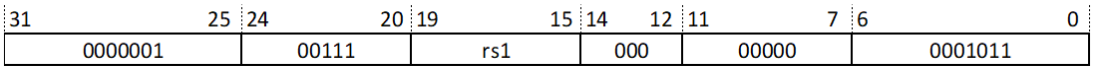
Exception:

The illegal instruction exception/The exception to load instruction pages

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, mxstatus.ucme =1, this instruction can be executed in U-mode.
- mxstatus.theadisaee=1, mxstatus.ucme =0, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



17.1.6 DCACHE.CPA——The Instruction to Clear Dirty Table Entries by Physical Addresses in D-CACHE

Syntax:

dcache.cpa rs1

Operation:

Writes the dcache/l2cache table entry corresponding to the physical address in rs1 back to the next level of store, operating on all cores and the L2 cache.

Execute permission:

M-mode/S-mode

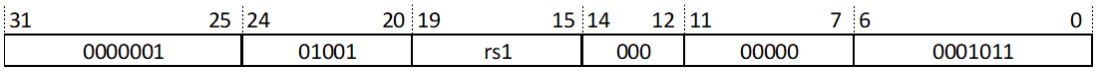
Exception:

The illegal instruction exception

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
0000001	01001	rs1	000	00000	0001011

17.1.7 DCACHE.CPAL1——The Instruction to Clear Dirty Table Entries by Physical Addresses in L1 D-CACHE

Syntax:

dcache.cpal1 rs1

Operation:

Writes the dcache table entry that matches the specified physical address in rs1 back into the next-level store, operating on all cores and the L1 Cache.

Execute permission:

M-mode/S-mode

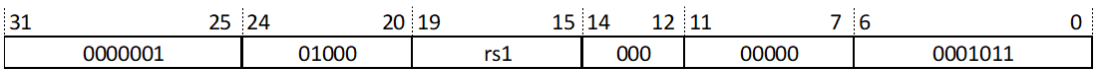
Exception:

The illegal instruction exception

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
0000001	01000	rs1	000	00000	0001011

17.1.8 DCACHE.CVA——The Instruction to Clear Dirty Table Entries by Virtual Addresses in D-CACHE

Syntax:

dcache.cva rs1

Operation:

Writes the dcache/L2 cache table entry belonging to the specified virtual address in rs1 back into the next-level store, operating on the current core and L2CACHE. And the sharing attribute of the virtual address determines whether to broadcast to other cores.

Execute permission:

M-mode/S-mode

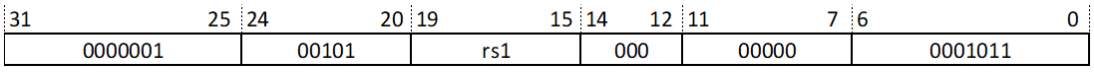
Exception:

The illegal instruction exception/The exception to load instruction pages

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, mxstatus.ucme=1, the instruction can be executed in U-mode.
- mxstatus.theadisaee=1, mxstatus.ucme=0, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



17.1.9 DCACHE.CVAL1——The Instruction to Clear Dirty Table Entries by Virtual Addresses in L1 D-CACHE

Syntax:

dcache.cval1 rs1

Operation:

Writes the D-Cache table entry that matches the specified virtual addres in s1 back into the next-level store, operating on all cores and the L1 Cache.

Execute permission:

M-mode/S-mode/U-mode

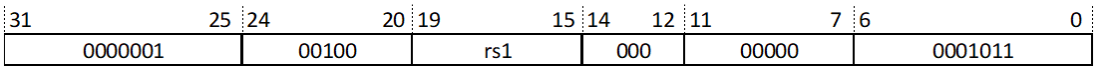
Exception:

The illegal instruction exception/The exception to load instruction pages

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, mxstatus.ucme =0, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
0000001	00100	rs1	000	00000	0001011

17.1.10 DCACHE.IPA——The DCACHE Invalid Instruction by Physical Addresses

Syntax:

dcache.ipa rs1

Operation:

Invalidates the dcache/l2 cache table entry that matches the specified physical address in rs1, operating on all cores and the L2 Cache.

Execute permission:

M-mode/S-mode

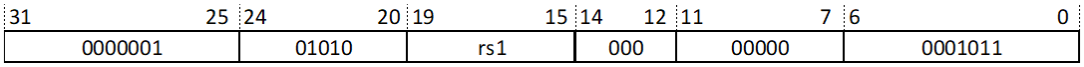
Exception:

The illegal instruction exception

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



31 25	24 20	19 15	14 12	11 7	6 0
0000001	01010	rs1	000	00000	0001011

17.1.11 DCACHE.ISW——The DCACHE Invalidation Instruction by Set/Way

Syntax:

dcache.isw rs1

Operation:

Invalidates the D-Cache table entries specified by SET and WAY, operating on the current core.

Execute permission:

M-mode/S-mode

Exception:

The illegal instruction exception

Note:

C920 D-Cache is a 2-way set-associative cache, where rs1[31] is the way encoding and rs1[w:6] is the set encoding. When the size of the dcache is 32K, the value of w is 13, and when the dcache size is 64K, the value of w is 14.

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0000001	00010	rs1	000	00000	0001011

17.1.12 DCACHE.IVA——The DCACHE Invalidation Instruction by Virtual Addresses

Syntax:

dcache.iva rs1

Operation:

Invalidates the dcache/l2 cache table entry that matches the specified virtual address in rs1, operating on the current core and L2CACHE. And the sharing attribute of the virtual address determines whether to broadcast to other cores.

Execute permission:

M-mode/S-mode

Exception:

The illegal instruction exception/The exception to load instruction pages

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0000001	00110	rs1	000	00000	0001011

17.1.13 DCACHE.IALL——The Instruction to Invalidate All Table Entries in the D-Cache

Syntax:

dcache.iall

Operation:

Invalidate all table entries in the L1 D-Cache, operating on the current core.

Execute permission:

M-mode/S-mode

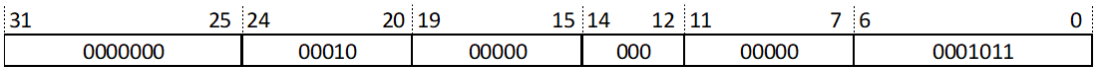
Exception:

The illegal instruction exception

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



17.1.14 ICACHE.IALL——The Instruction to Invalidate All Table Entries in the I-Cache

Syntax:

icache.iall

Operation:

Invalidates all table entries in the I-Cache, operating on the current core.

Execute permission:

M-mode/S-mode

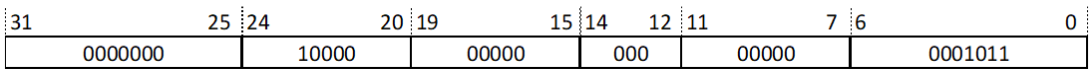
Exception:

The illegal instruction exception

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



17.1.15 ICACHE.IALLS——The Instruction to Invalidate All Table Entries in the I-Cache through Broadcasting

Syntax:

icache.ialls

Operation:

Invalidates all table entries in the I-Cache and other cores invalidate all their respective table entries in I-Cache through broadcasting, operating on all cores.

Execute permission:

M-mode/S-mode

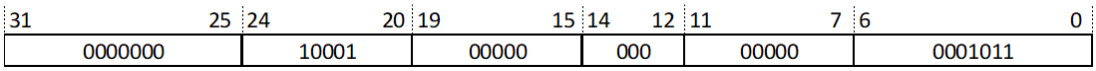
Exception:

The illegal instruction exception

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



17.1.16 ICACHE.IPA——The Instruction to Invalidate Table Entries by Physical Addresses in the I-Cache

Syntax:

icache.ipa rs1

Operation:

Invalidates the I-Cache table entry that matches the specified physical address in rs1, operating on all cores.

Execute permission:

M-mode/S-mode

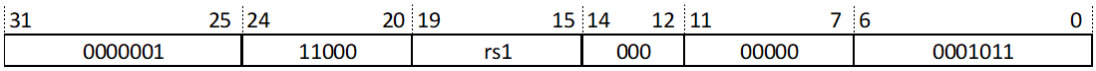
Exception:

The illegal instruction exception

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



17.1.17 ICACHE.IVA——The Instruction to Invalidate Table Entries by Virtual Addresses in the I-Cache

Syntax:

icache.iva rs1

Operation:

Invalidates the I-Cache table entry that matches the specified virtual address in rs1, operating on the current core. And the sharing attribute of the virtual address determines whether to broadcast to other cores.

Execute permission:

M-mode/S-mode/U-mode

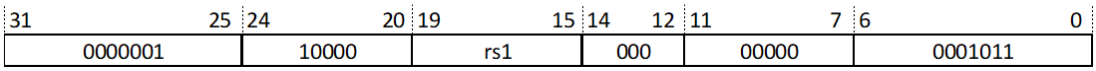
Exception:

The illegal instruction exception/The exception to load instruction pages

Note:

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, mxstatus.ucme=1, U-mode supports executing the instruction.
- mxstatus.theadisaee=1, mxstatus.ucme=0, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:



17.1.18 DCACHE.CSW——The Instruction to Clear Dirty Table Entries in the D-Cache by Set/Way

Syntax:

dcache.csw rs1

Operation:

Writes the the dirty table entry from the D-Cache back into the next-level store device based on the specified SET and WAY.

Execute permission:

M-mode/S-mode

Exception:

The illegal instruction exception

Note:

C920 D-Cache is a 2-way set-associative cache, where rs1[31] is the way encoding and rs1[w:6] is the set encoding. When the size of the dcache is 32K, the value of w is 13, and when the dcache size is 64K, the value of w is 14.

- mxstatus.theadisaee=0, executing this instruction causes an illegal instruction exception.
- mxstatus.theadisaee=1, executing this instruction in U-mode causes an illegal instruction exception.

Instruction format:

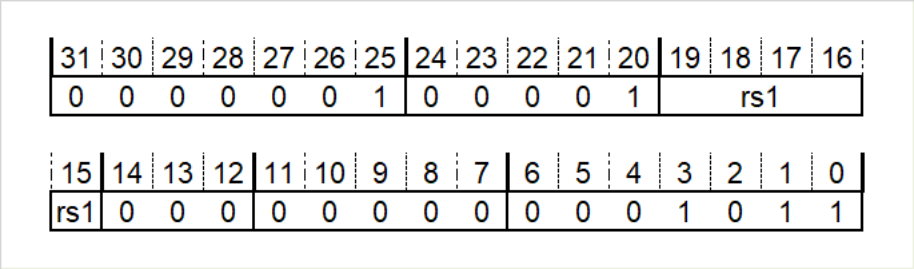


Fig. 17.1: **DCACHE.CSW**

17.2 Appendix B-2 Multi-core Synchronization Instructions

This synchronization instruction set implements the extension of multi-core synchronization instructions, 32-bit width for each instruction. And the following instructions are listed in alphabetic order.

17.2.1 SYNC——The Synchronization Instruction

Syntax:

sync

Operation:

Ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction.

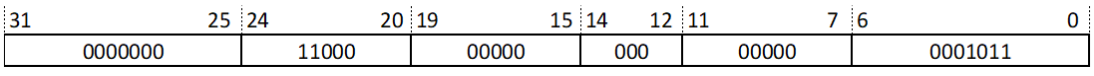
Execute Permission:

Machine Mode (M-mode)/Supervisor Mode (S-mode)/User Mode (U-mode)

Exception:

The illegal instruction exception

Instruction format:



17.2.2 SYNC.I——The Instruction to Synchronize the Clearing Operation

Syntax:

sync.i

Operation:

Ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction, and clears the pipeline when this instruction retires.

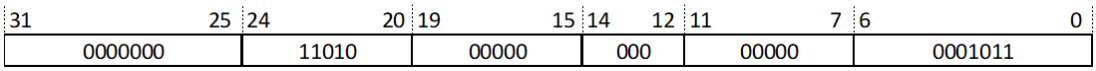
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.2.3 SYNC.IS——The Instruction to Synchronize the Clearing Operation and Broadcast

Syntax:

sync.is

Operation:

Ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction. Clears the pipeline when this instruction retires and broadcasts the request to other cores.

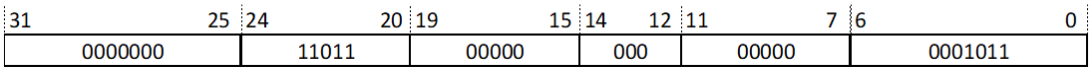
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.2.4 SYNC.S——The Instruction to Synchronize and Broadcast

Syntax:

sync.s

Operation:

Ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction, and broadcasts the request to other cores.

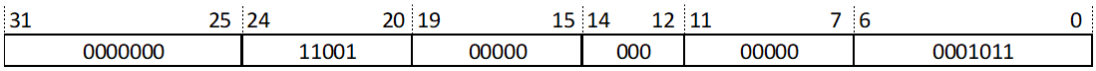
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.3 Appendix B-3 Arithmetic Operation Instructions

Arithmetic operation instruction subset implements the extension of arithmetic instructions, 32-bit width for each instruction.

And the following instructions are listed in alphabetic order.

17.3.1 ADDSL——The Shift and Add Instruction in Registers

Syntax:

addsl rd rs1, rs2, imm2

Operation:

rd ← rs1+ rs2<<imm2

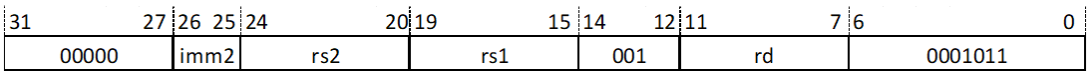
Execute Permission:

Machine Mode (M-mode)/Supervisor Mode (S-mode)/User Mode (U-mode)

Exception:

The illegal instruction exception

Instruction format:



17.3.2 MULA——The Multiply-add Instruction

Syntax:

mula rd, rs1, rs2

Operation:

rd ← rd+ (rs1 * rs2)[63:0]

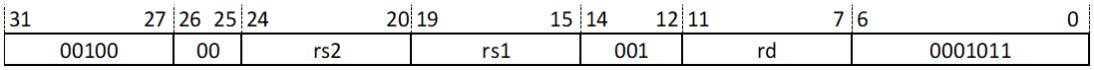
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



31 27	26 25	24 20	19 15	14 12	11 7	6 0
00100	00	rs2	rs1	001	rd	0001011

17.3.3 MULAH——The Multiply-add Instruction on the Lower 16 Bits

Syntax:

mulah rd, rs1, rs2

Operation:

tmp[31:0] ← rd[31:0]+ (rs1[15:0] * rs[15:0])

rd ←sign_extend(tmp[31:0])

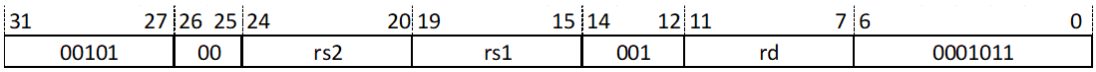
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



31 27	26 25	24 20	19 15	14 12	11 7	6 0
00101	00	rs2	rs1	001	rd	0001011

17.3.4 MULAW——The Multiply-add Instruction on the Lower 32 Bits

Syntax:

mulaw rd, rs1, rs2

Operation:

tmp[31:0] ← rd[31:0]+ (rs1[31:0] * rs[31:0])[31:0]

rd ←sign_extend(tmp[31:0])

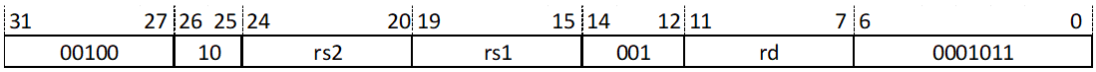
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.3.5 MULS——The Multiply-subtract Instruction

Syntax:

muls rd, rs1, rs2

Operation:

rd ← rd- (rs1 * rs2)[63:0]

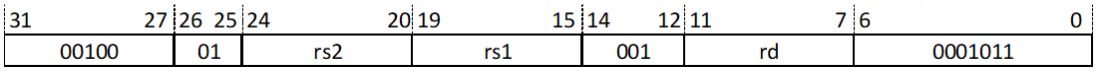
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.3.6 MULSH——The Multiply-subtract Instruction on the Lower 16 Bits

Syntax:

mulsh rd, rs1, rs2

Operation:

tmp[31:0] ← rd[31:0]- (rs1[15:0] * rs[15:0])

rd ←sign_extend(tmp[31:0])

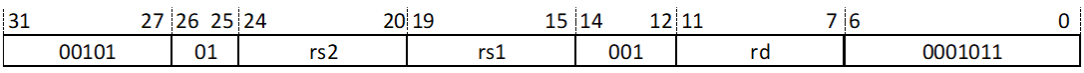
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.3.7 MULSW——The Multiply-subtract Instruction on the Lower 32 Bits

Syntax:

mulaw rd, rs1, rs2

Operation:

tmp[31:0] ← rd[31:0]- (rs1[31:0] * rs[31:0])

rd ←sign_extend(tmp[31:0])

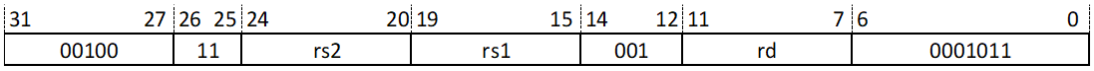
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.3.8 MVEQZ——The Transfer Instruction if Register is Zero

Syntax:

mveqz rd, rs1, rs2

Operation:

if (rs2 == 0)

rd ← rs1

else

rd ← rd

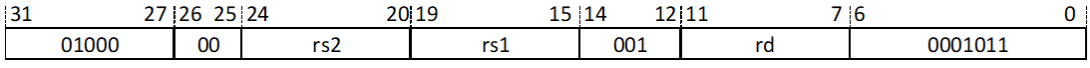
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.3.9 MVNEZ——The Transfer Instruction if Register is not Zero

Syntax:

mvnez rd, rs1, rs2

Operation:

if (rs2 != 0)

rd ← rs1

else

rd ← rd

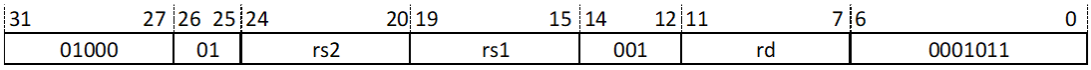
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.3.10 SRRI——The Rotate Right Instruction

Syntax:

srri rd, rs1, imm6

Operation:

rd ← rs1 >>>> imm6

Shifts right the original value of rs1, with the left bit shifted in and the right bit shifted out.

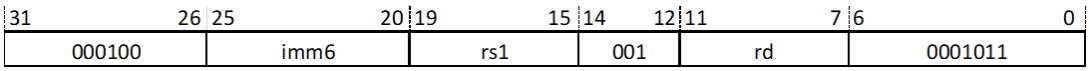
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.3.11 SRRIW——The Rotate Right Instruction on the Lower 32 Bits

Syntax:

srriw rd, rs1, imm5

Operation:

rd ← sign_extend(rs1[31:0] >>>> imm5)

Shifts right the original value of rs1[31:0], with the left bit shifted in and the right bit shifted out.

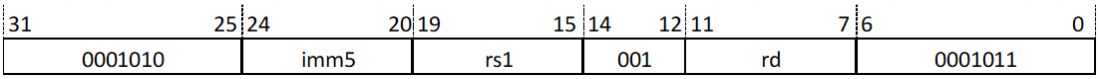
Execute Permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.4 Appendix B-4 Bitwise Operation Instruction

Bitwise operation instruction subset implements the extension of bitwise operation instructions, 32-bit width for each instruction.

And the following instructions are listed in alphabetic order.

17.4.1 EXT——The Instruction to Extract the Sign Bit and Extending in Consecutive Bits of a Register

Syntax:

ext rd, rs1, imm1,imm2

Operation:

rd←sign_extend(rs1[imm1:imm2])

Execute permission:

M-mode/S-mode/U-mode

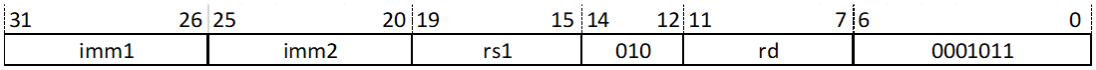
Exception:

The illegal instruction exception

Note:

If imm1<imm2, the behavior of this instruction is unpredictable.

Instruction format:



17.4.2 EXTU——The Zero Extension Instruction to Extract Consecutive Bits of a Register

Syntax:

extu rd, rs1, imm1,imm2

Operation:

rd←zero_extend(rs1[imm1:imm2])

Execute permission:

M-mode/S-mode/U-mode

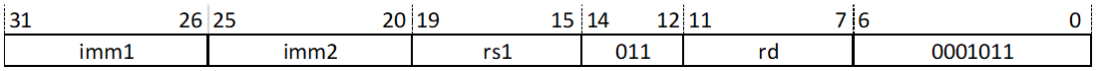
Exception:

The illegal instruction exception

Note:

If imm1<imm2, the behavior of this instruction is unpredictable.

Instruction format:



17.4.3 FF0——The Instruction to Find the First Bit With the Value of 0 in a Register

Syntax:

ff0 rd, rs1

Operation:

Finds the first bit with the value of 0 from the highest bit of rs1 and writes the result back into the rd register. If the highest bit of rs1 is 0, the result is 0. If there is no 0 in rs1, the result is 64.

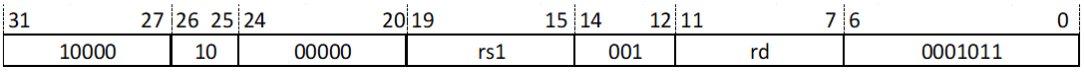
Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.4.4 FF1——The Instruction to Find the First Bit With the Value of 1 in a Register

Syntax:

ff1 rd, rs1

Operation:

Finds the first bit with the value of 1 from the highest bit of rs1 and writes the index of this bit back into rd. If the highest bit of rs1 is set to 0, the result is 0. If there is no 1 in rs1, the result is 64.

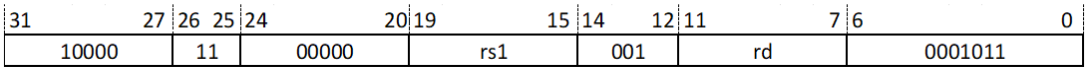
Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



31 27	26 25	24 20	19 15	14 12	11 7	6 0
10000	11	00000	rs1	001	rd	0001011

17.4.5 REV——The Instruction to Reverse the Byte Order

Syntax:

rev rd, rs1

Operation:

rd[63:56] ←rs1[7:0]

rd[55:48] ←rs1[15:8]

rd[47:40] ←rs1[23:16]

rd[39:32] ←rs1[31:24]

rd[31:24] ←rs1[39:32]

rd[23:16] ←rs1[47:40]

rd[15:8] ←rs1[55:48]

rd[7:0] ←rs1[63:56]

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
10000	01	00000	rs1	001	rd	0001011

17.4.6 REVW——The Instruction to Reverses the Byte Order on the Lower 32 Bits

Syntax:

revw rd, rs1

Operation:

tmp[31:24] ←rs1[7:0]

tmp [23:16] ←rs1[15:8]

tmp [15:8] ←rs1[23:16]

tmp [7:0] ←rs1[31:24]

rd ←sign_extend(tmp[31:0])

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
10010	00	00000	rs1	001	rd	0001011

17.4.7 TST——The Instruction to Test Bits with the Value of 0

Syntax:

tst rd, rs1, imm6

Operation:

if(rs1[imm6] == 1)

    rd←1

else

    rd←0

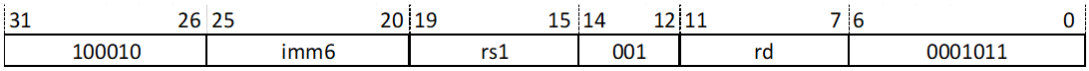
Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.4.8 TSTNBZ——The Instruction to Test Byte with the Value of 0

Syntax:

tstnbz rd, rs1

Operation:

rd[63:56] ← (rs1[63:56] == 0) ? 8'hff : 8'h0

rd[55:48] ← (rs1[55:48] == 0) ? 8'hff : 8'h0

rd[47:40] ← (rs1[47:40] == 0) ? 8'hff : 8'h0

rd[39:32] ← (rs1[39:32] == 0) ? 8'hff : 8'h0

rd[31:24] ← (rs1[31:24] == 0) ? 8'hff : 8'h0

rd[23:16] ← (rs1[23:16] == 0) ? 8'hff : 8'h0

rd[15:8] ← (rs1[15:8] == 0) ? 8'hff : 8'h0

rd[7:0] ← (rs1[7:0] == 0) ? 8'hff : 8'h0

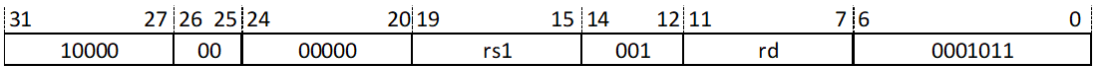
Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Instruction format:



17.5 Appendix B-5 Store Instructions

The store instruction subset implements the extension of store instructions, 32-bit width for each instruction.

And the following instructions are listed in alphabetic order.

17.5.1 FLRD——The Instruction to Shift and Load Doubleword in Floating-Point Registers

Syntax:

flrd rd, rs1, rs2, imm2

Operation:

rd ←mem[(rs1+rs2<<imm2)+7: (rs1+rs2<<imm2)]

Execute permission:

Machine Mode (M-mode)/Supervisor Mode (S-mode)/User Mode (U-mode)

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

If mxstatus.theadisaee=1' b0 or mstatus.fs =2' b00, executing this instruction causes an exception of illegal instruction.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
01100	imm2	rs2	rs1	110	rd	0001011

17.5.2 FLRW——The Instruction to Shift and Load Word in Floating-Point Registers

Syntax:

flrw rd, rs1, rs2, imm2

Operation:

rd ←one_extend(mem[(rs1+rs2<<imm2)+3: (rs1+rs2<<imm2)])

Execute permission:

M-mode/S-mode/U-mode

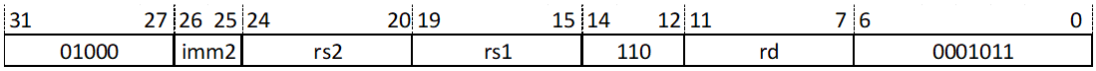
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

If mxstatus.theadisaee=1' b0 or mstatus.fs =2' b00, executing this instruction causes an exception of illegal instruction.

Instruction format:



17.5.3 FLURD——The Doubleword Load Instruction to Shift the Low 32 Bits of Floating-point Registers

Syntax:

flurd rd, rs1, rs2, imm2

Operation:

rd ←mem[(rs1+rs2[31:0]<<imm2)+7: (rs1+rs2[31:0]<<imm2)]

Execute permission:

M-mode/S-mode/U-mode

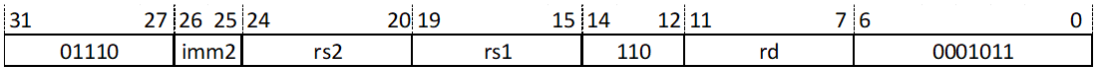
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

- rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.
- If mxstatus.theadisaee=1’ b0 or mstatus.fs = 2’ b00, executing this instruction causes an exception of illegal instruction.

Instruction format:



17.5.4 FLURW——The Load Word Instruction to Shift the Low 32 Bits of Floating-point Registers

Syntax:

flurw rd, rs1, rs2, imm2

Operation:

rd ←one_extend(mem[(rs1+rs2[31:0]<<imm2)+3: (rs1+rs2[31:0]<<imm2)])

Execute permission:

M-mode/S-mode/U-mode

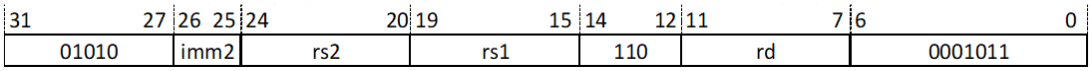
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

- rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.
- If mxstatus.theadisaee=1' b0 or mstatus.fs = 2' b00, executing this instruction causes an exception of illegal instruction.

Instruction format:



31 27	26 25	24 20	19 15	14 12	11 7	6 0
01010	imm2	rs2	rs1	110	rd	0001011

17.5.5 FSRD——The Instruction to Shift and Doubleword Store in Floating-Point Registers

Syntax:

fsrd rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2<<imm2)+7: (rs1+rs2<<imm2)] ←rd[63:0]

Execute permission:

M-mode/S-mode/U-mode

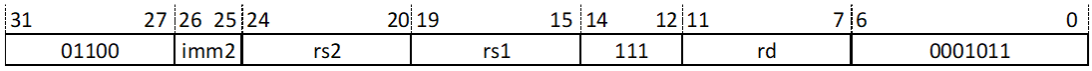
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Note:

If mxstatus.theadisaee=1' b0 or mstatus.fs =2' b00, executing this instruction causes an illegal instruction exception.

Instruction format:



31 27	26 25	24 20	19 15	14 12	11 7	6 0
01100	imm2	rs2	rs1	111	rd	0001011

17.5.6 FSRW——The Instruction to Shift and Store Word in Floating-Point Registers

Syntax:

fsrw rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2<<imm2)+3: (rs1+rs2<<imm2)] ←rd[31:0]

Execute permission:

M-mode/S-mode/U-mode

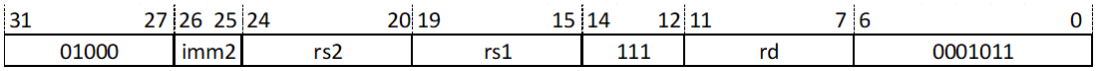
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Note:

If mxstatus.theadisaee=1' b0 or mstatus.fs =2' b00, executing this instruction causes an illegal instruction exception.

Instruction format:



17.5.7 FSURD——The Doubleword Store Instruction to Shift Low 32 Bits in Floating-point Registers

Syntax:

fsurd rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2[31:0]<<imm2)+7: (rs1+rs2[31:0]<<imm2)] ←rd[63:0]

Execute permission:

M-mode/S-mode/U-mode

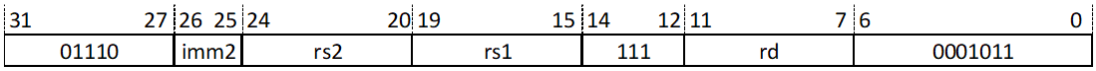
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Note:

- rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.
- If mxstatus.theadisaee=1' b0 or mstatus.fs = 2' b00, executing this instruction causes an exception of illegal instruction.

Instruction format:



17.5.8 FSURW——The Word Store Instruction to Shift Low 32 Bits in Floating-point Registers

Syntax:

fsurw rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2[31:0]<<imm2)+3: (rs1+rs2[31:0]<<imm2)] ←rd[31:0]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Note:

- rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.
- If mxstatus.theadisaee=1’ b0 or mstatus.fs = 2’ b00, executing this instruction causes an exception of illegal instruction.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
01010	imm2	rs2	rs1	111	rd	0001011

17.5.9 LBIA——The Base-address Auto-increment Instruction to Extend Signed Bits and Load Bytes

Syntax:

lbia rd, (rs1), imm5,imm2

Operation:

rd ←sign_extend(mem[rs1])

rs1←rs1 + sign_extend(imm5 << imm2)

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
00011	imm2	imm5	rs1	100	rd	0001011

17.5.10 LBIB——The Byte Load Instruction to Auto-increment the Base Address and Extend Signed Bits

Syntax:

lbib rd, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

rd ←sign_extend(mem[rs1])

Execute permission:

M-mode/S-mode/U-mode

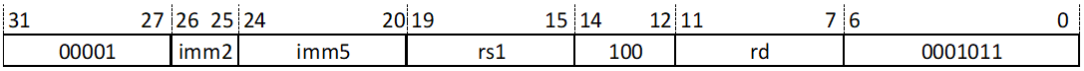
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



17.5.11 LBUIA——The Base-address Auto-increment Instruction to Extend Zero Bits and Load Bytes

Syntax:

lbuia rd, (rs1), imm5,imm2

Operation:

rd ←zero_extend(mem[rs1])

rs1←rs1 + sign_extend(imm5 << imm2)

Execute permission:

M-mode/S-mode/U-mode

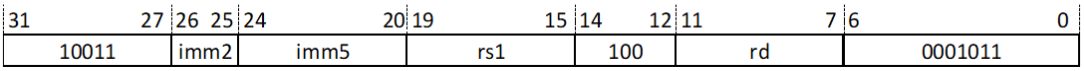
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



17.5.12 LBUIB——The Byte Load Instruction to Auto-increment the Base Address and Extend Zero Bits

Syntax:

lbuib rd, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

rd ←zero_extend(mem[rs1])

Execute permission:

M-mode/S-mode/U-mode

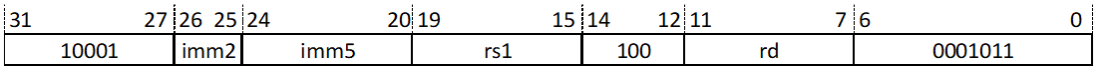
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



17.5.13 LDD——Dual-Register Load Instruction

Syntax:

ldd rd1,rd2, (rs1),imm2, 4

Operation:

address←rs1 + zero_extend(imm2<<4)

rd1←mem[address+7:address]

rd2←mem[address+15:address+8]

Execute permission:

M-mode/S-mode/U-mode

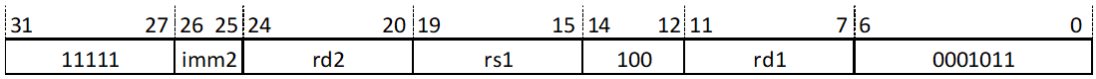
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd1, rd2 ,rs1 must not equal to each other.

Instruction format:



17.5.14 LDIA——The Base-address Auto-increment Instruction to Load Doublewords and Extend Signed Bits

Syntax:

ldia rd, (rs1), imm5,imm2

Operation:

rd ←sign_extend(mem[rs1+7:rs1])

rs1←rs1 + sign_extend(imm5 << imm2)

Execute permission:

M-mode/S-mode/U-mode

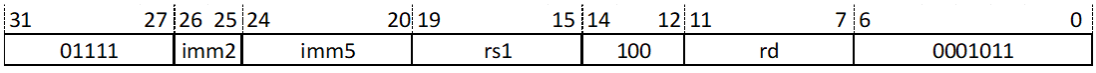
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



17.5.15 LDIB——The Doubleword Load Instruction to Auto-increment the Base Address and Extend the Signed Bits

Syntax:

ldib rd, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

rd ←sign_extend(mem[rs1+7:rs1])

Execute permission:

M-mode/S-mode/U-mode

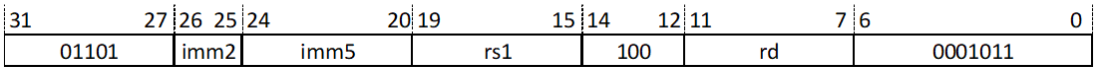
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



17.5.16 LHIA——The Base-address Auto-increment Instruction to Load Halfwords and Extend Signed Bits

Syntax:

lhia rd, (rs1), imm5,imm2

Operation:

rd ←sign_extend(mem[rs1+1:rs1])

rs1←rs1 + sign_extend(imm5 << imm2)

Execute permission:

M-mode/S-mode/U-mode

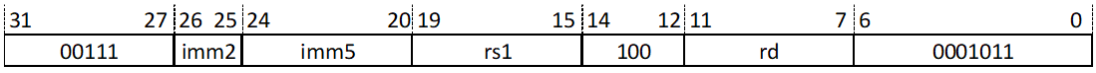
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



17.5.17 LHIB——The Halfword Load Instruction to Auto-increment the Base Address and Extend Signed Bits

Syntax:

lhib rd, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

rd ←sign_extend(mem[rs1+1:rs1])

Execute permission:

M-mode/S-mode/U-mode

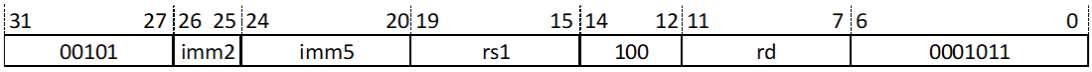
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



31 27	26 25	24 20	19 15	14 12	11 7	6 0
00101	imm2	imm5	rs1	100	rd	0001011

17.5.18 LHUIA——The Halfword Load Instruction to Auto-increment the Base Address and Extend Zero Bits

Syntax:

lhuia rd, (rs1), imm5,imm2

Operation:

rd ←zero_extend(mem[rs1+1:rs1])

rs1←rs1 + sign_extend(imm5 << imm2)

Execute permission:

M-mode/S-mode/U-mode

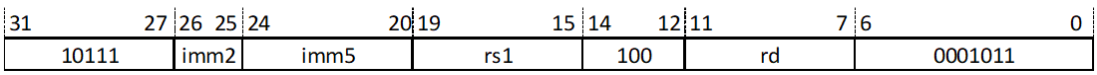
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



31 27	26 25	24 20	19 15	14 12	11 7	6 0
10111	imm2	imm5	rs1	100	rd	0001011

17.5.19 LHUIB——The Halfword Load Instruction to Auto-increment the Base Address and Extend Zero Bits

Syntax:

lhuib rd, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

rd ←zero_extend(mem[rs1+1:rs1])

Execute permission:

M-mode/S-mode/U-mode

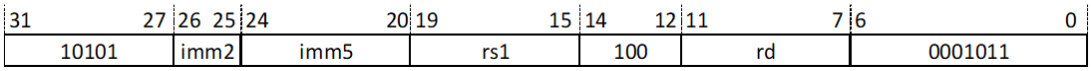
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



17.5.20 LRB——The Byte Load Instruction to Shift Registers and Extend Signed Bits

Syntax:

lrb rd, rs1, rs2, imm2

Operation:

rd ←sign_extend(mem[(rs1+rs2<<imm2)])

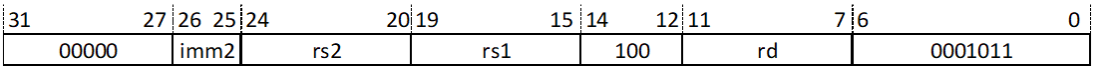
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Instruction format:



17.5.21 LRBU——The Byte Load Instruction to Shift Registers and Extend Zero Bits

Syntax:

lrbu rd, rs1, rs2, imm2

Operation:

rd ←zero_extend(mem[(rs1+rs2<<imm2)])

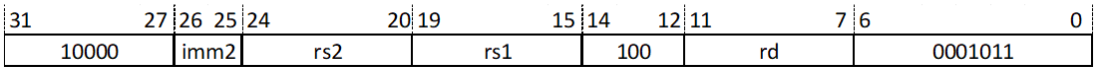
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Instruction format:



17.5.22 LRD——The Doubleword Load Instruction with Register Shift

Syntax:

lrd rd, rs1, rs2, imm2

Operation:

rd ←mem[(rs1+rs2<<imm2)+7: (rs1+rs2<<imm2)]

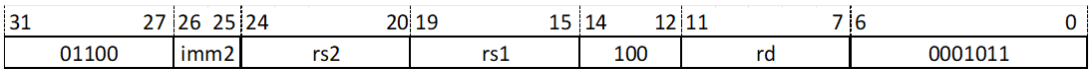
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Instruction format:



17.5.23 LRH——The Halfword Load Instruction to Shift Registers and Extend Signed Bits

Syntax:

lrh rd, rs1, rs2, imm2

Operation:

rd ←sign_extend(mem[(rs1+rs2<<imm2)+1: (rs1+rs2<<imm2)])

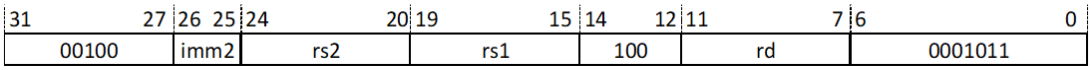
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Instruction format:



17.5.24 LRHU——The Halfword Load Instruction to Shift Registers and Extend Zero Bits

Syntax:

lrhu rd, rs1, rs2, imm2

Operation:

rd ←zero_extend(mem[(rs1+rs2<<imm2)+1: (rs1+rs2<<imm2)])

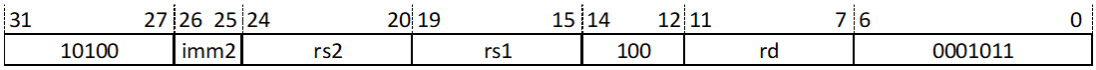
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Instruction format:



17.5.25 LRW——The Word Load Instruction to Shift Registers and Extend Signed Bits

Syntax:

lrw rd, rs1, rs2, imm2

Operation:

rd ←sign_extend(mem[(rs1+rs2<<imm2)+3: (rs1+rs2<<imm2)])

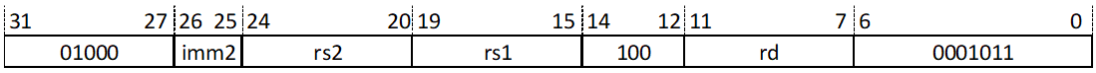
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Instruction format:



17.5.26 LRWU——The Word Load Instruction to Shift Registers and Extend Zero Bits

Syntax:

lrwu rd, rs1, rs2, imm2

Operation:

rd ←zero_extend(mem[(rs1+rs2<<imm2)+3: (rs1+rs2<<imm2)])

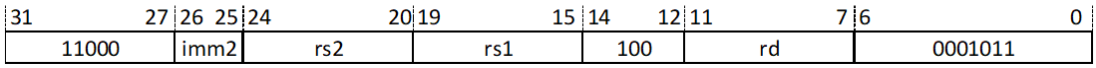
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Instruction format:



17.5.27 LURB——The Byte Load Instruction to Shift the Low 32 Bits of Registers and Extend Signed Bits

Syntax:

lurb rd, rs1, rs2, imm2

Operation:

rd ←sign_extend(mem[(rs1+rs2[31:0]<<imm2)])

Execute permission:

M-mode/S-mode/U-mode

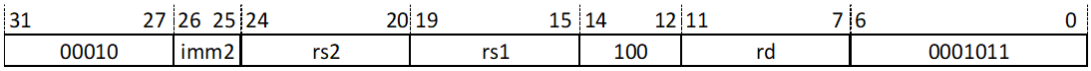
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.

Instruction format:



17.5.28 LURBU——The Byte Load Instruction to Shift the Low 32 Bits of Registers and Extend Zero Bits

Syntax:

lurbu rd, rs1, rs2, imm2

Operation:

rd ←zero_extend(mem[(rs1+rs2[31:0]<<imm2)])

Execute permission:

M-mode/S-mode/U-mode

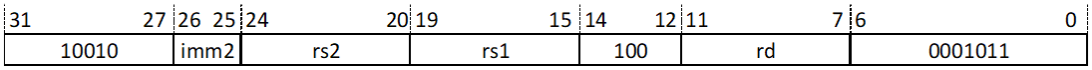
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.

Instruction format:



17.5.29 LURD——The Doubleword Load Instruction to Shift the Low 32 Bits of Registers

Syntax:

lurd rd, rs1, rs2, imm2

Operation:

rd ←mem[(rs1+rs2[31:0]<<imm2)+7: (rs1+rs2[31:0]<<imm2)]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
01110	imm2	rs2	rs1	100	rd	0001011

17.5.30 LURH——The Halfword Load Instruction to Shift the Low 32 Bits of Registers and Extend Signed Bits

Syntax:

lurh rd, rs1, rs2, imm2

Operation:

rd ←sign_extend(mem[(rs1+rs2[31:0]<<imm2)+1:

(rs1+rs2[31:0]<<imm2)])

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
00110	imm2	rs2	rs1	100	rd	0001011

17.5.31 LURHU——The Halfword Load Instruction to Shift the Low 32 Bits of Registers and Extend Zero Bits

Syntax:

lurhu rd, rs1, rs2, imm2

Operation:

rd ←zero_extend(mem[(rs1+rs2[31:0]<<imm2)+1:

(rs1+rs2[31:0]<<imm2)])

Execute permission:

M-mode/S-mode/U-mode

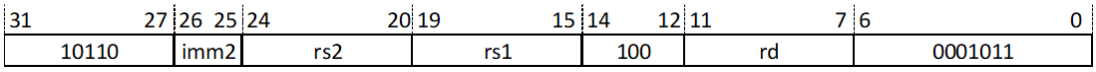
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.

Instruction format:



17.5.32 LURW——The Word Load Instruction to Shift the Low 32 Bits of Registers and Extend Signed Bits

Syntax:

lurw rd, rs1, rs2, imm2

Operation:

rd ←sign_extend(mem[(rs1+rs2[31:0]<<imm2)+3:

(rs1+rs2[31:0]<<imm2)])

Execute permission:

M-mode/S-mode/U-mode

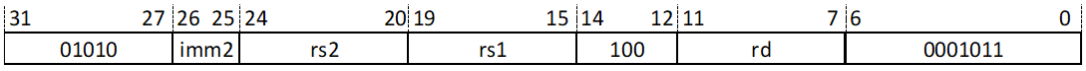
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.

Instruction format:



17.5.33 LURWU——The Word Load Instruction to Shift 32 Bits of Registers and Extend Zero Bits

Syntax:

lurwu rd, rs1, rs2,imm2

Operation:

rd ←zero_extend(mem[(rs1+rs2[31:0]<<imm2)+3:(rs1+rs2[31:0]<<imm2)])

Execute permission:

M-mode/S-mode/U-mode

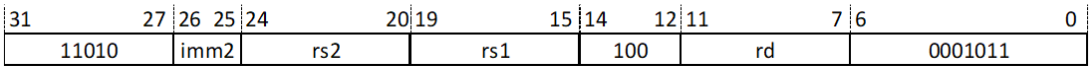
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number

Instruction format:



17.5.34 LWD——The Word Load Instruction in Double Registers with Sign Extension

Syntax:

lwd rd1, rd2, (rs1), imm2

Operation:

address←rs1+zero_extend(imm2<<3)

rd1 ←sign_extend(mem[address+3: address])

rd2 ←sign_extend(mem[address+7: address+4])

Execute permission:

M-mode/S-mode/U-mode

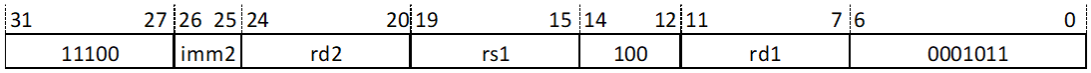
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd1, rd2 ,rs1 must not equal to each other.

Instruction format:



17.5.35 LWIA——The Base-address Auto-increment Instruction to Extend Signed Bits and Load Words

Syntax:

lwia rd, (rs1), imm5,imm2

Operation:

rd ←sign_extend(mem[rs1+3:rs1])

rs1←rs1 + sign_extend(imm5 << imm2)

Execute permission:

M-mode/S-mode/U-mode

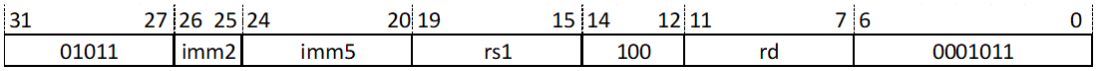
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



17.5.36 LWIB——The Word Load Instruction to Auto-increment the Base Address and Extend Signed Bits

Syntax:

lwib rd, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

rd ←sign_extend(mem[rs1+3:rs1])

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
01001	imm2	imm5	rs1	100	rd	0001011

17.5.37 LWUD——The Word Load Instruction in Double Registers With Zero Extension

Syntax:

lwud rd1,rd2, (rs1),imm2

Operation:

address←rs1+zero_extend(imm2<<3)

rd1 ←zero_extend(mem[address+3: address])

rd2 ←zero_extend(mem[address+7: address+4])

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd1, rd2 ,rs1 must not equal to each other.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
11110	imm2	rd2	rs1	100	rd1	0001011

17.5.38 LWUIA——The Base-address Auto-increment Instruction to Extend Zero Bits and Load words

Syntax:

lwuia rd, (rs1), imm5,imm2

Operation:

rd ←zero_extend(mem[rs1+3:rs1])

rs1←rs1 + sign_extend(imm5 << imm2)

Execute permission:

M-mode/S-mode/U-mode

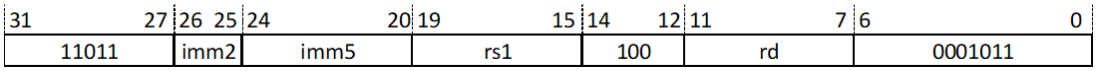
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



17.5.39 LWUIB——The Word Load Instruction to Auto-increment the Base address and Extend zero bits

Syntax:

lwuib rd, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

rd ←zero_extend(mem[rs1+3:rs1])

Execute permission:

M-mode/S-mode/U-mode

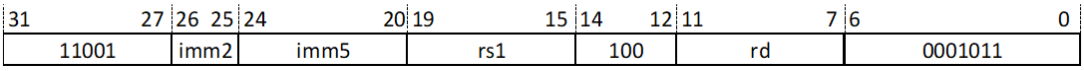
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Note:

The values of rd and rs1 must not be the same.

Instruction format:



17.5.40 SBIA——The Byte Store Instruction with Auto-increment Base-address

Syntax:

sbia rs2, (rs1), imm5,imm2

Operation:

mem[rs1]←rs2[7:0]

rs1←rs1 + sign_extend(imm5 << imm2)

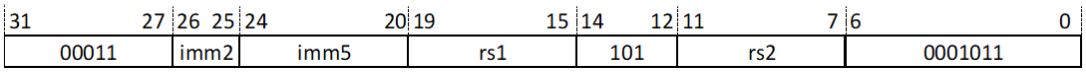
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.41 SBIB——The Byte Store Instruction to Auto-increment the Base Address

Syntax:

sbib rs2, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

mem[rs1] ←rs2[7:0]

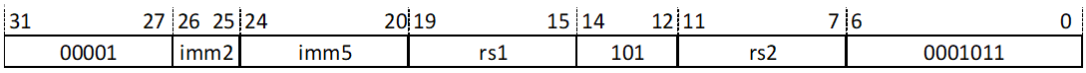
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.42 SDD——Dual Register Store Instruction

Syntax:

sdd rd1,rd2, (rs1),imm2, 4

Operation:

address←rs1 + zero_extend(imm2<<4)

mem[address+7:address] ←rd1

mem[address+15:address+8]←rd2

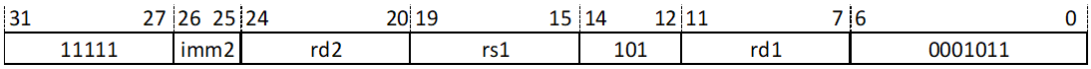
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.43 SDIA——The Base-address Auto-increment Instruction to Store Doublewords

Syntax:

sdia rs2, (rs1), imm5,imm2

Operation:

mem[rs1+7:rs1]←rs2[63:0]

rs1←rs1 + sign_extend(imm5 << imm2)

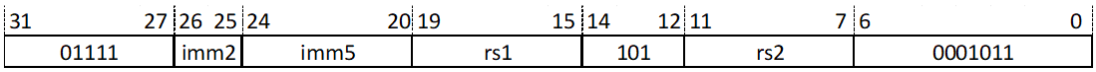
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.44 SDIB——The Doubleword Store Instruction to Auto-increment the Base Address

Syntax:

sdib rs2, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

mem[rs1+7:rs1] ←rs2[63:0]

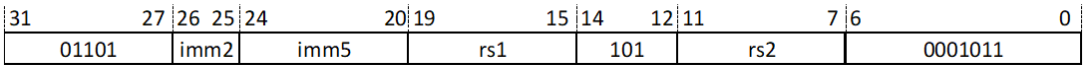
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.45 SHIA——The Base-address Auto-increment Instruction to Store Halfwords

Syntax:

shia rs2, (rs1), imm5,imm2

Operation:

mem[rs1+1:rs1]←rs2[15:0]

rs1←rs1 + sign_extend(imm5 << imm2)

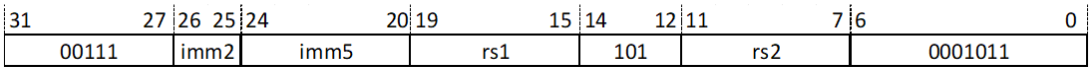
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.46 SHIB——The Halfword Store Instruction to Auto-increment the Base Address

Syntax:

shib rs2, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

mem[rs1+1:rs1] ←rs2[15:0]

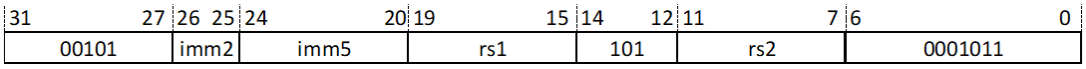
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.47 SRB——The Instruction to Shift and Store Bytes in Registers

Syntax:

srb rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2<<imm2)] ←rd[7:0]

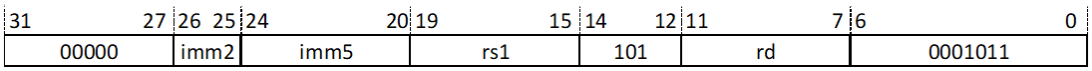
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.48 SRD——The Instruction to Shift and Store Doubleword from Registers

Syntax:

srd rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2<<imm2)+7: (rs1+rs2<<imm2)] ←rd[63:0]

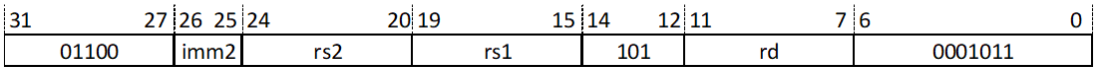
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.49 SRH——The Instruction to Shift and Store Halfword in Registers

Syntax:

srh rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2<<imm2)+1: (rs1+rs2<<imm2)] ←rd[15:0]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
00100	imm2	rs2	rs1	101	rd	0001011

17.5.50 SRW——The Instruction to Shift and Store Word in Registers

Syntax:

srw rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2<<imm2)+3: (rs1+rs2<<imm2)] ←rd[31:0]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
01000	imm2	rs2	rs1	101	rd	0001011

17.5.51 SURB——The Byte Store Instruction to Shift the Low 32 Bits of Registers

Syntax:

surb rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2[31:0]<<imm2)] ←rd[7:0]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
00010	imm2	rs2	rs1	101	rd	0001011

17.5.52 SURD——The Doubleword Store Instruction to Shift the Low 32 Bits of Registers

Syntax:

surd rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2[31:0]<<imm2)+7: (rs1+rs2[31:0]<<imm2)] ←rd[63:0]

Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.

Instruction format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
01110	imm2	rs2	rs1	101	rd	0001011

17.5.53 SURH——The Halfword Store Instruction to Shift the Low 32 Bits of Registers

Syntax:

surh rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2[31:0]<<imm2)+1: (rs1+rs2[31:0]<<imm2)] ←rd[15:0]

Execute permission:

M-mode/S-mode/U-mode

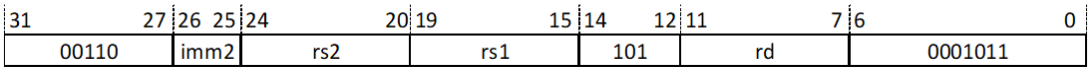
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.

Instruction format:



17.5.54 SURW——The Word Store Instruction to Shift the Low 32 Bits of Registers

Syntax:

surw rd, rs1, rs2, imm2

Operation:

mem[(rs1+rs2[31:0]<<imm2)+3: (rs1+rs2[31:0]<<imm2)] ←rd[31:0]

Execute permission:

M-mode/S-mode/U-mode

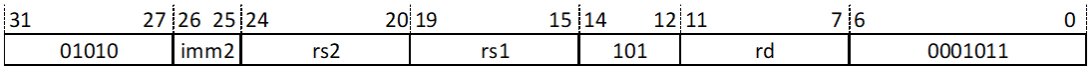
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Note:

rs2[31:0] is an unsigned number, and the upper bits [63:32] are filled with zeros for address calculation.

Instruction format:



17.5.55 SWIA——The Base-address Auto-increment Instruction to Stores Words

Syntax:

swia rs2, (rs1), imm5,imm2

Operation:

mem[rs1+3:rs1]←rs2[31:0]

rs1←rs1 + sign_extend(imm5 << imm2)

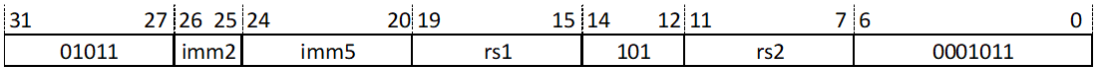
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.56 SWIB——The Word Store Instruction to Auto-increment the Base Address

Syntax:

swib rs2, (rs1), imm5,imm2

Operation:

rs1←rs1 + sign_extend(imm5 << imm2)

mem[rs1+3:rs1] ←rs2[31:0]

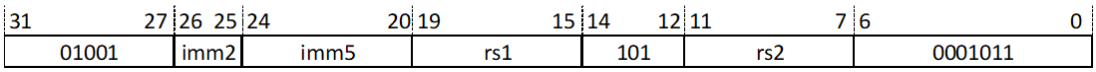
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.5.57 SWD——The Instruction to Store the Low 32 Bits of Double Registers

Syntax:

swd rd1,rd2,(rs1),imm2

Operation:

address←rs1+ zero_extend(imm2<<3)

mem[address+3:address] ←rd1[31:0]

mem[address+7:address+4]←rd2[31:0]

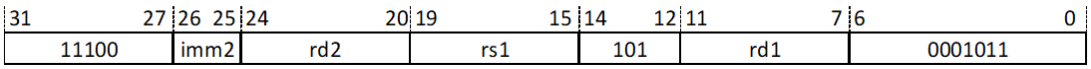
Execute permission:

M-mode/S-mode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.6 Appendix B-6 Half-precision Floating-point Instructions

The half-precision floating-point subset implements the half-precision floating-point, 32-bit width for each instruction, and the following instructions are listed in alphabetic order.

17.6.1 FADD.H——The Half-precision Floating-point Add Instruction

Syntax:

fadd.h fd, fs1, fs2, rm

Operation:

fd ← fs1 + fs2

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit: Invalid Operation (NV)/Overflow (OF)/Inexact (NX)

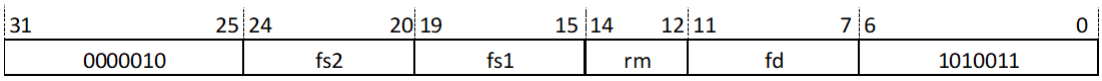
Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fadd.h fd, fs1,fs2,rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fadd.h fd, fs1,fs2,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fadd.h fd, fs1,fs2,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fadd.h fd, fs1,fs2,rup.
- 3' b100: Rounds to the nearest large value. And the corresponding assembly instruction is fadd.h fd, fs1,fs2,rmm.
- 3' b101: This code is reserved and not used.

- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fadd.h fd, fs1,fs2.

Instruction format:



17.6.2 FCLASS.H——The Half-precision Floating-point Classification Instruction

Syntax:

fclass.h rd, fs1

Operation:

if (fs1 = -inf)

rd ← 64’ h1

if (fs1 = -norm)

rd ← 64’ h2

if (fs1 = -subnorm)

rd ← 64’ h4

if (fs1 = -zero)

rd ← 64’ h8

if (fs1 = +zero)

rd ← 64’ h10

if (fs1 = +subnorm)

rd ← 64’ h20

if (fs1 = +norm)

rd ← 64’ h40

if (fs1 = +inf)

rd ← 64’ h80

if (fs1 = sNaN)

rd ← 64’ h100

if (fs1 = qNaN)

rd ← 64’ h200

Execute permission:

M-mode/Smode/U-mode

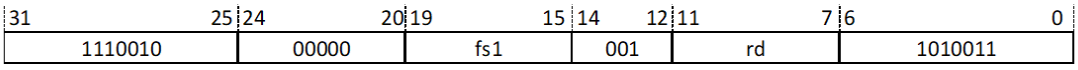
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



17.6.3 FCVT.D.H——The Instruction to Convert a Half-precision Floating-Point Number into a Double-precision Floating-point Number

Syntax:

fcvt.d.h fd, fs1

Operation:

fd ← half_convert_to_double(fs1)

Execute permission:

M-mode/S-mode/U-mode

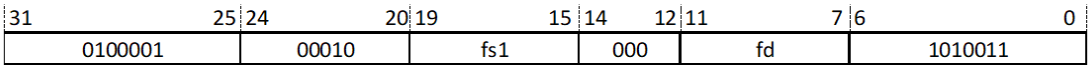
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



17.6.4 FCVT.H.D——The Instruction to Convert a Double-precision Floating-Point Number into a Half-precision Floating-point Number

Syntax:

fcvt.h.d fd, fs1, rm

Operation:

fd ← double_convert_to_half(fs1)

Execute permission:

M-mode/S-mode/U-mode

Exception:

The illegal instruction exception

Affected flag:

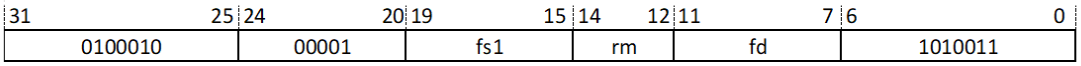
Floating-point status bit NV/OF/UF/NX

Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fcvt.h.d fd,fs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fcvt.h.d fd,fs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fcvt.h.d fd,fs1,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fcvt.h.d fd,fs1,rup.
- 3' b100: Rounds to the nearest large value. And the corresponding assembly instruction is fcvt.h.d fd,fs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fcvt.h.d fd, fs1.

Instruction format:



17.6.5 FCVT.H.L——The Instruction to Convert a Signed Long Integer into a Half-precision Floating-point Number

Syntax:

fcvt.h.l fd, rs1, rm

Operation:

fd ← signed_long_convert_to_half(rs1)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NX/OF

Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fcvt.h.l fd,rs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fcvt.h.l fd,rs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fcvt.h.l fd,rs1,fdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fcvt.h.l fd,rs1,rup.
- 3' b100: Rounds to the nearest large value. And the corresponding assembly instruction is fcvt.h.l fd,rs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fcvt.h.l fd, rs1.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1101010	00010	rs1	rm	fd	1010011

17.6.6 FCVT.H.LU——The Instruction to Convert an Unsigned Long Integer into a Half-precision Floating-point Number

Syntax:

fcvt.h.lu fd, rs1, rm

Operation:

fd ← unsigned_long_convert_to_half_fp(rs1)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NX/OF

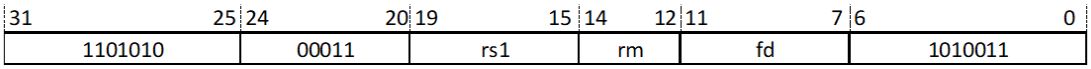
Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fcvt.h.lu fd,rs1,rne.

- 3’ b001: Rounds to zero. And the corresponding assembly instruction is fcvt.h.lu fd, rs1,rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembly instruction is fcvt.h.lu fd, rs1,fdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembly instruction is fcvt.h.lu fd, rs1,rup.
- 3’ b100: Rounds to the nearest large value. And the corresponding assembly instruction is fcvt.h.lu fd, rs1,rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fcvt.h.lu fd, rs1.

Instruction format:



17.6.7 FCVT.H.S——The Instruction to Convert a Single Precision Floating-point Number to a Half-precision Floating-point Number

Syntax:

fcvt.h.s fd, fs1, rm

Operation:

fd ← single_convert_to_half(fs1)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV/OF/UF/NX

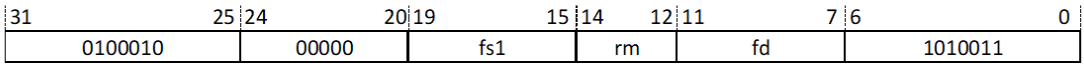
Note:

rm determines the rounding mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembly instruction is fcvt.h.s fd,fs1,rne.
- 3’ b001: Rounds to zero. And the corresponding assembly instruction is fcvt.h.s fd,fs1,rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembly instruction is fcvt.h.s fd,fs1,fdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembly instruction is fcvt.h.s fd,fs1,rup.
- 3’ b100: Rounds to the nearest large value. And the corresponding assembly instruction is fcvt.h.s fd,fs1,rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.

- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fcvt.h.s fd, fs1.

Instruction format:



17.6.8 FCVT.H.W——The Instruction to Convert a Signed Integer into a Half-precision Floating-point Number

Syntax:

fcvt.h.w fd, rs1, rm

Operation:

fd ← signed_int_convert_to_half(rs1)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

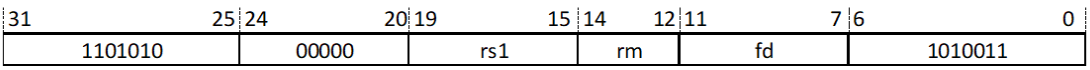
Floating-point status bit NX/OF

Note:

rm determines the rounding mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembly instruction is fcvt.h.w fd,rs1,rne.
- 3’ b001: Rounds to zero. And the corresponding assembly instruction is fcvt.h.w fd,rs1,rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembly instruction is fcvt.h.w fd,rs1,fdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembly instruction is fcvt.h.w fd,rs1,rup.
- 3’ b100: Rounds to the nearest large value. And the corresponding assembly instruction is fcvt.h.w fd,rs1,rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fcvt.h.w fd, rs1.

Instruction format:



17.6.9 FCVT.H.WU——The Instruction to Convert an Unsigned Integer into a Half-precision Floating-point Number

Syntax:

fcvt.h.wu fd, rs1, rm

Operation:

fd ← unsigned_int_convert_to_half_fp(rs1)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

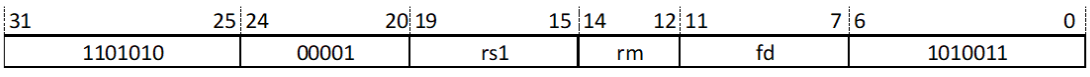
Floating-point status bit NX/OF

Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fcvt.h.wu fd,rs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fcvt.h.wu fd,rs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fcvt.h.wu fd,rs1,fdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fcvt.h.wu fd,rs1,rup.
- 3'b100: Rounds to the nearest large value. And the corresponding assembly instruction is fcvt.h.wu fd,rs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fcvt.h.wu fd, rs1.

Instruction format:



17.6.10 FCVT.L.H——The Instruction to Convert a Half-precision Floating-point Data to a Signed Long Integer

Syntax:

fcvt.l.h rd, fs1, rm

Operation:

rd ← half_convert_to_signed_long(fs1)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV/NX

Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fcvt.l.h rd,fs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fcvt.l.h rd,fs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fcvt.l.h rd,fs1,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fcvt.l.h rd,fs1,rup.
- 3' b100: Rounds to the nearest large value. And the corresponding assembly instruction is fcvt.l.h rd,fs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fcvt.l.h rd, fs1.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
1100010	00010	fs1	rm	rd	1010011

17.6.11 FCVT.LU.H——The Instruction to Convert a Half-precision Floating-point Number to an Unsigned Long Integer

Syntax:

fcvt.lu.h rd, fs1, rm

Operation:

rd ← half_convert_to_unsigned_long(fs1)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

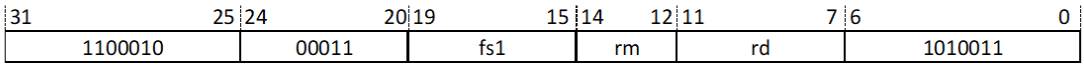
Floating-point status bit NV/NX

Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fcvt.lu.h rd,fs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fcvt.lu.h rd,fs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fcvt.lu.h rd,fs1,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fcvt.lu.h rd,fs1,rup.
- 3' b100: Rounds to the nearest large value. And the corresponding assembly instruction is fcvt.lu.h rd,fs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fcvt.lu.h rd, fs1.

Instruction format:



17.6.12 FCVT.S.H——The Instruction to Convert a Half-precision Floating-point Number to a Single Precision Floating-point Number

Syntax:

fcvt.s.h fd, fs1

Operation:

fd ← half_convert_to_single(fs1)

Execute permission:

M-mode/Smode/U-mode

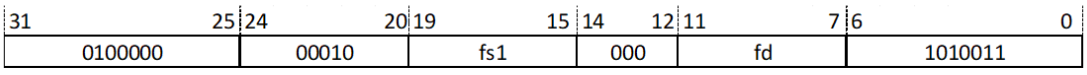
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



17.6.13 FCVT.W.H——The Instruction to Convert a Half-precision Floating-point Number to a Signed Integer

Syntax:

fcvt.w.h rd, fs1, rm

Operation:

tmp ← half_convert_to_signed_int(fs1)

rd←sign_extend(tmp)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

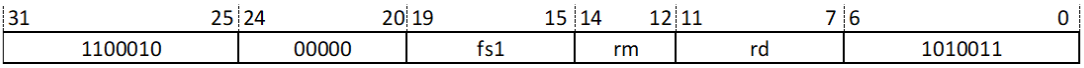
Floating-point status bit NV/NX

Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fcvt.w.h rd,fs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fcvt.w.h rd,fs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fcvt.w.h rd,fs1,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fcvt.w.h rd,fs1,rup.
- 3' b100: Rounds to the nearest large value. And the corresponding assembly instruction is fcvt.w.h rd,fs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fcvt.w.h rd, fs1.

Instruction format:



17.6.14 FCVT.WU.H——The Instruction to Convert a Half-precision Floating-point Number to an Unsigned Integer

Syntax:

fcvt.wu.h rd, fs1, rm

Operation:

tmp ← half_convert_to_unsigned_int(fs1)

rd←sign_extend(tmp)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

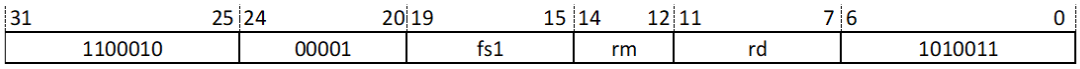
Floating-point status bit NV/NX

Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fcvt.wu.h rd,fs1,rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fcvt.wu.h rd,fs1,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fcvt.wu.h rd,fs1,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fcvt.wu.h rd,fs1,rup.
- 3'b100: Rounds to the nearest large value. And the corresponding assembly instruction is fcvt.wu.h rd,fs1,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fcvt.wu.h rd, fs1.

Instruction format:



17.6.15 FDIV.H——The Half-precision Floating-point Divide Instruction

Syntax:

fdiv.h fd, fs1, fs2, rm

Operation:

fd ← fs1 / fs2

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV/DZ/OF/UF/NX

Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fdiv.h fs1,fs2,rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fdiv.h fd fs1,fs2,rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fdiv.h fd, fs1,fs2,rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fdiv.h fd, fs1,fs2,rup.
- 3'b100: Rounds to the nearest large value. And the corresponding assembly instruction is fdiv.h fd, fs1,fs2,rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fdiv.h fd, fs1,fs2.

Instruction format:

31 25	24 20	19 15	14 12	11 7	6 0
0001110	fs2	fs1	rm	fd	1010011

17.6.16 FEQ.H——The Compare-if-equal-to Instruction of Half-precision Floating-Point Numbers

Syntax:

feq.h rd, fs1, fs2

Operation:

if(fs1 == fs2)

rd ← 1

else

rd ← 0

Execute permission:

M-mode/Smode/U-mode

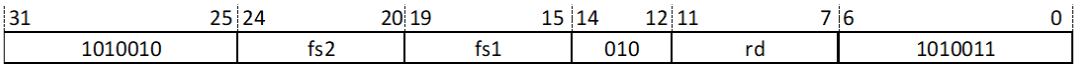
Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV

Instruction format:



17.6.17 FLE.H——The Compare-if-less-than-or-equal-to Instruction of Half-precision Floating-Point Numbers

Syntax:

fle.h rd, fs1, fs2

Operation:

if(fs1 <= fs2)

rd ← 1

else

rd ← 0

Execute permission:

M-mode/Smode/U-mode

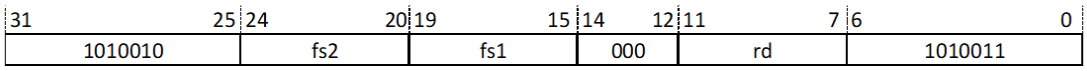
Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV

Instruction format:



17.6.18 FLH——The Half-precision Floating-point Load Instruction

Syntax:

flh fd, imm12(rs1)

Operation:

address←rs1+sign_extend(imm12)

fd[15:0] ← mem[(address+1):address]

fd[63:16] ← 48’ hffffffffffff

Execute permission:

M-mode/Smode/U-mode

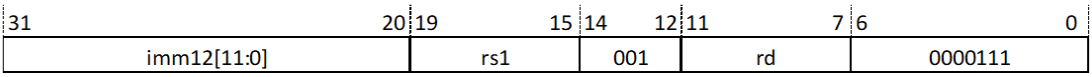
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for load instructions, and the illegal instruction exception.

Affected flag:

None

Instruction format:



17.6.19 FLT.H——The Compare-if-less-than Instruction of Half-precision Floating-Point Numbers

Syntax:

flt.h rd, fs1, fs2

Operation:

if(fs1 < fs2)

rd ← 1

else

rd ← 0

Execute permission:

M-mode/Smode/U-mode

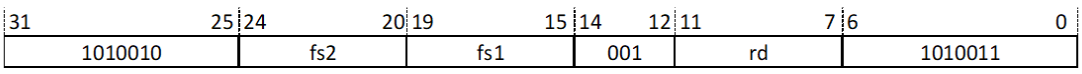
Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV

Instruction format:



17.6.20 FMADD.H——The Half-precision Floating-point Multiply-add Instruction

Syntax:

fmadd.h fd, fs1, fs2, fs3, rm

Operation:

fd ← fs1*fs2 + fs3

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

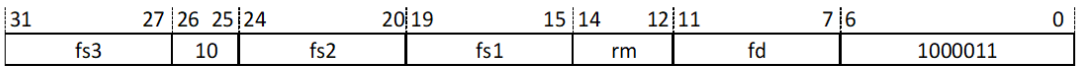
Floating-point status bit NV/OF/UF/IX

Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fmadd.h fd,fs1, fs2, fs3, rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fmadd.h fd,fs1, fs2, fs3, rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fmadd.h fd,fs1, fs2, fs3, rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fmadd.h fd,fs1, fs2, fs3, rup.
- 3' b100: Rounds to the nearest large value. And the corresponding assembly instruction is fmadd.h fd,fs1, fs2, fs3, rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fmadd.h fd,fs1, fs2, fs3.

Instruction format:



17.6.21 FMAX.H——The Half-precision Floating-point Maximum Instruction

Syntax:

fmax.h fd, fs1, fs2

Operation:

if(fs1 >= fs2)

fd ← fs1

else

fd ← fs2

Execute permission:

M-mode/Smode/U-mode

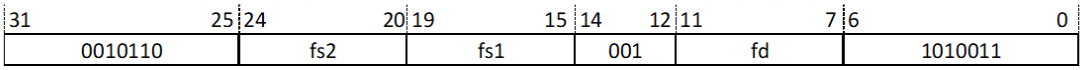
Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV

Instruction format:



17.6.22 FMIN.H——The Half-precision Floating-point Minimum Instruction

Syntax:

fmin.h fd, fs1, fs2

Operation:

if(fs1 >= fs2)

fd ← fs2

else

fd ← fs1

Execute permission:

M-mode/Smode/U-mode

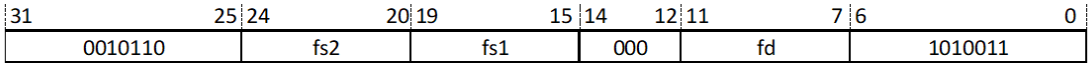
Exception:

The illegal instruction exception

Affected flag:

Floating-point status bit NV

Instruction format:



17.6.23 FMSUB.H——The Half-precision Floating-point Multiply-subtract Instruction

Syntax:

fmsub.h fd, fs1, fs2, fs3, rm

Operation:

fd ← fs1*fs2 - fs3

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

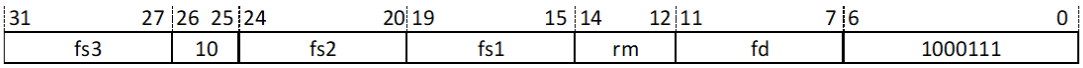
Floating-point status bit NV/OF/UF/IX

Note:

rm determines the rounding mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembly instruction is fmsub.h fd,fs1, fs2, fs3, rne.
- 3’ b001: Rounds to zero. And the corresponding assembly instruction is fmsub.h fd,fs1, fs2, fs3, rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembly instruction is fmsub.h fd,fs1, fs2, fs3, rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembly instruction is fmsub.h fd,fs1, fs2, fs3, rup.
- 3’ b100: Rounds to the nearest large value. And the corresponding assembly instruction is fmsub.h fd, fs1, fs2, fs3,rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fmsub.h fd,fs1, fs2, fs3.

Instruction format:



17.6.24 FMUL.H——The Half-precision Floating-point Multiply Instruction

Syntax:

fmul.h fd, fs1, fs2, rm

Operation:

fd ← fs1 * fs2

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

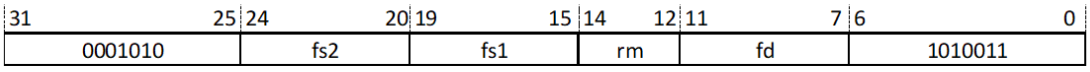
Floating-point status bit NV/OF/UF/NX

Note:

rm determines the rounding mode:

- 3' b000: Rounds to the nearest even number. And the corresponding assembly instruction is fmul.h fd, fs1, fs2, rne.
- 3' b001: Rounds to zero. And the corresponding assembly instruction is fmul.h fd, fs1, fs2, rtz.
- 3' b010: Rounds to negative infinity. And the corresponding assembly instruction is fmul.h fd, fs1, fs2, rdn.
- 3' b011: Rounds to positive infinity. And the corresponding assembly instruction is fmul.h fd, fs1, fs2, rup.
- 3' b100: Rounds to the nearest large value. And the corresponding assembly instruction is fmul.h fd, fs1,fs2 , rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fmul.h fs1,fs2.

Instruction format:



17.6.25 FMV.H.X——The Half Precision Floating-point Write Transfer Instruction

Syntax:

fmv.h.x fd, rs1

Operation:

fd[15:0] ← rs1[15:0]

fd[63:16] ← 48’ hffffffffffff

Execute permission:

M-mode/Smode/U-mode

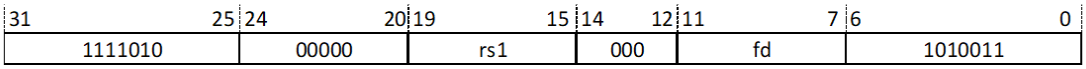
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



17.6.26 FMV.X.H——The Half Precision Floating-point Read Transfer Instruction

Syntax:

fmv.x.h rd, fs1

Operation:

tmp[15:0] ← fs1[15:0]

rd ← sign_extend(tmp[15:0])

Execute permission:

M-mode/Smode/U-mode

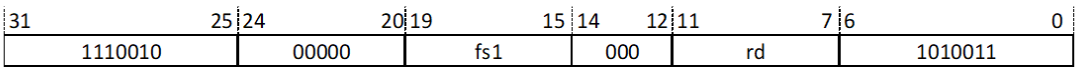
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



17.6.27 FNMADD.H——The Half-precision Floating-point Negate-(Multiply-add) Instruction

Syntax:

fnmadd.h fd, fs1, fs2, fs3, rm

Operation:

fd ←-(fs1*fs2 + fs3)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

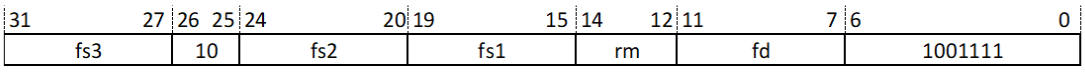
Floating-point status bit NV/OF/UF/IX

Note:

rm determines the rounding mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembly instruction is fnmadd.h fd,fs1, fs2, fs3, rne.
- 3’ b001: Rounds to zero. And the corresponding assembly instruction is fnmadd.h fd,fs1, fs2, fs3, rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembly instruction is fnmadd.h fd,fs1, fs2, fs3, rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembly instruction is fnmadd.h fd,fs1, fs2, fs3, rup.
- 3’ b100: Rounds to the nearest large value. And the corresponding assembly instruction is fnmadd.h fd,fs1, fs2, fs3, rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fnmadd.h fd,fs1, fs2, fs3.

Instruction format:



17.6.28 FNMSUB.H——The Half-precision Floating-point Negate-(Multiply-subtract) Instruction

Syntax:

fnmsub.h fd, fs1, fs2, fs3, rm

Operation:

fd ← -(fs1*fs2 - fs3)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

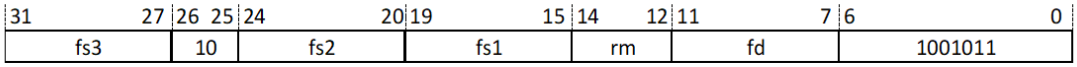
Floating-point status bit NV/OF/UF/IX

Note:

rm determines the rounding mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembly instruction is fnmsub.h fd,fs1, fs2, fs3, rne.
- 3’ b001: Rounds to zero. And the corresponding assembly instruction is fnmsub.h fd,fs1, fs2, fs3, rtz.
- 3’ b010: Rounds to negative infinity. And the corresponding assembly instruction is fnmsub.h fd,fs1, fs2, fs3, rdn.
- 3’ b011: Rounds to positive infinity. And the corresponding assembly instruction is fnmsub.h fd,fs1, fs2, fs3, rup.
- 3’ b100: Rounds to the nearest large value. And the corresponding assembly instruction is fnmsub.h fd,fs1, fs2, fs3, rmm.
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fnmsub.h fd,fs1, fs2, fs3.

Instruction format:



17.6.29 FSGNJ.H——The Half-precision Floating-point Sign-injection Instruction

Syntax:

fsgnj.h fd, fs1, fs2

Operation:

fd[14:0] ← fs1[14:0]

fd[15] ← fs2[15]

fd[63:16] ← 48’ hffffffffffff

Execute permission:

M-mode/Smode/U-mode

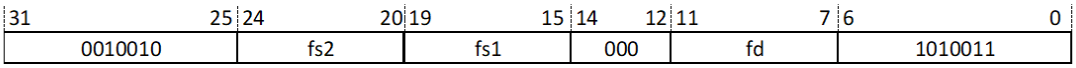
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



17.6.30 FSGNJN.H——The Half-precision Floating-point Sign-injection Negate Instruction

Syntax:

fsgnjn.h fd, fs1, fs2

Operation:

fd[14:0] ← fs1[14:0]

fd[15] ← ! fs2[15]

fd[63:16] ← 48' hffffffffff

Execute permission:

M-mode/Smode/U-mode

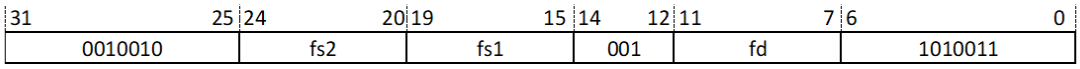
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



17.6.31 FSGNJX.H——The Half-precision Floating-point Sign XOR Injection Instruction

Syntax:

fsgnjx.h fd, fs1, fs2

Operation:

fd[14:0] ← fs1[14:0]

fd[15] ← fs1[15] ^ fs2[15]

fd[63:16] ← 48' hffffffffff

Execute permission:

M-mode/Smode/U-mode

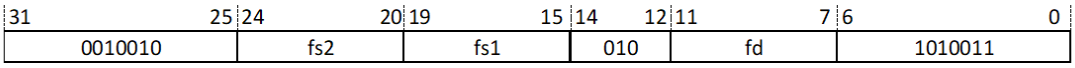
Exception:

The illegal instruction exception

Affected flag:

None

Instruction format:



17.6.32 FSH——The Half-precision Floating-point Store Instruction

Syntax:

fsh fs2, imm12(fs1)

Operation:

address←fs1+sign_extend(imm12)

mem[(address+1):address] ← fs2[15:0]

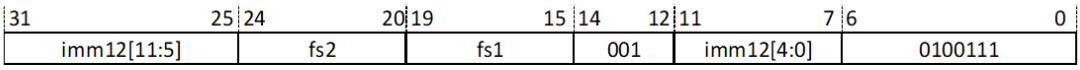
Execute permission:

M-mode/Smode/U-mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions for store instructions, and the illegal instruction exception.

Instruction format:



17.6.33 FSQRT.H——The Square Root Instruction of Half-precision Floating-point

Syntax:

fsqrt.h fd, fs1, rm

Operation:

fd ← sqrt(fs1)

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

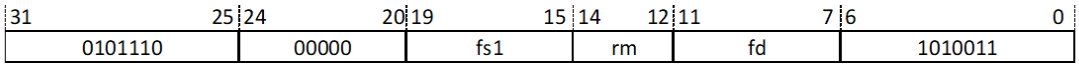
Floating-point status bit NV/NX

Note:

rm determines the rounding mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembly instruction is fsqrt.h fd, fs1,rne
- 3’ b001: Rounds to zero. And the corresponding assembly instruction is fsqrt.h fd, fs1,rtz
- 3’ b010: Rounds to negative infinity. And the corresponding assembly instruction is fsqrt.h fd, fs1,rdn
- 3’ b011: Rounds to positive infinity. And the corresponding assembly instruction is fsqrt.h fd, fs1,rup
- 3’ b100: Rounds to the nearest large value. And the corresponding assembly instruction is fsqrt.h fd, fs1,rmm
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fsqrt.h fd, fs1.

Instruction format:



17.6.34 FSUB.H——The Half-precision Floating-point Subtract Instruction

Syntax:

fsub.h fd, fs1, fs2, rm

Operation:

fd ← fs1 - fs2

Execute permission:

M-mode/Smode/U-mode

Exception:

The illegal instruction exception

Affected flag:

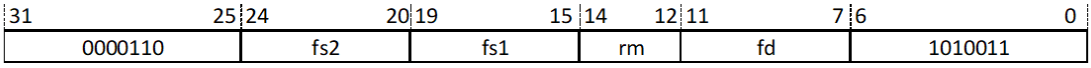
Floating-point status bit NV/OF/NX

Note:

rm determines the rounding mode:

- 3’ b000: Rounds to the nearest even number. And the corresponding assembly instruction is fsub.h fd, fs1,fs2,rne
- 3’ b001: Rounds to zero. And the corresponding assembly instruction is fsub.h fd, fs1,fs2,rtz
- 3’ b010: Rounds to negative infinity. And the corresponding assembly instruction is fsub.h fd, fs1,fs2,rdn
- 3’ b011: Rounds to positive infinity. And the corresponding assembly instruction is fsub.h fd, fs1,fs2,rup
- 3’ b100: Rounds to the nearest large value. And the corresponding assembly instruction is fsub.h fd, fs1,fs2,rmm
- 3’ b101: This code is reserved and not used.
- 3’ b110: This code is reserved and not used.
- 3’ b111: Dynamic rounding, which determines the rounding mode based on the rm bit in the floating-point control register fcsr. And the corresponding assembly instruction is fsub.h fd, fs1,fs2.

Instruction format:



CHAPTER 18

Appendix C System Control Registers

This section describes the Machine Mode (M-mode) control registers, Supervisor Mode (S-mode) control registers, and User Mode (U-mode) control registers in details.

18.1 Appendix C-1 RISC-V Standard Machine Mode Control and Status Registers

18.1.1 M-mode Information Register Group

18.1.1.1 M-mode Vendor ID register (MVENDORID)

The MVENDORID register stores the vendor IDs of Xuantie CPU, currently bound to 0x5B7.

This register is 64-bit wide and read-only in M-mode. Accesses in non-machine mode and writes in Machine Mode (M-mode) will cause an illegal instruction exception.

18.1.1.2 M-mode Architecture ID register (MARCHID)

The MARCHID register stores the architecture IDs of CPU cores. It stores the internal IDs of Xuantie CPU and its reset value is subject to the product.

This register is 64-bit wide and read-only in M-mode. Accesses in non-machine mode and writes in M-mode will cause an illegal instruction exception.

18.1.1.3 M-mode Implementation ID register (MIMPID)

The MIMPID register stores hardware implementation IDs of CPU cores. This register is not implemented by C920 currently, and tits read access value is 0.

This register is 64-bit wide and is read-only in M-mode. Accesses in non-machine mode and writes in M-mode will cause an illegal instruction exception.

18.1.1.4 M-mode Hart ID Register (MHARTID)

MHARTID stores the hardware logic core number of CPU cores.

This register is 64-bit wide and is read-only in M-mode. Accesses in non-machine mode and writes in M-mode will cause an illegal instruction exception.

18.1.1.5 M-mode Configuration Data Structure Pointer (MCONFIGPTR)

MCONFIGPTR stores the physical address corresponding to the configuration data structure. When the value of this register is zero, it indicates that the data structure does not exist.

This register is 64-bit wide and is read-only in M-mode. Accesses in non-machine mode and writes in M-mode will cause an illegal instruction exception.

18.1.2 M-mode Exception Configuration Register Group

18.1.2.1 M-Mode Status Register (MSTATUS)

The MSTATUS register stores status and control information of the CPU in M-mode, including the global interrupt enable bit, exception preserve interrupt enable bit, exception preserve privilege mode bit and so on.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

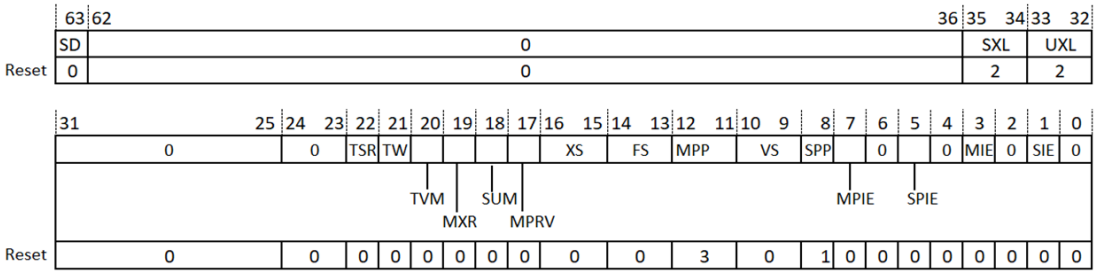


Fig. 18.1: **M-mode Status Register (MSTATUS)**

SIE——S-mode interrupt enable bit:

- When SIE is set to 0, S-mode interrupts are invalid.
- When SIE is set to 1, S-mode interrupts are valid.

This bit is reset to 0 when the CPU is delegated to the S-mode in response to interrupts, and is set to the value of SPIE when the CPU exits the interrupt service routine (ISR).

MIE——M-mode interrupt enable bit:

- When MIE is set to 0, M-mode interrupts are invalid.
- When MIE is set to 1, M-mode interrupts are valid.

This bit is reset to 0 when the response is interrupted in M-mode on the CPU, and is set to the value of MPIE when the CPU exits the ISR.

SPIE——S-mode preserved interrupt enable bit:

This bit stores the value of the SIE bit before the CPU responds to an interrupt in S-mode.

This bit will be reset to 0, and set to 1 when the CPU exits the ISR.

MPIE——M-mode preserved interrupt enable bit:

This bit stores the value of the MIE bit before the CPU responds to an interrupt in M-mode.

This bit will be reset to 0, and set to 1 when the CPU exits the ISR.

SPP——S-mode privileged preserved status bit:

This bit stores the privilege status before the CPU accesses the exception service program in S-mode.

- When SPP is 1' b0, the CPU is in User Mode (U-mode) before accessing the exception service program.
- When SPP is 1' b1, the CPU is in S-mode before accessing the exception service program.

This bit will be reset to 1' b1.

MPP——S-mode privileged preserved status bit:

This bit stores the privilege status before the CPU accesses the exception service program in M-mode.

- When MPP is 2' b00, the CPU is in U-mode before entering the exception service program.
- When MPP is 2' b01, the CPU is in S-mode before accessing the exception service program.
- When MPP is 2' b11, the CPU is in M-mode before entering the exception service program.

This bit will be reset to 2' b11.

FS——Floating-point status bit

This bit determines whether to store floating-point registers during context switching.

- When FS is 2' b00, the floating-point unit is in the Off state and exceptions will occur for accesses to the related floating-point registers.
- When FS is 2' b01, the floating-point unit is in the Initial state.
- When FS is 2' b10, the floating-point unit is in the Clean state.

- When FS is 2’ b11, the floating-point unit is in the Dirty state, which indicates the floating-point register and CSRs have been modified.

XS——Extended unit status bit:

Extension units are not available in C920, and therefore this bit is fixed to 0.

MPRV——Modify privilege mode:

- When MPRV is set to 1, load and store requests are executed based on the privilege mode in MPP.
- When MPRV is set to 0, load and store requests are executed based on the current privilege mode of the CPU.

SUM——Allow S-mode accesses to U-mode virtual memory spaces:

- When SUM is set to 1, load, store, and fetch requests can be initiated in S-mode to access virtual memory areas that are marked as U-mode.
- When SUM is set to 0, load, store, and fetch requests cannot be initiated in S-mode to access virtual memory areas that are marked as U-mode.

MXR——Allow accesses of load requests to memory spaces marked as executable:

- When MXR is set to 1, accesses of load requests are allowed to virtual memory spaces marked as executable or readable.
- When MXR is set to 0, accesses of load requests are allowed only to virtual memory spaces marked as readable.

TVM——Trap into virtual memory:

-When TVM is set to 1, an illegal instruction exception occurs for reads and writes to the satp CSRs and for the execution of the sfence instruction in S-mode.

- When TVM is set to 0, reads and writes to the satp CSRs and the execution of the sfence instruction are allowed in S-mode.

TW——Timeout wait:

- When TW is set to 1, an illegal instruction exception occurs if the WFI instruction is executed in S-mode.
- When TW is set to 0, the WFI instruction can be executed in S-mode.

TSR——Trap sret:

- When TSR is set to 1, an illegal instruction exception occurs if the sret instruction is executed in S-mode.
- When TSR is set to 0, the sret instruction can be executed in S-mode.

VS——Vector status bit:

VS bit determines whether to store vector registers during context switching.

- When VS is set to 2’ b00, the vector unit is in the Off state and exceptions will occur for accesses to related vector registers.
- When VS is set to 2’ b01, the vector unit is in the Initial state.

- When VS is set to 2’ b10, the vector unit is in the Clean state.
- When VS is set to 2’ b11, the vector unit is in the Dirty state, which indicates the vector registers and vector CSRs have been modified.

The VS bit is valid only when the vector execution unit is configured, otherwise it is always 0.

UXL——Register width:

This bit is read-only and the fixed value is 2, indicating the register is 64-bit wide in U-mode.

SXL——Register width:

This bit is read-only and the fixed value is 2, indicating the register is 64-bit wide in S-mode.

SD——The dirty state sum bit of the floating-point, vector, and extension units:

- When SD is set to 1, the floating-point unit, vector unit, or extension unit is in the Dirty state.
- When SD is set to 0, none of the floating-point, vector, and extension units is in the Dirty state.

18.1.2.2 M-mode Instruction Set Architecture Register (MISA)

The misa register stores the features of the instruction set architecture supported by the CPU.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

C920 supports the RV64GC instruction set architecture, and the reset value of the MISA register is set to 0x8000000000b4112f。For detailed information about the assignment rules, please refer to the official document of RISC-V——*riscv-privileged* .

C920 does not support the dynamic configuration of the MISA register and writes to this register do not take effect.

18.1.2.3 M-mode Exception Degradation Register (MEDELEG)

The MEDELEG register can delegate exceptions that occur in S-mode and U-mode to S-mode responses. The lower 16 bits of the MEDELEG register are in one-to-one correspondence to exception vector tables, allowing for the selection of which exceptions can be delegated to be handled in Supervisor Mode.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

18.1.2.4 M-mode Interrupt Downgrade register (MIDELEG)

The MIDELEG register can delegate S-mode interrupts to S-mode responses.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

MCIE_DELEG——M-mode ECC interrupt:

- When MCIE_DELEG is set to 1, ECC interrupts can be handled when CPU is delegated to S-mode.

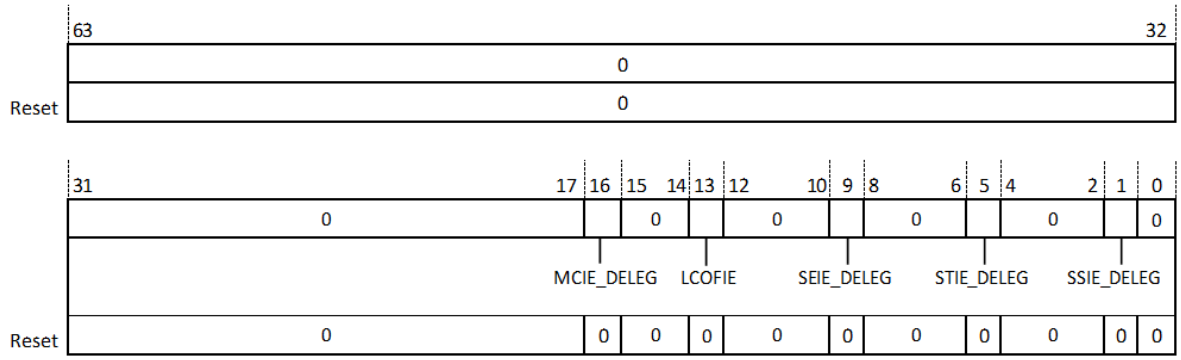


Fig. 18.2: **M-mode Interrupt Downgrade Register (MIDELEG)**

- When MCIE_DELEG is set to 0, ECC interrupts can only be handled as cpu is delegated to M-mode.

LCOFIE——Performance Monitoring Unit (PMU) Event Counter Overflow Interrupt:

- When LCOFIE is set to 1, PMU Event Counter Overflow Interrupt can be handled when the processor is delegated to S-mode.
- When LCOFIE is set to 0, PMU Event Counter Overflow Interrupt can only be handled as cpu is delegated to M-mode.

SEIE_DELEG——S-mode external interrupt:

- When SEIE_DELEG is set to 1, S-mode external interrupt can be handled when the CPU is delegated to S-mode.
- When SEIE_DELEG is set to 0, S-mode external interrupt can only be handled as cpu is delegated to M-mode.

STIE_DELEG——S-mode timer interrupt

- When STIE_DELEG is set to 1, S-mode timer interrupt can be handled when the CPU is delegated to S-mode.
- When STIE_DELEG is set to 0, S-mode timer interrupt can only be handled as cpu is delegated to M-mode.

SSIE_DELEGG——S-mode software interrupt:

- When SSIE_DELEG is set to 1, S-mode software interrupt can be handled when the CPU is delegated to S-mode.
- When SSIE_DELEG is set to 0, S-mode software interrupt can only be handled as cpu is delegated to M-mode.

18.1.2.5 M-mode Interrupt Enable Register (MIE)

The MIE register enables and masks different types of interrupt.This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

SSIE——S-mode software interrupt enable bit:

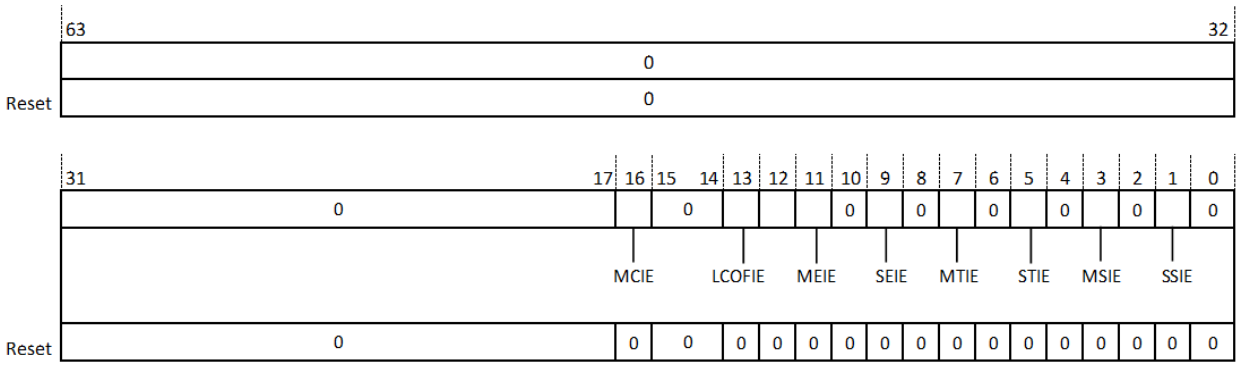


Fig. 18.3: **M-mode Interrupt Enable Register (MIE)**

- When SSIE is set to 0, S-mode software external interrupts are invalid.
- When SSIE is set to 1, S-mode software external interrupts are valid.

MSIE——M-mode software interrupt enable bit:

- When MSIE is set to 0, M-mode software interrupts are invalid.
- When MSIE is set to 1, M-mode software interrupts are valid.

STIE——S-mode timer interrupt enable bit:

- When STIE is set to 0, S-mode timer interrupts are invalid.
- When STIE is set to 1, S-mode timer external interrupts are valid.

MTIE——M-mode timer interrupt enable bit:

- When MTIE is set to 0, M-mode timer interrupts are invalid.
- When MTIE is set to 1, M-mode timer interrupts are valid.

SEIE——S-mode external interrupt enable bit:

- When SEIE is set to 0, S-mode external interrupts are invalid.
- When SEIE is set to 1, S-mode external interrupts are valid.

MEIE——M-mode external interupt enable bit:

- When MEIE is set to 0, M-mode external interrupts are invalid.
- When MEIE is set to 1, M-mode external interrupts are valid.

LCOFIE——M-mode event counter overflow interrupt enable bit:

- When LCOFIE is set to 0, M-mode counter overflow interrupts are invalid.
- When LCOFIE is set to 1, M-mode counter overflow interrupts are valid.

MCIE——M-mode ECC interrupt enable bit:

- When MCIE is set to 0, M-mode ECC interrupts are invalid.
- When MCIE is set to 1, M-mode ECC interrupts are valid.

18.1.2.6 M-mode Vector Base Address (MTVEC)

The MTVEC stores the entry address of the exception service program.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

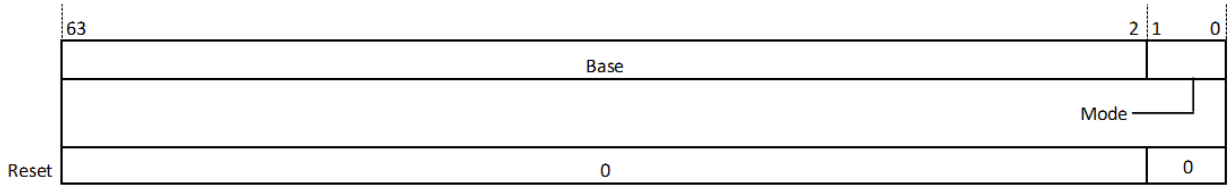


Fig. 18.4: **M-mode Vector Base Address (MTVEC)**

BASE——Vector base address bit:

The BASE bit indicates the upper 62 bits of the entry address of the exception service program. Combining this base address with 2' b00 obtains the entry address of the exception service program.

This bit will be reset to 0.

MODE——Vector entry mode bit:

- When MODE[1:0] is set to 2' b00, the base address is applied as the entry address for both exceptions and interrupts.
- When MODE[1:0] is set to 2' b01, the base address is applied as the entry address for exceptions, while BASE + 4*cause is used as the entry address for interrupts

18.1.2.7 M-Mode Counter Enable Register (MCOUNTEREN)

The mcounteren register determines whether U-mode counters can be accessed in S-mode.

For detailed information, please refer to *Mcounteren Register* .

18.1.3 M-mode Exception Handling Register Group

18.1.3.1 Machine Mode Scratch Register for Exception Temporal Data Backup (MSCRATCH)

The MSCRATCH is applied in exception service routines for the backup of temporary data and to store the entry pointer value of local context space in M-mode in general.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

18.1.3.2 M-mode Exception program counter register (MEPC)

The MEPC register stores the program counter value (PC value) when the CPU exits from the exception service program. C920 supports 16-bit wide instructions. The MEPC value is aligned with 16 bits and the lowest bit 0.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

18.1.3.3 M-Mode Exception Cause Register (MCAUSE)

The MCAUSE register stores the vector numbers of events that trigger exceptions, to handle corresponding events in the exception service program.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

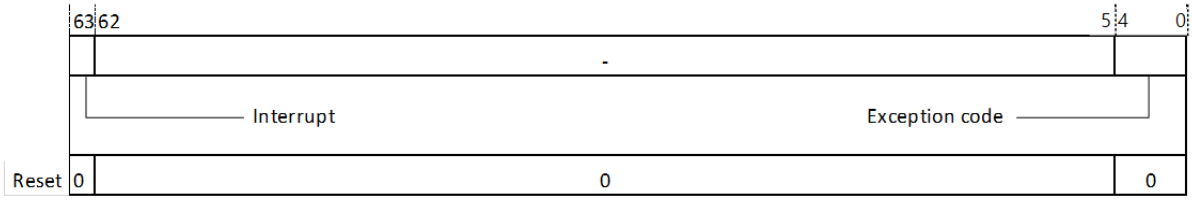


Fig. 18.5: **M-Mode Exception Cause Register (MCAUSE)**

Interrupt——Interrupt flag:

- When the Interrupt bit is set to 0, the source of the triggering exception is not an interrupt; instead, the Exception Code is interpreted according to the exception resolution process.
- When the Interrupt bit is set to 1, the corresponding exception is triggered by an interrupt. The exception code is is interpreted according to interrupt resolution.

Exception Code——Exception vector number:

When the CPU encounters an exception, the Exception Code bit will be updated to the value of the exception source.

18.1.3.4 Machine Trap Value Register (MTVAL)

The MTVAL register is designed to store the cause of the exception event that triggered it, such as the address where the exception occurred or the instruction code, for processing the corresponding event in the exception service program.

This register is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception. The bit width of the register is MXLEN.

18.1.3.5 M-mode Interrupt Pending Register (MIP)

The MIP register stores THE pending interrupt information. When the CPU cannot immediately respond to an interrupt, the corresponding bit in the mip register will be set.

Writing the MSIP and SSIP registers in the CLINT interrupt controller can trigger corresponding interrupts. After the interrupts become valid, the MSIP bit and SSIP bit can be queried based on the corresponding bits in the MIP register.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

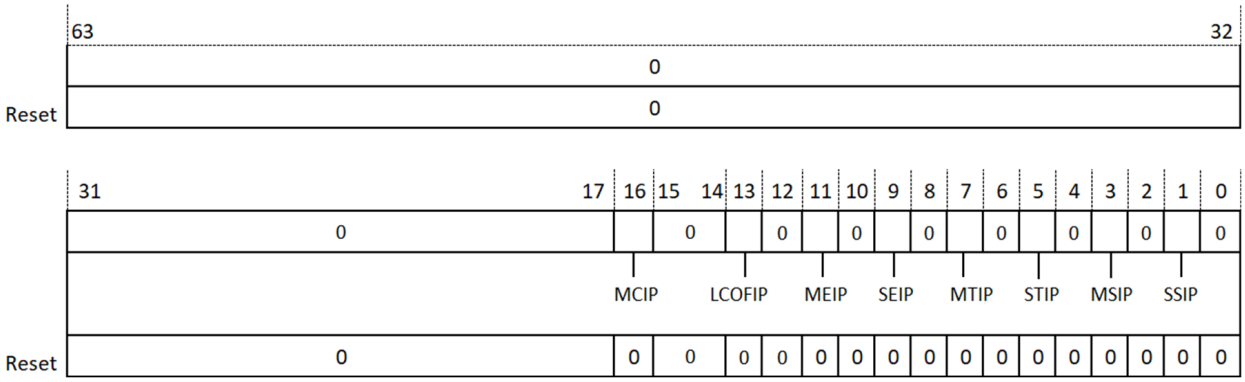


Fig. 18.6: **M-mode Interrupt Pending Register (MIP)**

SSIP——S-mode software interrupt pending bit:

- When SSIP is set to 0, there is no pending S-mode software interrupt on the CPU.
- When SSIP is set to 1, there are pending S-mode software interrupts on the CPU.

The SSIP bit is readable and writable in M-mode. After it is delegated to S-mode, it is readable and writable in S-mode. Otherwise, it is read-only in S-mode.

MSIP——M-mode software interrupt pending bit:

- When MSIP is set to 0, there is no pending M-mode software interrupt on the CPU.
- When MSIP is set to 1, there are pending M-mode software interrupts on the CPU.

This bit is read-only.

STIP——S-mode timer interrupt pending bit:

- When STIP is set to 0, there is no pending S-mode timer interrupt on the CPU.
- When STIP is set to 1, there are pending S-mode timer interrupts on the CPU.

MTIP——M-mode timer interrupt pending bit:

- When MTIP is set to 0, there is no pending M-mode timer interrupt on the CPU.

- When MTIP is set to 1, there are pending M-mode timer interrupts on the CPU.

SEIP——S-mode external interrupt pending bit:

- When SEIP is set to 0, there is no pending S-mode external interrupt on the CPU.
- When SEIP is set to 1, there are pending S-mode external interrupts on the CPU.

MEIP——M-mode external interrupt pending bit:

- When MEIP is set to 0, there is no pending M-mode external interrupt on the CPU.
- When MEIP is set to 1, there are pending M-mode external interrupts on the CPU.

LCOFIP——M-mode event counter overflow interrupt pending bit:

- When MEIP is set to 0, there is no pending M-mode counter overflow interrupt on the CPU.
- When MEIP is set to 1, there are pending M-mode counter overflow interrupts on the CPU.

MCIP——M-mode ECC interrupt pending bit:

- When MEIP is set to 0, there is no pending M-mode ECC interrupt on the CPU.
- When MEIP is set to 1, there are pending M-mode ECC interrupts on the CPU.

18.1.4 M-Mode Environment Configuration Register Group

18.1.4.1 M-Mode Environment Configuration Registe (MENVCFG)

	63	62	61 — 8	7	6	5 — 4	3 — 1	0
	STCE	PBMTE	0	CBZE	CBCFE	CBIE	0	FIOM
Reset	0	0	0	0	0	0	0	0

Fig. 18.7: **M-Mode Environment Configuration Register (MENVCFG)**

The MENVCFG register is designed to control the characteristics of the execution environment when running at privilege levels lower than M-mode.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

STCE——S-Mode timer interrupt comparator value register (STIMECMP) enable bit:

When STCE is set to 0 and the upper/lower bits of STCE in CLINT——(STIMECMPH/STIMECMPL) are less or equal to the current value of the system timer, a S-mode timer interrupt is generated. STIMECMPH/STIMECMPL are memory-mapped registers.

When STCE is set to 1, the STIMECMP register is less or equal to the current value of the system timer, a S-mode timer interrupt is generated. STIMECMPH is CSR registers.

PBMTE——Svpbmt enable bit:

When PBMTE is set to 0, Svpbmt is disabled; When PBMTE is set to 1, Svpbmt is enabled.

CBZE——CBO.ZERO instruction enable bit:

When CBZE is set to 0, executing the CBO.ZERO instruction results in an illegal instruction exception in a lower privilege mode.

When CBZE is set to 1, CBO.ZERO instruction is normally executed in a lower privilege mode.

CBCFE——CBO.CLEAN and CBO.FLUSH instruction enable bits:

When CBCFE is set to 0, executing the CBO.CLEAN and CBO.FLUSH instructions result in illegal instruction exceptions in a lower privilege mode.

When CBCFE is set to 1, CBO.CLEAN and CBO.FLUSH instructions are normally executed in a lower privilege mode.

CBIE——CBO.INVAL instruction enable bit:

When CBIE is set to 0, executing the CBO.INVAL instruction results in an illegal instruction exception in a lower privilege mode.

When CBIE is set to 1, CBO.INVAL instruction is executed according to the CBO.FLUSH instruction.

When CBIE is set to 2, which is a reserved value and should not be configured;

When CBIE is set to 3, CBO.INVAL instruction is normally executed in a lower privilege mode.

FIOM——IO fence including memory accesses

In terms of RV standard definition, when menvcfg.FIOM is set to 1, IO synchronization requests at a level below M-mode must also include memory Read and Write (RW) synchronization.

Note:

In Xuantie C920, all fence synchronizations for I/O inherently encompass memory RW synchronization, no longer controlled by this particular bit. Fence synchronizations for I/O will always include Read-Write synchronization regardless of the value of this bit.

18.1.4.2 M-mode Secure Configuration Register (MSECCFG/MSECCFGH)

The MSECCFG/MSECCFGH register is to record execution secure configuration information. The M-mode extended security configuration register is primarily designed to extend Physical Memory Protection (PMP) permission rules, and facilitate memory access protection (MAP) and memory execution protection (MEP) among M-mode, S-mode and U-mode. This register is exclusively accessible in M-mode.

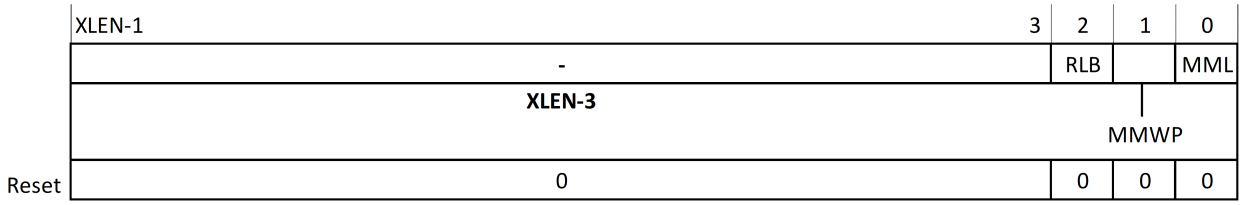


Fig. 18.8: **MSECCFG Register**

RLB: Rule Locking Bypass

- When mseccfg.RLB is set to 1, the pmpcfg.L bits can be edited, which means that even if a PMP table entry is locked, it can be unlocked for modification when RLB = 1, allowing the entry' s rules to be revised.
- If any entries in the PMP table entries are locked (pmpcfg.L = 1) when RLB = 0, the mseccfg.RLB bit also becomes locked and unmodifiable, and can only be reset to its initial value through a hardware reset.

MMWP: Machine Mode Whitelist Policy

- Once the mseccfg.MMWP bit is set to 1, it becomes unmodifiable, and can only be reset to its initial value through a hardware reset.
- When mseccfg.MMWP is set to 1, M-mode only permits accesses according to the rules defined in the PMP configuration table, denying any access attempts not covered by these rules.

MML: Machine Mode Lockdown

- When mseccfg.MML is set to 1, it becomes locked and unmodifiable and can only be reset to its initial value through a hardware reset.
- When mseccfg.MML is set to 1, the access rule definitions in PMP table entries are illustrated in Table 18.1 :

Table 18.1: **Permission Rules of PMP Table Entries with mseccfg.MML=1**

Bits on pmpcfg Register				Result	
L	**R**	**W**	**X**	**M Mode**	**S/U Mode**
0	0	0	0	Inaccessible region (Access Exception)	
0	0	0	1	Access Exception	Execute-only region
0	0	1	0	Shared data region: Read/write on M mode, read-only on S/U mode	
0	0	1	1	Shared data region: Read/write for both M and S/U mode	
0	1	0	0	Access Exception	Read-only region
0	1	0	1	Access Exception	Read/Execute region
0	1	1	0	Access Exception	Read/Write region
0	1	1	1	Access Exception	Read/ Write/Execute region
1	0	0	0	Locked inaccessible region* (Access Exception)	
1	0	0	1	Locked Execute-only region*	Access Exception
1	0	1	0	Locked Shared code region: Execute only on both M and S/U mode.*	
1	0	1	1	Locked Shared code region: Execute only on S/U mode, read/execute on M mode.*	
1	1	0	0	Locked Read-only region*	Access Exception
1	1	0	1	Locked Read/Execute region*	Access Exception
1	1	1	0	Locked Read/Write region*	Access Exception
1	1	1	1	Locked Shared data region: Read only on both M and S/U mode.*	

*: Locked entries cannot be removed or modified until a hard reset, unless mseccfg.RLB is set.

For detailed information, please refer to RISC-V PMP Enhancements for memory access and execution prevention on Machine mode (Smepmp)

18.1.5 M-mode Memory Protection Register Group

M-mode memory protection register group is associated with configuring the Memory Protection Unit (MPU).

18.1.5.1 M-mode Physical Memory Protection Configuration Registe (PMPCFG)

The PMPCFG register is designed to configure the access permissions and address matching modes of physical memory.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

For detailed information, please refer to *PMPCFG Register* .

18.1.5.2 M-mode Physical Memory Protection Address Register (PMPADDR)

The PMPADDR register is designed to configure the address range for each entry in the physical memory table.

This register is 64-bit wide and is readable and writable in M-mode. Accesses in non-machine mode will cause an illegal instruction exception.

For detailed information, please refer to *PMPADDR Register* .

18.1.6 M-mode Timer and Counter Register Group

M-mode counter register group belongs to Performance Monitor Unit(PMU), applied to collect software information and certain hardware information during program execution, to assist software developers in optimizing their program

18.1.6.1 M-mode Cycle Counter (MCYCLE)

The MCYCLE register stores the number of cycles already executed by the processor. When the processor is in an execution state (i.e., not in a low-power state), the MCYCLE register increments its count on each clock cycle.

The MCYCLE counter is 64-bit wide and will be reset to 0.

For detailed information, please refer to *Event Counters* .

18.1.6.2 M-Mode Instruction Retire Counter (MINSTRET)

The MINSTRET register stores the number of retired instructions of the CPU. The MINSTRET register increments its count on each instruction retirement.

The minstret counter is 64-bit wide and will be reset to 0.

For detailed information, please refer to *Event Counters* 。

18.1.6.3 M-mode Event Counter (MHPMCOUNTERn)

The mhpmcountern counter counts events.

The mhpmcountern counter is 64-bit wide and will be reset to 0.

For detailed information, please refer to *Event Counters* 。

18.1.7 M-mode Counter Configuration Register Group

The M-mode counter configuration register selects events for M-mode event counters.

18.1.7.1 M-Mode Counter Inhibit Register (MCOUNTINHIBIT)

The MCOUNTINHIBIT is designed to disable m-mode event counters. And disabling these counters can reduce processor power consumption in situations where performance monitoring is unnecessary.

For detailed information, please refer to *Mcountinhibit Register* .

18.1.7.2 M-mode Performance Monitor Event Select Register (MHPMEVENTn)

The M-mode performance monitor event select register (mhpmevent3-31) is applied to select the counting event corresponding to each counter. Each counter in C920 supports the configuration of any event. The counter can normally count for the configured events by writing the event index value to the performance monitoring event selection register

For detailed information, please refer to *M-mode Performance Monitor Event Select Register* .

18.1.8 Debug/Trace Register Group (Shared with Debug Mode)

18.1.8.1 Debug/Trace Trigger Selection Register (TSELECT)

The TSELECT register is designed to select one from multiple available triggers, to read from/write to the registers of the trigger.

	63 ... 4	3 ... 0
	-	SELECT
Reset	-	0

Fig. 18.9: **Debug/Trace Trigger Selection Register(TSELECT)**

SELECT——Debug/Trace Trigger Selection

- Record the currently selected debug/trace trigger number. For example, to configure Trigger Number 2, write 0x2 into the SELECT register.

18.1.8.2 Debug/Trace Trigger Data Register 1 (TDATA1)

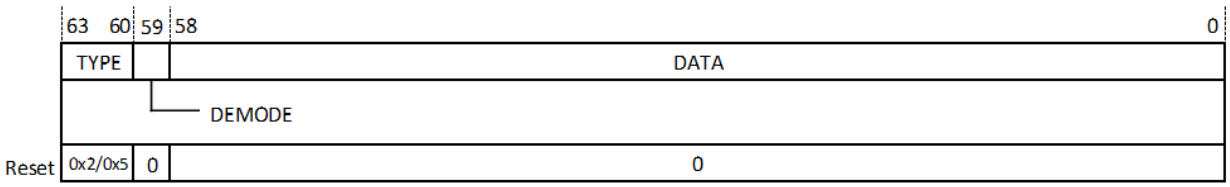


Fig. 18.10: **Debug/Trace Trigger Data Register 1 (TDATA1)**

TYPE——Debug/Trace Trigger Type Selection

- TYPE = 2, indicating the current trigger type is mcontrol;
- TYPE = 3, indicating the current trigger type is icount;
- TYPE = 4, indicating the current trigger type is itrigger;
- TYPE = 5, indicating the current trigger type is etrigger。

C920 supports 2 types of triggers:

1. mcontrol trigger, The TYPE field is hardwired to 0x2;
2. itrigger/etrigger/icount configurable registers, and the TYPE field can be configured as 0x3, 0x4, or 0x5; its reset value is 0x5.

DEMODE——Control the write permissions for Debug/Trace Trigger Data Registers 1, 2, and 3(TDATA1/TDATA2/TDATA3)

- When DEMODE is set to 0, Debug mode and M-mode both allow for writing to Debug/Trace Trigger Data Registers;
- When DEMODE is set to 1, Only Debug mode allows for writing to Debug/Trace Trigger Data Registers.

DATA——Control Debug/Trace Trigger Data Register 1

- The specific meaning of DATA field is determined by TYPE.(For the specific infomration, please refer to the chapter 5.2 in *RISC-V debug spec v0.13.2)*.

18.1.8.3 Debug/Trace Trigger Data Register 2 (TDATA2)

DATA——Data of Debug/Trace Trigger Data Register 2

DATA is designed to set the trigger value, and the specific meaning is determined by the TYPE field of TDATA1.

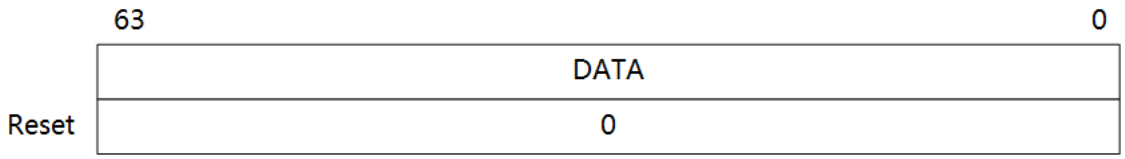


Fig. 18.11: **Debug/Trace Trigger Data Register 2 (TDATA2)**

18.1.8.4 Debug/Trace Trigger Data Register 3 (TDATA3)

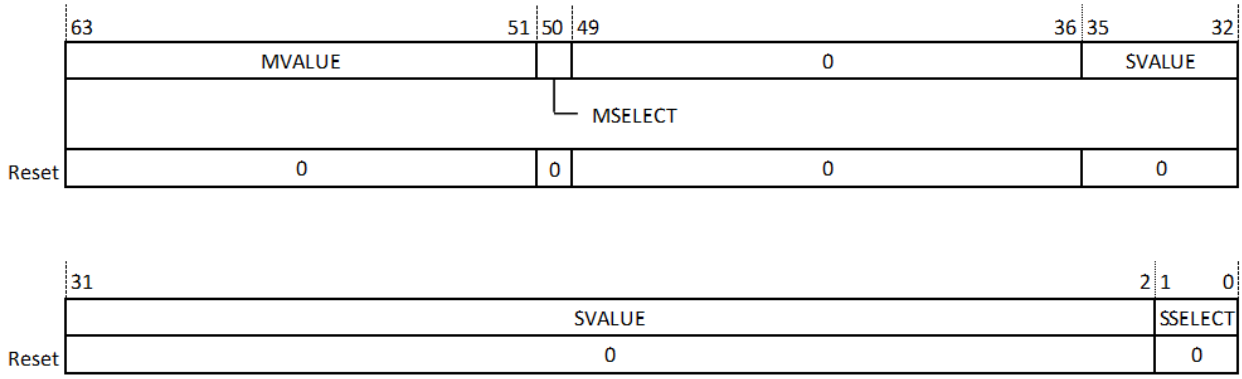


Fig. 18.12: **Debug/Trace Trigger Data Register 3 (TDATA3)**

SSELECT——Control Trigger S-Mode Content Matching

- When SSELECT is set to 0: close trigger S-mode content matching.
- When SSELECT is set to 1: the trigger is capable of matching CPU information, as the S-mode Content Register(SCONTEXT) is equal to the Trigger S-mode Content Matching Data (SVALUE).
- When SSELECT is set to 2: the trigger is capable of matching CPU information, as the value of ASID field in satp register is equal to the trigger S-mode Content Matching data.

SVALUE——Trigger S-Mode Content Matching Data

SVALUE is designed to set the desired value of content in S-mode to be matched.

MSELECT——Trigger M-mode Content Matching Control

- When MSELECT is set to 0: close trigger M-mode content matching.
- When MSELECT is set to 1: the trigger is capable of matching CPU information, as M-mode Content Register (MCONTEXT) is equal to Trigger M-ode Content Matching Data (MVALUE).

MVALUE——Trigger M-mode Content Matching Data

MVALUE is designed to set the desired value of content in M-mode to be matched.

18.1.8.5 Debug/Trace Trigger Information Register (TINFO)

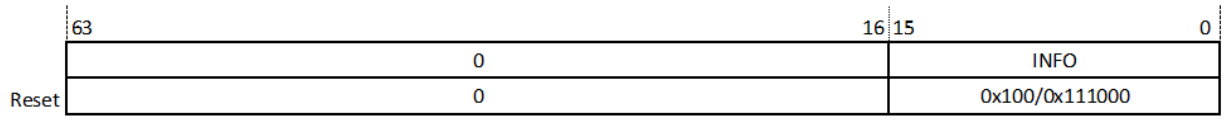


Fig. 18.13: **Debug/Trace Trigger Information Register (TINFO)**

INFO——The Types Supported by the Trigger

When bit[n] is set to 1, it indicates that the TYPE field of the trigger TDATA1 can be configured as n.

C920 supports 2 types of triggers:

1. mcontrol trigger, INFO is hardwired to 0x100, indicating the TYPE field in TDATA1 can only be set as 2;
2. itrigger/etrigger/icount configurable registers, and INFO is hardwired to 0x111000, indicating the TYPE field can be configured as

18.1.8.6 Debug/Trace Trigger CSR (TCONTROL)

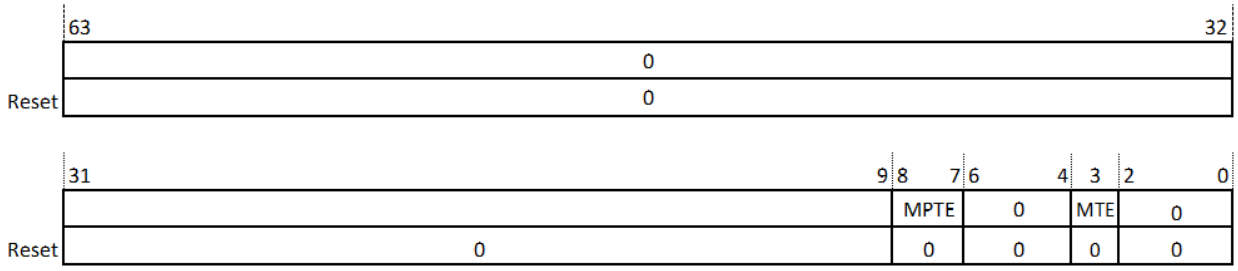


Fig. 18.14: **Debug/Trace Trigger CSR (TCONTROL)**

MTE——M-mode Trigger Enable Control

- When MTE is set to 0: the trigger that generates a breakpoint exception can not be triggered in M-mode;
- When MTE is set to 1: the trigger can be triggered in M-mode.

The hardware sets MTE to 0 when entering a M-mode exception/interrupt handling routine; The hardware resets MTE to the value of MPTE upon returning from the M-mode exception/interrupt handler handling routine.

MPTE——M-mode Trigger Enable Backup

The hardware stores the value of MTE into MPTE, when entering a M-mode exception/interrupt handling routine.

18.1.8.7 M-mode Content Register (MCONTEXT)

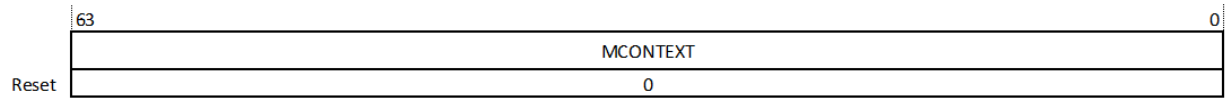


Fig. 18.15: **M-mode Content Register (MCONTEXT)**

MCONTEXT——M-mode Content

In M-mode, software can write to a specific context and control the trigger to be triggered only in specific M-mode contexts.

M-mode software can program a specific context, and it can control triggering to occur only in specific M-mode contexts, by combining the MSELECT with MVALUE in TDATA3.

18.1.9 Debug Mode Register Group/Trace Register Group

18.1.9.1 Debug Mode Control and Status Register (DCSR)

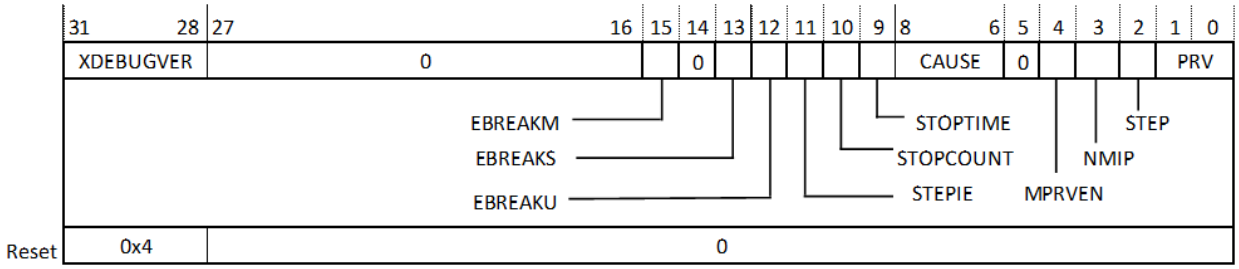


Fig. 18.16: **Debug Mode Control and Status Register (DCSR)**

XDEBUGVER:

- 0: No debug system
- 4: A debugging system supporting riscv debug spec v0.13.2
- 15: A debugging system without supporting riscv debug spec v0.13.2

EBREAKM:

- 0: Executing the ebreak instruction generates a breakpoint exception in M-mode.
- 1: Executing the ebreak instruction enters debug mode in M-mode.

EBREAKS:

- 0: Executing the ebreak instruction generates a breakpoint exception in S-mode.
- 1: Executing the ebreak instruction enters debug mode in S-mode.

EBREAKU:

- 0: Executing the ebreak instruction generates a breakpoint exception in U-mode.
- 1: Executing the ebreak instruction enters debug mode in U-mode.

STEPIE:

- 0: Interrupts are not responded to in single-step debugging.
- 1: Interrupts are responded to in single-step debugging.

STOPCOUNT:

- 0: Performance monitoring counters continue to count normally in debug mode.
- 1: During debug mode, the performance monitoring counters, specifically MCYCLE and MINSTRET, cease to increment their counts.

STOPTIME:

- 0: The processor core’ s private clock counter continues to count normally in debug mode.
- 1: During debug mode, the processor core’ s private clock counter does not count.

CAUSE:

- 1: The reason for entering debug mode is the execution of the ebreak instruction.
- 2: The reason for entering debug mode is the triggering of the trigger.
- 3: The reason for entering debug mode is the synchronous debug request.
- 4: The reason for entering debug mode is the single-step debug request.
- 5: The reason for entering debug mode is the reset debug request.

MPRVEN:

- 0: The MPRV field in the MSTATUS register is ineffective during debug mode.
- 1: The MPRV field in the MSTATUS register is effective, and CPU handles the address translation and protection of memory access instruction, based on the configurations of both MPRV and MPP.

NMIP:

- 0: No NMI Interrupts in CPU.
- 1: NMI Interrupts occur in CPU.

STEP:

- 0: No single-step debug.

- 1: Initiate Single-Step Debug Mode.

PRV:

- The CPU privilege mode is stored into the PRV when entering debug mode; And the CPU sets its privilege mode according to the PRV field upon exiting debug mode.

18.1.9.2 Debug Mode Program Counter (DPC)

DPC[63:0]:

- The hardware writes the address of the next instruction into DPC when entering debug mode. The CPU resumes fetching and executing instructions from the address saved within the DPC Upon exiting debug mode

18.1.9.3 Debug Scratch Register 0 (DSCRATCH0)

DSCRATCH0[63:0]:

- Hardware mechanism for data exchange between the debug system and the processor core.

18.1.9.4 Debug Mode Temporary Data Scratch Register 1 (DSCRATCH1)

DSCRATCH1[63:0]:

- Hardware mechanism for data exchange between the debug system and the processor core.

18.2 Appendix C-2 RISC-V Standard S-mode Control Register

18.2.1 S-mode Exception Configuration Register Group

When exceptions and interrupts are delegated to S-mode responses, exceptions must be configured through the S-mode exception configuration register group, like in M-mode.

18.2.1.1 S-mode Status Register (SSTATUS)

The SSTATUS register stores status and control information of the CPU in S-mode, including the global interrupt enable bit, exception preserve interrupt enable bit, exception preserve privilege mode bit and so on. The SSTATUS register is a partial mapping of the mstatus register.

This register is 64-bit wide and is readable and writable in M-mode and S-mode. Accesses in U-mode will cause an illegal instruction exception.

For detailed information, please refer to *M-Mode Status Register (MSTATUS)* .

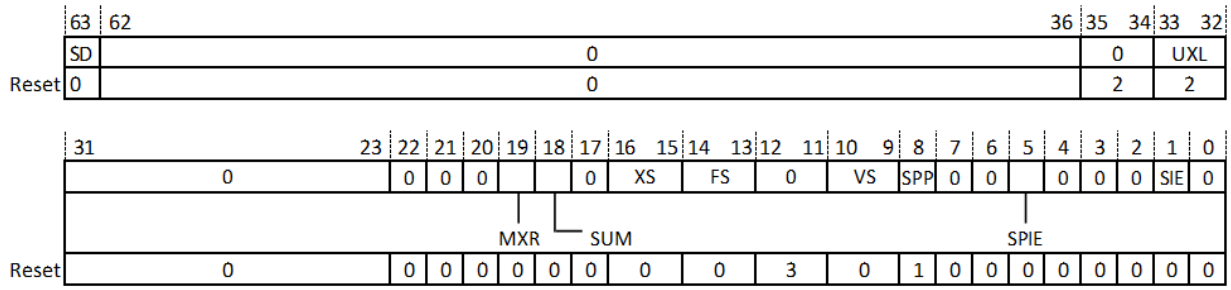


Fig. 18.17: **S-mode Status Register (SSTATUS)**

18.2.1.2 S-mode Interrupt Enable register (SIE)

The SIE register controls the enable and mask of different types of interrupts, and is a partial mapping of the MIE register.

This register is 64-bit wide and readable in S-mode. The write permission in S-mode is determined by the mideleg register of the corresponding bit. Accesses in U-mode will cause an illegal instruction exception.

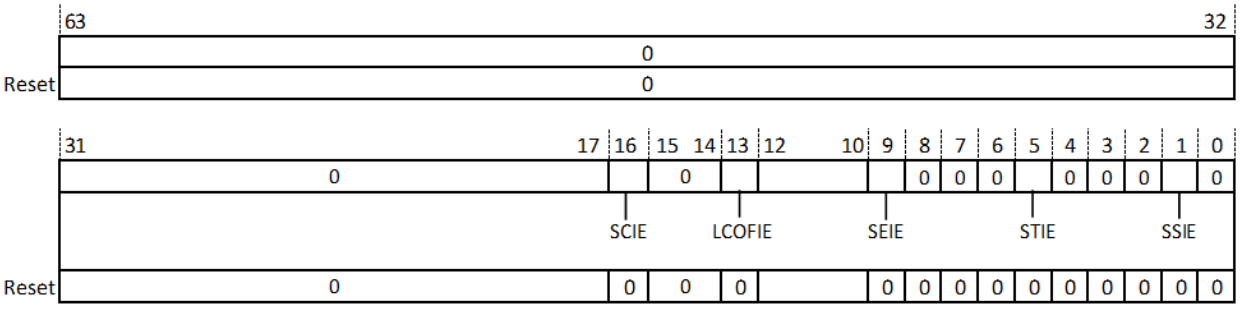


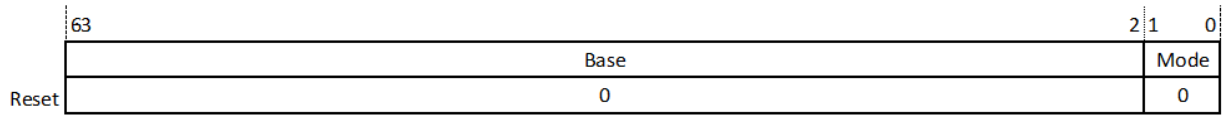
Fig. 18.18: **S-mode Interrupt Enable register (SIE)**

For detailed information, please refer to *M-mode Interrupt Enable Register (MIE)* 。

18.2.1.3 S-mode Trap Vector Base Address Register (STVEC)

STVEC register stores the entry address of the exception service program.

This register is 64-bit wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.



	63 ... 2	1 ... 0
	Base	Mode
Reset	0	0

Fig. 18.19: **S-mode Trap Vector Base Address Register (STVEC)**

For detailed information, please refer to *M-mode Vector Base Address (MTVEC)* .

18.2.1.4 S-mode Counter Enable Register (SCOUNTEREN)

The scounteren register determines whether U-mode counters can be accessed in U-mode.

For detailed information, please refer to *Scounteren Register* .

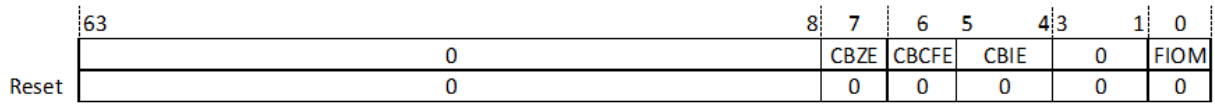
18.2.1.5 S-mode Counter Interrupt Overflow Register (SCOUNTOVF)

SCOUNTOVF register is the extension of sscofpmf.

For detailed information, please refer to *SCOUNTOVF Register* .

18.2.2 S-mode Environment Configuration Register Group

18.2.2.1 S-mode Environment Configuration Register(SENVCFG)



	63 ... 8	7	6	5 ... 4	3 ... 1	0
	0	CBZE	CBCFE	CBIE	0	FIOM
Reset	0	0	0	0	0	0

Fig. 18.20: **S-mode Environment Configuration Register(SENVCFG)**

The SENVCFG register is applied to control features within the execution environment when running at privilege levels below S-mode.

This register is 64-bit wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

CBZE - CBO.ZERO instruction enable bit

When CBZE is set to 0, executing CBO.ZERO instruction in a lower privilege mode will occur an illegal instruction exception;

When CBZE is set to 1, the CBO.ZERO instruction executes normally in less privileged modes.

CBCFE - the enable bit of CBO.CLEAN and CBO.FLUSH instructions

When CBCFE is set to 0, executing CBO.CLEAN and CBO.FLUSH instructions in a lower privilege mode will occur illegal instruction exceptions;

When CBCFE is set to 1, the CBO.CLEAN and CBO.FLUSH instructions execute normally in less privileged modes.

CBIE - CBO.INVAL instruction enable bit

When CBIE is set to 0, executing CBO.INVAL instruction in a lower privilege mode will occur an illegal instruction exception;

When CBIE is set to 1, the CBO.INVAL instruction is executed as the CBO.FLUSH instruction in a lower privilege mode;

When CBIE is set to 2, the reserved value should not be configured;

When CBIE is set to 3, the CBO.INVAL instruction executes normally in less privileged modes.

FIOM - IO fence including memory access

Regardless of the value of the bit, fence synchronization for IO will always include both read and write(RW) synchronization.

Note:

In Xuantie 920, all fence synchronizations for IO inherently include RW synchronization, and hence are not controlled by this particular bit. In terms of the standard RV definition, when senvcfg.FIOM is set to 1, IO synchronization requests from from privilege levels below S-mode need to be synchronized with memory RWs.

18.2.3 S-mode Exception Handling Register Group

18.2.3.1 S-Mode Scratch Register for Exception Temporal Data Backup (SSCRATCH)

The SSCRATCH register is applied to back up temporary data in the exception service program. It is usually used to store the entry pointer value of the local context space in S-mode.

This register is 64-bit wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

18.2.3.2 S-mode Exception Program Counter Register (SEPC)

The SEPC register stores the program counter value (PC value) when the CPU exits from the exception service program. C920 supports 16-bit wide instructions. The value of SEPC is aligned to a 16-bit boundary, with the least significant bit being zero.

This register is 64-bit wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

18.2.3.3 S-mode Exception Cause Register (SCAUSE)

The SCAUSE register stores the vector numbers of events that trigger exceptions, to handle corresponding events in the exception service program.

This register is 64-bit wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

18.2.3.4 S-Mode Interrupt Pending Status Register (SIP)

The SIP register stores information about pending interrupts. When the CPU can not immediately respond to an interrupt, the corresponding bit in the SIP register will be set.

This register is 64-bit wide and readable in S-mode. The write permission is determined by the mideleg register of the corresponding bit. Accesses in U-mode will cause an illegal instruction exception.

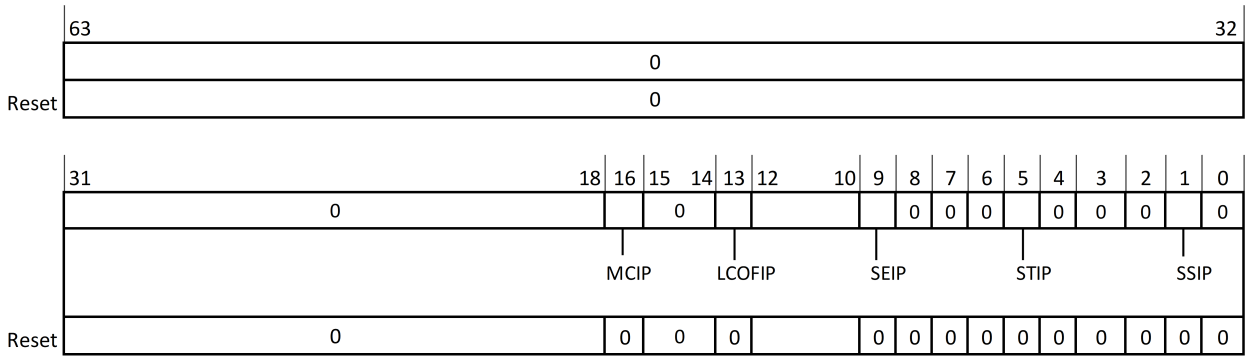


Fig. 18.21: **S-Mode Interrupt Pending Status Register (SIP)**

18.2.4 S-mode Address Protection Register Group

In S-mode, the need arises to access virtual memory space. The S-mode Address Translation and Protection Register (SATP) is applied to control the mode switching of the MMU unit, as well as to set the base address for hardware page table walks and the process number.

18.2.4.1 S-mode Address Translation and Protection Register (SATP)

The SATP register is applied to control the mode switching of the MMU unit, as well as to set the base address for hardware page table walks and the process number.

This register is 64-bit wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For detailed information, please refer to *MMU Address Translation Register (SATP)* .

18.2.5 S-mode Debug Register Group

18.2.5.1 S-mode Content Register Content Register (SCONTEXT)

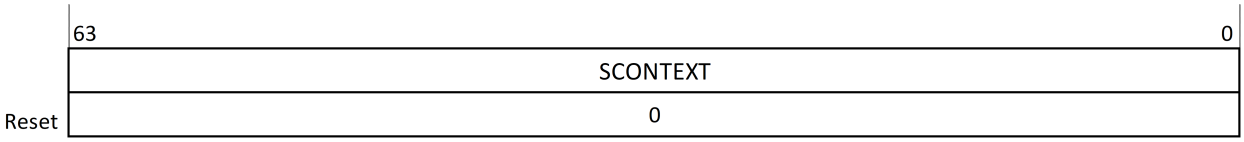


Fig. 18.22: **S-mode Content Register Content Register (SCONTEXT)**

SCONTEXT[63:0]——S-mode Content

In S-mode, software can write to a specific context and control over triggering that only occurs in specific S-mode contexts by combining with SSELECT and SVALUE within the Debug/Trace Trigger Data Register 3 (TDATA3).

18.2.6 S-mode Timer and Counter Register Group

18.2.6.1 S-mode Timer Interrupt Compare Value Register (STIMECMP)

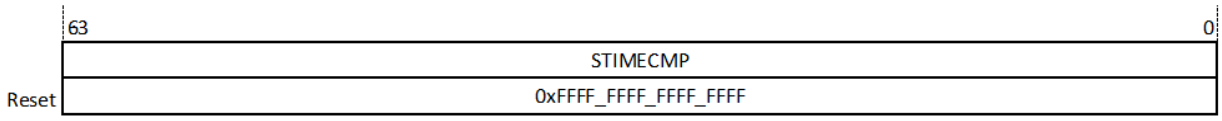


Fig. 18.23: **S-mode Timer Interrupt Compare Value Register (STIMECMP)**

The STIMECMP register is applied to store a timer comparison value. The value in STIMECMP is compared against the current value of the system timer to determine whether a S-mode timer interrupt should be generated. No interrupt is generated when the value in STIMECMP is greater than the system timer' s value; While, an interrupt for the S-mode timer occurs when the STIMECMP value is less than or equal to the system timer' s value and the STCE field within the M-mode Environment Configuration Register (MENVCFG) is set to 1.

Software can clear S-mode timer interrupts caused by STIMECMP by rewriting STIMECMP value.

This register is 64-bit wide. In S-mode, the register is readable and writable when the STCE field in MENVCFG register is set to 1 and the TM field in the M-mode Counter Access Enable Register (MCOUNTEREN) is also set to 1, otherwise it will result in an illegal instruction exception. Access from U-mode will result in an illegal instruction exception.

18.3 Appendix C-3 RISC-V Standard U-mode Control Register

U-mode control registers are mainly divided into floating-point registers, counter registers, and vector control registers by features.

18.3.1 U-mode Floating-point Control Register Group

18.3.1.1 Floating Point Accrued Exception Flags Register (FFLAGS)

The FFLAGS register is the field mapping of accrued exceptions of the Floating-Point Control and Status Register (FCSR). For detailed information, please refer to *Floating-Point Control and Status Register (FCSR)* .

18.3.1.2 Floating-point Dynamic Rounding Mode Register (FRM)

The FRM register is the field mapping of the rounding mode of the FCSR register. For detailed information, please refer to *Floating-Point Control and Status Register (FCSR)* .

18.3.1.3 Floating-Point Control and Status Register (FCSR)

FCSR records floating-point accrued exceptions and the rounding mode cntrol.

This register is 64-bit wide and readable and writable in any privilage mode.

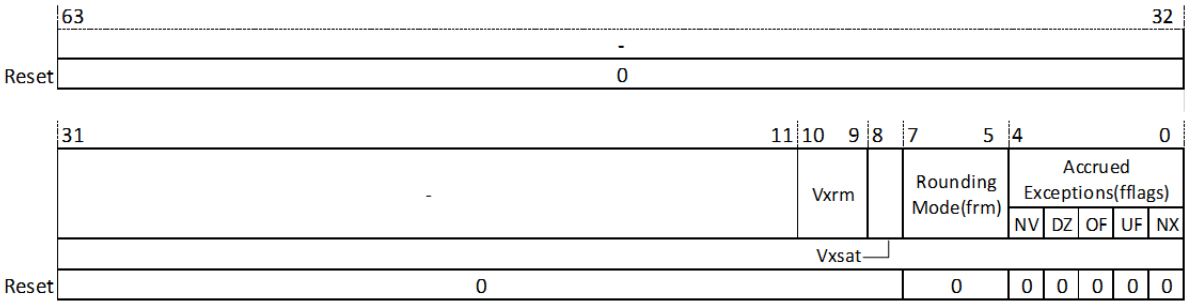


Fig. 18.24: **Floating-Point Control and Status Register (FCSR)**

NX——imprecise exception:

- When NX is set to 0, no imprecise exception occurs.
- When NX is set to 1, imprecise exceptions occur.

UF——underflow exception:

- When UF is set to 0, no underflow exception occurs.
- When UF is set to 1, underflow exceptions occur.

OF——overflow exception:

- When OF is set to 0, no overflow exception occurs.
- When OF is set to 1, overflow exceptions occur.

DZ——division by zero exception:

- When DZ is set to 0, no division by zero exception occurs.
- When DZ is set to 1, division by zero exceptions occur.

NV——invalid operand exception:

- When NV is set to 0, no exception of illegal instruction operands occurs.
- When NV is set to 1, exceptions of illegal instruction operands occur.

RM——rounding mode:

- When RM is set to 0, the RNE rounding mode takes effect, and values are rounded off to the nearest even number.
- When RM is set to 1, the RTZ rounding mode takes effect, and values are rounded off to zero.
- When RM is 2, the RDN rounding mode takes effect, and values are rounded off to negative infinity.
- When RM is set to 3, the RUP rounding mode takes effect, and values are rounded off to positive infinity.
- When RM is set to 4, the RMM rounding mode takes effect, and values are rounded off to the nearest number.

VXSAT——vector overflow flag:

Mapping of the VXSAT flag

VXRM——vector rounding mode bit

Mapping of the VXRM flag

18.3.2 U-mode Timer/Counter Register Group

18.3.2.1 U-Mode Cycle Counter (CYCLE)

The CYCLE stores the cycles executed by the CPU. When the CPU is in the execution state (non-low power state), the CYCLE register increments its count automatically on every execution cycle

The CYCLE is a 64-bit register, and it will be reset to zero.

For detailed information, please refer to *Event Counters* .

18.3.2.2 U-Mode Timer Counter (TIME)

TIME is a read-only mapping of Machine Time (MTIME) register

For detailed information, please refer to *Event Counters* .

18.3.2.3 U-mode Instructions Retired Counter (INSTRET)

INSTRET stores the number of retired instructions of the CPU. The INSTRET increments its count when each instruction retires.

The CYCLE is a 64-bit register, and it will be reset to zero.

For detailed information, please refer to *Event Counters* .

18.3.2.4 U-mode Event Counter (HPMCOUNTERn)

HPMCOUNTERn is the mapping of M-mode Event Counter MHPMCOUNTERn.

For detailed information, please refer to *Event Counters* .

18.3.3 Vector Extension Register Group

18.3.3.1 Vector Start Position Register (VSTART)

The VSTART register specifies the start position of the element when executing vector instructions. The VSTART will be reset to 0 after each vector instruction is executed.

18.3.3.2 Fixed-point Overflow Flag Register (VXSAT)

The VXSAT register specifies whether any fixed-point instruction overflows.

18.3.3.3 Fixed-point Rounding Mode Register (VXRM)

The VXRM register specifies the rounding mode used by fixed-point instructions.

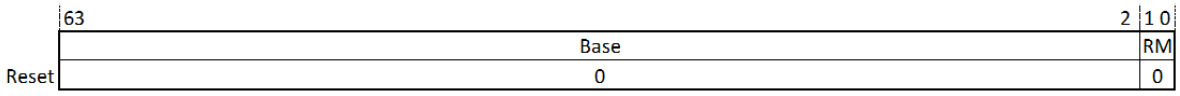


Fig. 18.25: **Fixed-point Rounding Mode Register (VXRM)**

RM——fixed-point rounding mode:

- When RM is set to 0, the RNU rounding mode takes effect, and values are rounded off to a large number.
- When RM is set to 1, the RNE rounding mode takes effect, and values are rounded off to an even number.
- When RM is set to 2, the RDN rounding mode takes effect, and values are rounded off to zero.
- When RM is set to 3, the ROD rounding mode takes effect, and values are rounded off to an odd number.

18.3.3.4 Vector Length Register (VL)

The VL register specifies the range of the destination register to be updated by a vector instruction. The vector instruction updates the elements with a sequence number smaller than the VL register value in the destination register, and clears those with a sequence number greater than the VL register value. Particularly, when vstart >= VL or VL is 0, all elements in the destination register are not updated.

This register is read-only in any mode, but its value can be updated by the vsetvli, vsetvl, and fault-only-first instructions.

18.3.3.5 Vector Control and Status Register (VCSR)

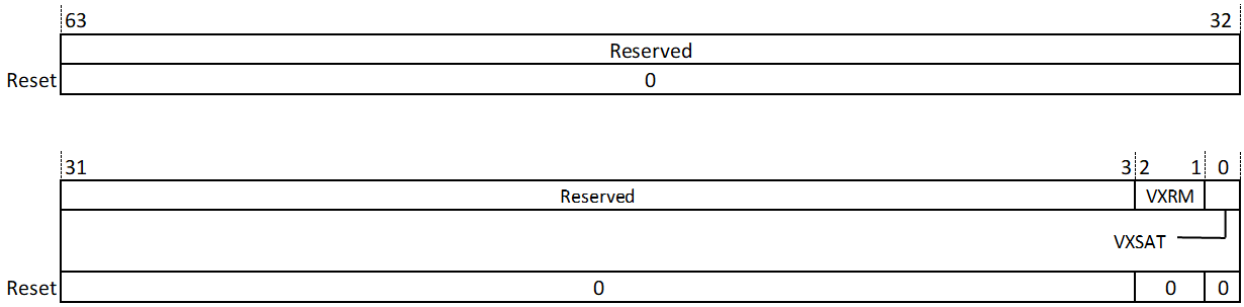


Fig. 18.26: **Vector Control and Status Register (VCSR)**

The VXRM and VXSAT information correspond to the mirrored values of their respective registers.

18.3.3.6 Vector Data Type Register (VTYPE)

The vtype register specifies the data type and elements of the vector registers.

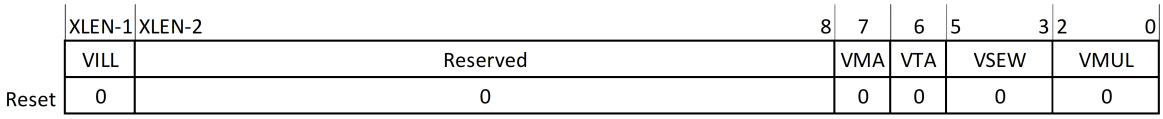


Fig. 18.27: **Vector Data Type Register (VTYPE)**

This register is read-only in any mode, but its value can be updated by the vsetvli and vsetvl instructions.

VILL——illegal operation flag:

When VTYPE register is updated by vsetvli/vsetvl/vsetivli instructions with the value not supported by C920, this flag is set; otherwise, it remains at 0. When this flag is set, executing vector instructions that depend on vtype results in an illegal instruction exception.

Note:

Vector instructions that do not depend on vtype include vset{i}vl{i} instructions and load/store/move instructions related to the entire register group.

VMA——mask element configuration bit

VTA——tail element configuration bit

VTA and VMA modify the behavior of tail elements and masked elements in the destination during vector instruction execution.

VTA modifies the behavior of tail elements in the destination during vector instruction execution.

- VTA=0 indicates that tail elements in the destination will be set to an undisturbed state.
- VTA=1 signifies that tail elements will be set to an agnostic state.

VMA modifies the behavior of masked elements in the destination during vector instruction execution.

- VMA=0 indicates that masked elements in the destination will be set to an undisturbed state.
- VMA=1 indicates that masked elements in the destination will be set to an agnostic state.

VSEW——vector element width setting bit:

VSEW determines the Vector Element Width (SEW), and the SEW supported by C920 is illustrated in Table 18.2 .

Table 18.2: **Vector Element Width (VSEW)**

VSEW[2:0]			Element Bit Width
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64

When VSEW is set to other values, the C920 generates an illegal instruction exception upon executing vector instructions.

VLMUL——vector register grouping setting bit:

Multiple vector registers can consti a vector register group. Vector instructions operate on all vector registers within such a group. VLMUL determines the number of vector registers in the vector register group, as shown in Table 18.3 .

Table 18.3: **Number of Registers in the Vector Register Group**

VLMUL[2:0]			LMUL
1	0	1	1/8
1	1	0	1/4
1	1	1	1/2
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	8

18.3.3.7 Vector Width (Unit: Byte) Register (VLENB)

The VLENB register specifies the CPU’ s vector width in bytes.

The vector width of C910 is 128 bits (VLEN=128). Therefore, VLENB = 128/8 = 16.

18.4 Appendix C-4 C920 Extended M-mode Control Register

18.4.1 M-mode Mode Processor Control and Status Extension register group

18.4.1.1 M-Mode Extension Status Register (MXSTATUS)

The MXSTATUS stores the current privilage mode of CPU and C920 extension enable bit.

This register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

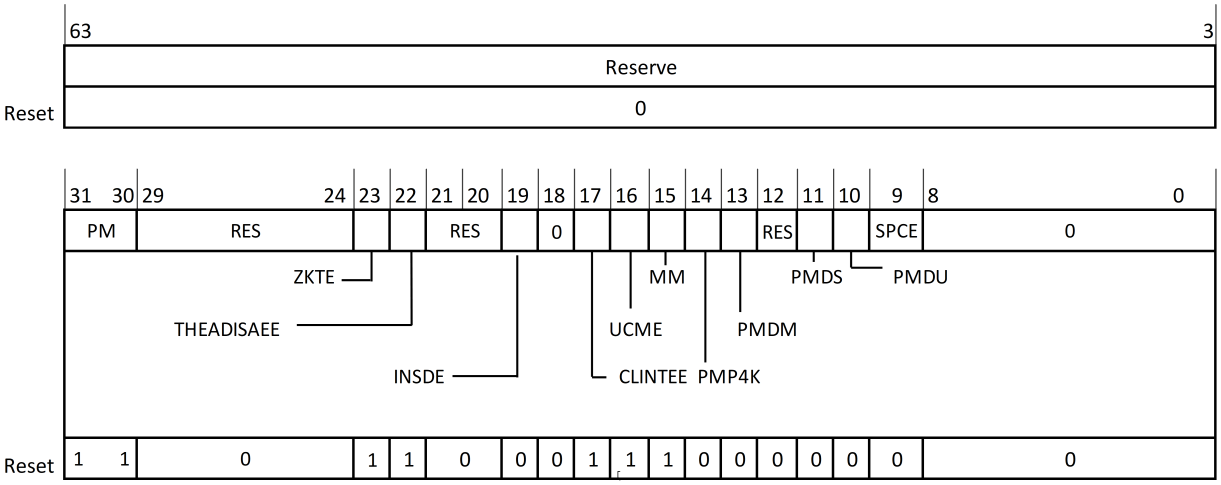


Fig. 18.28: **M-Mode Extension Status Register (MXSTATUS)**

SPCE——S-mode Cache Partition Control Enable:

When SPCE is set to 0, S-mode does not support configuration of L2 Cache partion.

When SPCE is set to 1, S-mode supports configuration of L2 Cache partion.

PMDU——U-mode Performance Monitor Count Enable Bit:

When PMDU is set to 0, performance counter counting is enabled in U-mode.

When PMDU is set to 1, performance counter counting is disabled in U-mode.

PMDS——S-mode Performance Monitor Counter Enable Bit:

When PMDS is set to 0, performance counter counting is enabled in S-mode.

When PMDS is set to 1, performance counter counting is disabled in S-mode.

PMDM——M-mode Performance Monitor Counter Enable Bit:

When PMDM is set to 0, performance counter counting is enabled in M-mode.

When PMDMis set to 1, performance counter counting is disabled in M-mode.

PMP4K——PMP Minimum Granularity Control Bit:

Currently, C920 supports a PMP minimum granularity of 4K only, and is not affected by that bit.

MM——Non-aligned Access Enable Bit:

When MM is set to 0, non-aligned access is not supported, and the access will generate an unaligned exception.

When MM is set to 1, aligned access is supported, and the hardware handles non-aligned accesses. (The default value in C920 is 1.)

UCME——U Mode Cache Extension Instructions:

When UCME is set to 0, execution of extended cache operation instructions is not allowed in U-mode and will result in an illegal instruction exception.

When UCME is set to 1, execution of extended cache operation instructions is supported in U-mode

CLINTEE——Clint Timer/Software Interrupt S-mode Extensions Enable Bit:

When CLINTEE is set to 0, S-mode software interrupts and timer interrupts initiated by CLINT will not be responded to.

When CLINTEE is set to 1, the response to S-mode software interrupts and timer interrupts initiated by CLINT is supported.

INSDE——Disable Icache snoop Dcache:

When INSDE is set to 0, it will snoop the Dcache after an Icache miss.

When INSDE is set to 1, it will not snoop the Dcache after an Icache miss.

THEADISAEE——Enable Extension Instruction Set

When THEADISAEE is set to 0, an illegal instruction exception occurs upon the application of the C920 extended instruction set.

When THEADISAEE is set to 1, C920 extended instruction set is supported.

ZKTE——zkt enable bit:

When ZKTE is set to 0, the execution latency of zkt-related instructions is variable.

When ZKTE is set to 1, the execution latency of zkt-related instructions is fixed.

The value is fixed to 1 in C920.

PM——Privileged mode the CPU is in:

When PM is set to 2’ b00, CPU is in U-mode.

When PM is set to 2’ b01, CPU is in S-mode.

When PM is set to 2’ b11, CPU is in M-mode (switch into M-mode after a reset).

18.4.1.2 M-mode Hardware Configuration Register (MHCR)

The MHCR register is applied to configure the CPU in terms of its performance and functionality.

This register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

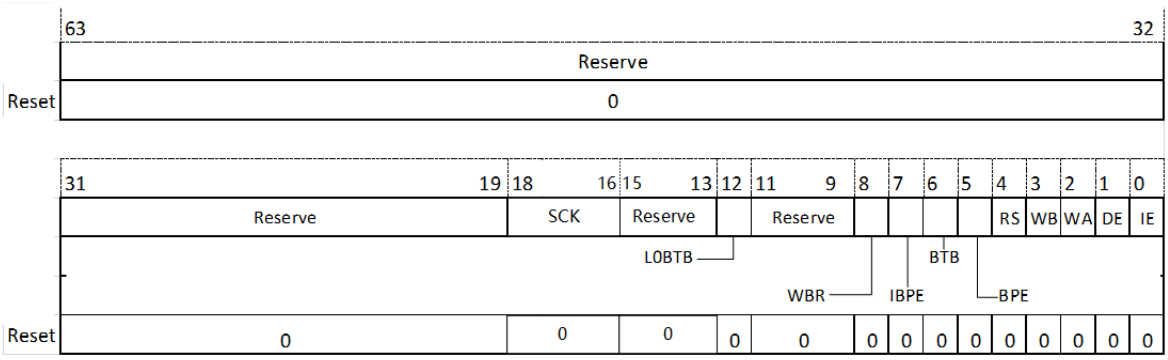


Fig. 18.29: **M-mode Hardware Configuration Register (MHCR)**

IE——Icache enable bit:

When IE is set to 0, Icache is disabled.

When IE is set to 1, Icache is enbaled.

DE——Dcache enable bit:

When DE is set to 0, Dcache is disabled.

When DE is set to 1, Dcache is enbaled.

WA——Cache Write Allocate Bit:

When WA is set to 0, Data cache is in write non-allocate mode.

When WA is set to 1, Data cache is in write allocate mode.

WB——Cache Write-Back Bit:

When WB is set to 0, Data cache is in Write-Through mode.

When WB is set to 1, Data cache is in Write-Back mode.

C920 only supports Write-Back mode, with the fixed WB value 1.

RS——Return Address Stack Bit:

When RS is set to 0, return-address stack is disabled.

When RS is set to 1, return-address stack is enabled.

BPE——Branch Prediction Enable Bit

When BPE is set to 0, branch prediction is disabled.

When BPE is set to 1, branch prediction is enabled.

BTB——Branch Target Prediction Enable Bit:

When BTB is set to 0, branch target prediction is disabled.

When BTB is set to 1, branch target prediction is enabled.

IBPE——Indirect-jump Prediction Enable Bit:

When IBPE is set to 0, indirect-jump prediction is disabled.

When IBPE is set to 1, indirect-jump prediction is enabled.

WBR——Write Burst Transfer Enable Bit

When WBR is set to 0, write burst transfer enable bit is not supported.

When WBR is set to 1, write burst transfer enable bit is supported.

This default value is 1 in C920 and not adjustable.

L0BTB——First-level Branch Target Prediction Enable Bit:

When L0BTB is set to 0, the first-level branch target prediction is disabled.

When L0BTB is set to 1, the first-level branch target prediction is enabled.

SCK——System to Processor Clock Ratio

The SCK field in C920 is fixed at 0 and does not indicate clock ratio information.

18.4.1.3 M-mode Hardware Operation Register (MCOR)

The MCOR register is applied to operate on the cache and branch prediction units.

This register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

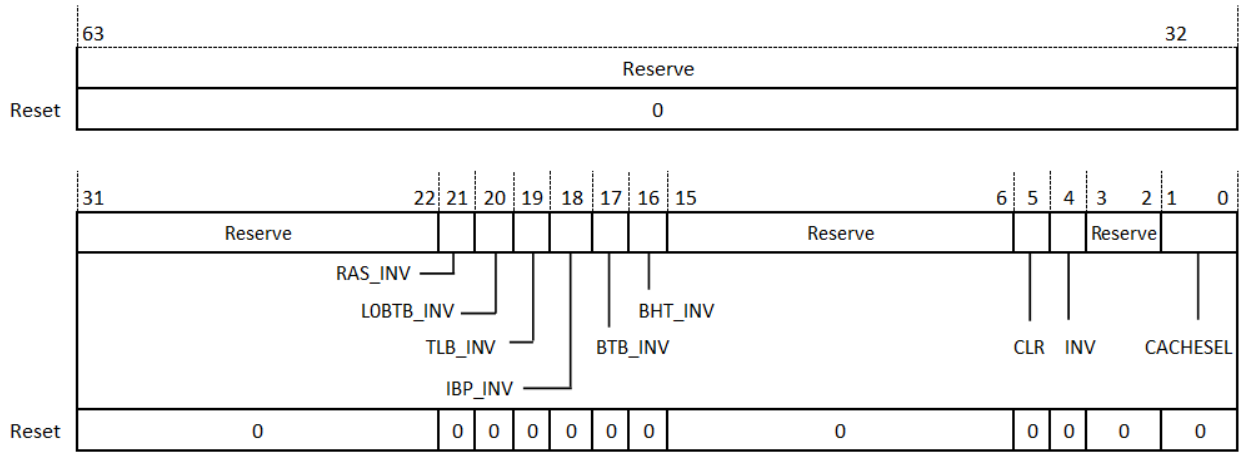


Fig. 18.30: **M-mode Hardware Operation Register (MCOR)**

CACHESEL——Cache Select Bit:

When CACHE_SEL is set to 2’ b01, select instruction cache.

When CACHE_SEL is set to 2’ b10, select data cache.

When CACHE_SEL is set to 2’ b11, select instruction and data cache.

INV——Cache Invalidate Bit:

When INV is set to 0, cache will not be invalidated.

When INV is set to 1, cache will be invalidated.

CLR——Cache Dirty Entry Clear Bit:

When CLR is set to 0, cache entries marked dirty will not be written back to off-chip memory.

When CLR is set to 1, cache entries marked dirty will be written back to off-chip memory.

BHT_INV——BHT Invalidate Bit:

When BHT_INV is set to 0, branch history table entries will not be invalidated.

When BHT_INV is set to 1, branch history table entries will be invalidated.

BTB_INV——BTB Invalidate Bit:

When BTB_INV is set to 0, data in the branch target buffer will not be invalidated.

When BTB_INV is set to 1, data in the branch target buffer will be invalidated.

IBP_INV——IBP Invalidate Bit:

When IBP_INV is set to 0, data for indirect jump branch predictions will not be invalidated.

When IBP_INV is set to 1, data for indirect jump branch predictions will be invalidated.

For all the invalidation and cleaar operations mentioned above, the corresponding bits are set high during the write process and cleared back to 0 upon completion of the operation.

TLB_INV——inv tlb Set Bit:

When IBP_INV is set to 0, data in TLB will not be invalid.

When IBP_INV is set to 1, data in TLB will be invalid.

L0BTB_INV——L0_BTB Invalid Set Bit:

When L0_BTB is set to 0, data of L0_BTB will not be invalidated.

When L0_BTB is set to 1, data of L0_BTB will be invalidated.

RAS_INV——RAS Invalid Set Bit:

When RAS_INV is set to 0, data of RAS will not be invalidated.

When RAS_INV is set to 1, data of RAS will be invalidated.

18.4.1.4 M-mode L2Cache Control Register (MCCR2)

The MCCR2 register is applied to configure access latency for individual memory modules within a shared L2 cache, the enable/disable status of the L2 cache itself, instruction prefetch capabilities, Translation Lookaside Buffer (TLB) prefetch enabling, and Error Correction Code (ECC) checking enablement.

This register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

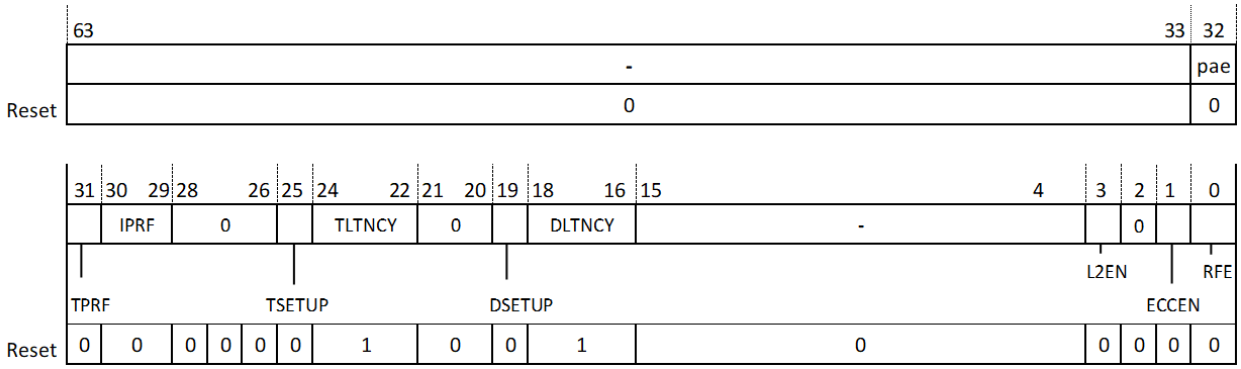


Fig. 18.31: **M-mode L2Cache Control Register (MCCR2)**

RFE——Data Access Read Allocate Enable Bit:

When RFE is set to 0 and there is a data access miss in the L2 Cache, instead of refilling the L2 Cache, the data is directly refilled into the D Cache, which means that there exists an exclusive relationship between the L1 DCache and the L2 Cache.

When RFE is set to 1, and there is a data access miss in the L2 Cache, instead of refilling the L2 Cache, the data is directly refilled into the D Cache, which means that there exists an inclusive relationship between the L1 DCache and the L2 Cache. (The fixed value in C920 is 1)

ECCEN——ECC Enable Bit:

When ECCEN is set to 0, L2Cache ECC is disabled.

When ECCEN is set to 1, L2Cache ECC is enabled.

L2EN——L2Cache Enable Bit:

When L2EN is set to 0, L2Cache is disabled.

When L2EN is set to 1, L2Cache is enabled.(The fixed value in C920 is 1)

DLTNCY——L2Cache DATA RAM Access Cycle Configuration Bit:

When DLTNCY is set to 0, DATA RAM access cycle is 1.

When DLTNCY is set to 1, DATA RAM access cycle is 2.

When DLTNCY is set to 2, DATA RAM access cycle is 3.

When DLTNCY is set to 3, DATA RAM access cycle is 4.

When DLTNCY is set to 4, DATA RAM access cycle is 5.

When DLTNCY is set to 5, DATA RAM access cycle is 6.

When DLTNCY is set to 6, DATA RAM access cycle is 7.

When DLTNCY is set to 7, DATA RAM access cycle is 8.

DSETUP——L2Cache DATA RAM Setup Configuration Bit:

When DSETUP is set to 0, DATA RAM does not require an additional setup cycle;

When DSETUP is set to 1, DATA RAM requires an additional setup cycle.

This bit is read-only.

TLTNCY——L2Cache TAG RAM Access Cycle Configuration Bit:

When TLTNCY is set to 0, TAG RAM access cycle is 1;

When TLTNCY is set to 1, TAG RAM access cycle is 2;

When TLTNCY is set to 2, TAG RAM access cycle is 3;

When TLTNCY is set to 3, TAG RAM access cycle is 4;

When TLTNCY is set to 4, TAG RAM access cycle is 5.

TSETUP——L2 CACHE TAG RAM Setup Configuration Bit:

When TSETUP is set to 0, TAG RAM does not require an additional setup cycle;

When TSETUP is set to 1, TAG RAM requires an additional setup cycle.

This bit is read-only.

IPRF——L2Cache Instruction Prefetch Capability:

The number of cache lines to prefetch upon a fetch request miss for instructions in the L2Cache:

When IPRF is set to 0, L2Cache instruction prefetch is disabled;

When IPRF is set to 1, prefetch one cache line;

When IPRF is set to 2, prefetch two cache line;

When IPRF is set to 3, prefetch three cache line.

TPRF——L2Cache TLB Prefetch Enable:

When TPRF is set to 0, L2 Cache TLB prefetch is disabled;

When TPRF is set to 1, L2 Cache TLB prefetch is enabled.

pae——Partition Access Enable Bit:

When pae is set to 0, L2 cache does not support partition access.

When pae is set to 1, L2 cache does support partition access.

18.4.1.5 M-mode L2 Cache ECC Control Register(MCER2)

MCER2 register is applied to configure L2 Cache ECC (Error Correction Code). L2 cache supports configurable ECC, which supports 1bit error correction and 2 bit error detection.When a 2 bit error is detected, the hardware automatically sets the ERR_VLD bit within the MCER2 register, along with information about the location of the error, for software inquiry. Software can write 0 to clear the ERR_VLD; however, it cannot set to 1.

This MCER2 register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

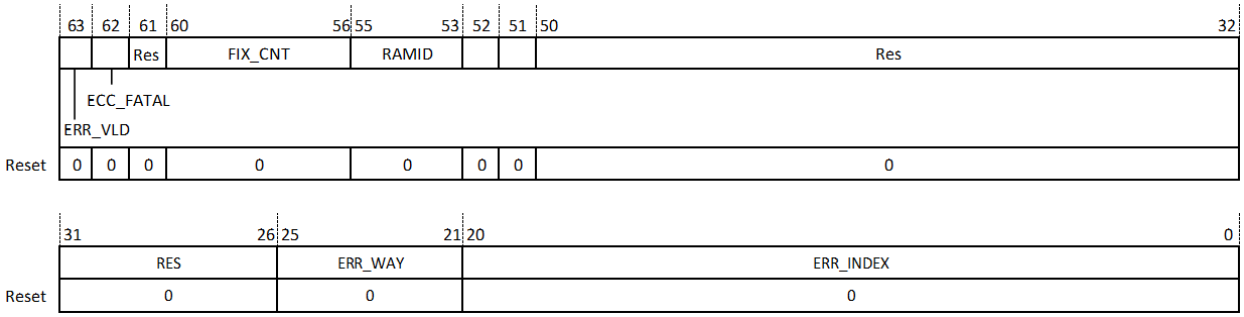


Fig. 18.32: **M-mode L2 Cache ECC Control Register (MCER2)**

ERR_VLD——L2 CACHE Parity/ECC Error Indicator Bit:

When ERR_VLD is set to 0, no ECC Error, Parity Error, or Bus Write Error occurs in L2 CACHE.

When ERR_VLD is set to 1, an ECC Error, Parity Error, or a Bus Write Error occurs in L2 CACHE.

Software can clear this error bit within an exception service but can not set it high.

ECC_FATAL——L2 CACHE Fatal Error Bit:

When ECC_FATAL is set to 0, no 2 bit ECC error occurs in L2 CACHE.

When ECC_FATAL is set to 1, 2 bit ECC errors occur in L2 CACHE.

FIX_CNT[4:0]——The number of ECC Errors that have been corrected:

It records the number of ECC Errors that have been corrected.

RAMID[2:0]——The SRAM ID number associated with the ECC Error:

It records the SRAM ID number associated with the ECC Error.

ID=0: L2 CACHE TAG RAM

ID=1: L2 CACHE DATA RAM

ID=2: L2 CACHE DIRTY RAM

ID=3: SNOOP FILTER CORE0;

ID=4: SNOOP FILTER CORE1;

ID=5: SNOOP FILTER CORE2;

ID=6: SNOOP FILTER CORE3.

ERR_WAY——L2 CACHE Parity/ECC Error Bit Position Information

It records the first occurrence of a 2-bit parity/ECC error location in the L2 CACHE.

ERR_INDEX——L2 CACHE Error Correction Index Information:

It records the index location of the first occurrence of a 2-bit parity/ECC error in the L2 CACHE.

18.4.1.6 M-mode Implicit Operation Register (MHINT)

The MHINT is applied to multiple functional switches within the cache System.

This MHINT register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

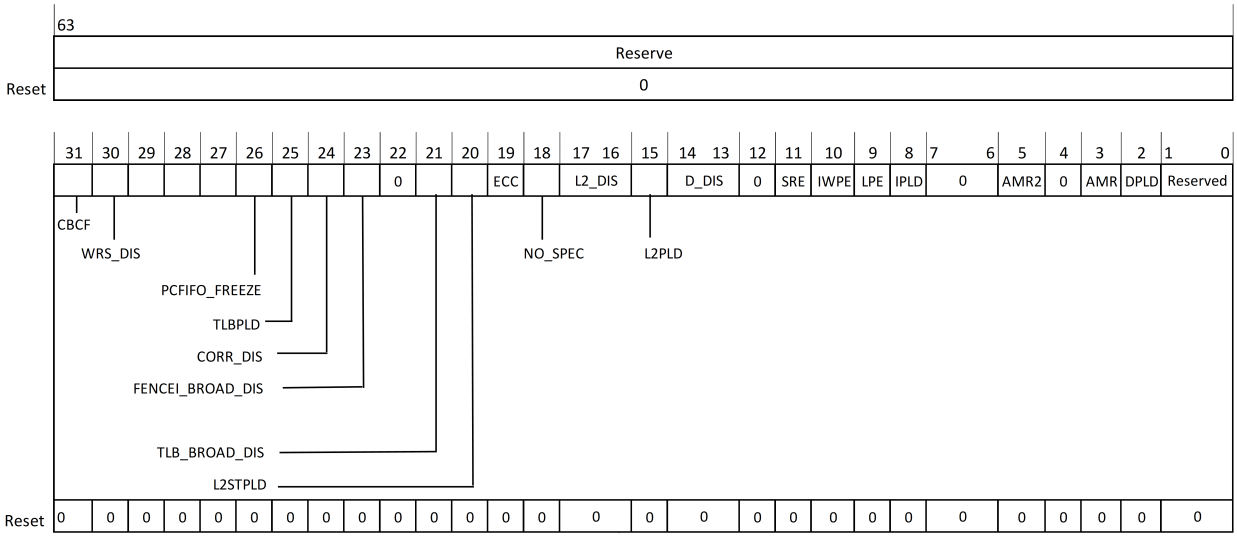


Fig. 18.33: **M-mode Implicit Operation Register (MHINT)**

DPLD——DCACHE Prefetch Enable Bit:

When DPLD is set to 0, DCACHE prefetch is disabled;

When DPLD is set to 1, DCACHE prefetch is enabled.

AMR——L1 Cache Write Allocate Auto-Tuning Enable Bit:

When AMR is set to 0, the write allocation is determined by the WA (Write Allocate) attribute of the accessed address' s page.

When AMR is set to 1, subsequent contiguous address store operations are not written into the L1 Cache in the event of consecutive cache line store operations.

AMR2——L2Cache Write Allocate Auto-Tuning Enable Bit

When AMR2 is set to 0, the write allocation is determined by the WA attribute of the accessed page.

When AMR2 is set to 1, subsequent contiguous address store operations are not written into the L2 Cache in the event of consecutive cache line store operations.

IPLD——ICACHE Prefetch Enable Bit:

When IPLD is set to 0, ICACHE prefetch is disabled.

When IPLD is set to 1, ICACHE prefetch is enabled.

LPE——Loop Acceleration Enable Bit:

When LPE is set to 0, loop acceleration is disabled.

When LPE is set to 1, loop acceleration is enabled.

IWPE——ICACHE Route Prediction Enable Bit:

When IWPE is set to 0, ICACHE route prediction is disabled;

When IWPE is set to 1, ICACHE route prediction is enabled.

SRE——Single Retire Mode Bit:

When SRE is set to 0, single retire mode is disabled;

When SRE is set to 1, single retire mode is enabled.

D_DIS——DCACHE Number of Prefetched Cache Lines:

When DPLD is set to 0, prefetch two cache lines.

When DPLD is set to 1, prefetch four cache lines.

When DPLD is set to 2, prefetch 8 cache lines.

When DPLD is set to 3, prefetch 16 cache lines.

The default value is 0.

L2PLD——L2CACHE Prefetch Enable Bit:

When L2PLD is set to 0, L2CACHE prefetch is disabled;

When L2PLD is set to 1, L2CACHE prefetch is enabled.

L2_DIS——L2CACHE Number of Prefetched Cache Lines:

When L2_DIS is set to 0, prefetch 8 cache lines;

When L2_DIS is set to 1, prefetch 16 cache lines;

When L2_DIS is set to 2, prefetch 32 cache lines;

When L2_DIS is set to 3, prefetch 64 cache lines;

L2 Cache prefetch is performed on the basis of L1 Cache.

NO_SPEC——SPEC FAIL Prefetch Enable Bit:

When NO_SPEC is set to 0, spec fail prefetch is disabled;

When NO_SPEC is set to 1, spec fail prefetch is enabled.

ECC——L1 CACHE Checksum Enable Bit:

When ECC is set to 0, L1Cache Checksum is disabled;

When ECC is set to 1, L1Cache Checksum is enabled.

L2STPLD——L2 Cache Store Prefetch Enable Bit:

When L2STPLD is set to 0, L2 CACHE store prefetch is disabled;

When L2STPLD is set to 1 , L2 CACHE store prefetch is enabled.

TLB_BROAD_DIS——TLB fence Broadcast Invalidate Bit:

When TLB_BROAD_DIS is set to 0 , sfence.vma instruction is broadcast to other cores;

When TLB_BROAD_DIS is set to 1 , sfence.vma instruction will not be broadcast.

The bit does not exist in the single core.

FENCEI_BROAD_DIS——fence.i Broadcast Invalidate Bit:

When FENCEI_BROAD_DIS is set to 0 , fence.i instruction is broadcast to other cores;

When FENCEI_BROAD_DIS is set to 1 , fence.i instruction will not be broadcast.

The bit does not exist in the single core.

CORR_DIS——RAR Out-of-order Correction in RAR

When CORR_DIS is set to 0, it indicates a more conservative approach of out-of-order RAR detection, which enables error correction processing once out-of-order RAR occurs

When CORR_DIS is set to 1, it indicates a more performance-optimized approach of out-of-order RAR detection, which enables error correction processing only when data errors arise in out-of-orde RAR;

TLBPLD——TLB Prefetch Enable Bit:

When TLBPLD is set to 0, TLB prefetch is disabled;

When TLBPLD is set to 1, TLB prefetch is enabled;

PCFIFO_FREEZE——the PCFIFO of DEBUG Records the Jump Target PC Enable Bit

When PCFIFO_FREEZE is set to 0, PCFIFO records jump target PC normally;

When PCFIFO_FREEZE is set to 1, PCFIFO disables the recording of jump target PC;

WRS_DIS——Control the Normal Execution of WRS Instruction or Execute as No Operation instruction (NOP).

When WRS_DIS is set to 0, WRS instruction is normally executed.

When WRS_DIS is set to 1, WRS instruction is executed as NOP.

CBCF——Invalidation is added to the Data Cache Clear instruction

When CBCF is set to 0, Data Cache Clear instruction is normally executed, only including a Clear operation.

When CBCF is set to 1, Data Cache Clear instruction includes an Invalid operation (Flush) besides the Clear operation.

Instructions influenced by this bit include:cbo.clean、dcache.call、dcache.cpa、dcache.cpal1、dcache.cva、dcache.cval1、dcache.csw.

18.4.1.7 M-mode Reset Register (MRMR)

Note:

mrmr register has been removed from C920 (the version above R1S4), and the related features has been deleted. But the software can still access this register. And the result is that reads return zero, and writes are ineffective without causing any exceptions to be raised.

mrmr register is applied to enable the release of reset for each C920 core during multi-core initialization. Each processing core shares a common MRMR Register, which enables the processor core to programmatically release other cores from their reset condition by setting MRMR.

This MRMR register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

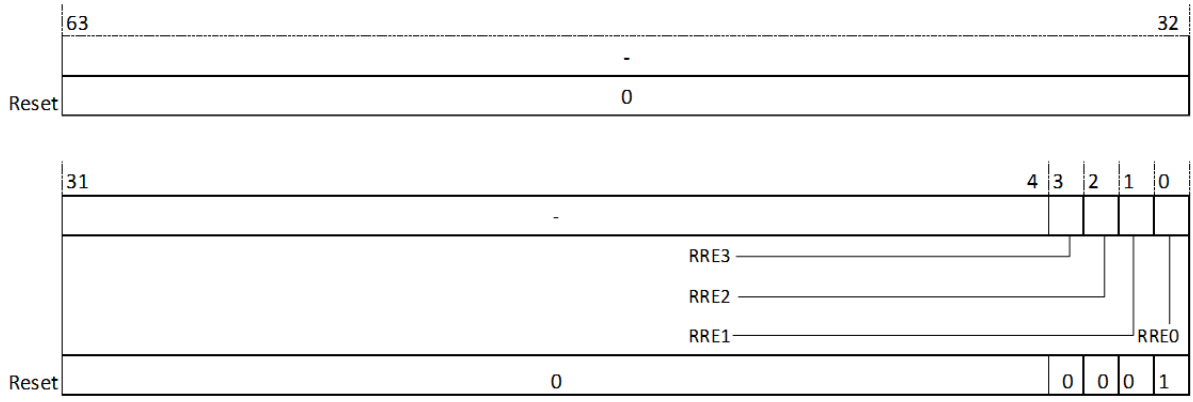


Fig. 18.34: **M-mode Reset Register (MRMR)**

RRE3/2/1/0——Reset Release Enable Bit:

It controls the reset release enable bit of each core.

When RREx is set to 0, and the corresponding C920 Core is in reset state.

When RREx is set to 1, and the corresponding C920 Core is in reset release.

18.4.1.8 M-mode Reset Vector Base Address Register (MRVBR)

MRVBR register is applied to store reset exception vector address. Each C920 core contains its own independent MRVBR register.

This MRMR register is 64-bit wide and read-only in M-mode. The access in non-machine mode will result in an illegal instruction exception.

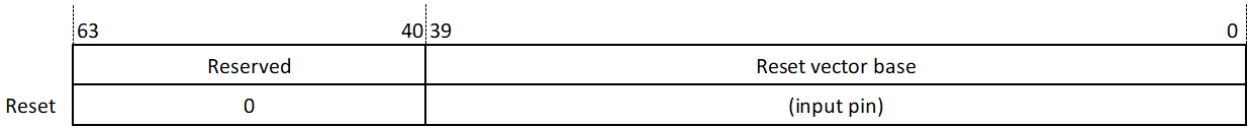


Fig. 18.35: **M-mode Reset Vector Base Address Register (MRVBR)**

Reset vector base——Reset Base Address:

It controls the reset base address of cores.

18.4.1.9 M-mode L1Cache ECC Register (MCER)

MCER register is applied to configure the L1 Cache ECC. The L1 cache supports configurable ECC, which enables single-bit error correction and double-bit error correction.

When the two or more bit errors are detected, the hardware automatically sets the ERR_FATAL bit within the MCER register, along with the information about the error location, for software inquiry. Software can clear the ERR_FATAL bit by writing 0 to it, but can not set it to 1.

This MCER register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

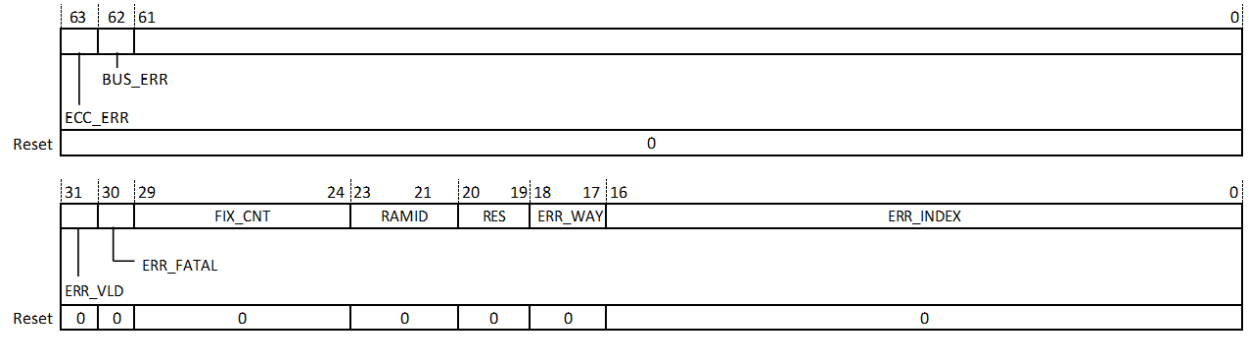


Fig. 18.36: **L1Cache ECC Register (MCER)**

ECC_ERR——ECC Information Valid Bit, Valid When ERR_VLD is Set High

When ECC_err is set to 0, L1 CACHE ECC infomation is invaild;

When ECC_err is set to 1, L1 CACHE ECC is vaild.

BUS_ERR——Vaild Bit of Bus Error:

It is vaild when ERR_VLD is set high.

When bus_err is set to 0, no ld bus exception occurs;

When bus_err is set to 1, ld bus exceptions occur.

ERR_VLD——ECC Information Vaild Bit:

When ERR_VLD is set to 0, no bus error or ECC err occurs;

When ERR_VLD is set to 1, bus errors or ECC errs occur.

ERR_FATAL——L1CACHE Checksum or Parity Error Bit:

When ERR_FATALis set to 0, hardware can correct ERR;

When ERR_FATAL is set to 1, the bit can only be cleared by software as 2 or more bit errors occur.

FIX_CNT——Corrected Error Count Bit:

This bit records the number of corrected errors, and it is automatically cleared when the ECC_VLD is reset.

RAMID——RAM with ECC FATAL Error:

When RAMIDis set to 0, L1 ICACHE TAG RAM checksum error occrus;

When RAMID is set to 1, L1 ICACHE DATA RAM checksum error occrus;

When RAMID is set to 2, L1 DCACHE TAG RAM checksum error occrus;

When RAMID is set to 3, L1 DCACHE DATA RAM checksum error occrus;

When RAMID is set to 4, JTLB TAG RAMchecksum error occrus;

When RAMID is set to 5, JTLB DATA RAMchecksum error occrus.

ERR_WAY——The Address of the first ECC FATAL Error Occurs:

It records the address of the first ECC FATAL ERROR occurred, and subsequent errors will not update this information until the first error is processed by software.

ERR_INDEX——The Index Position of the First ECC FATAL Error OccurreS:

It records the index position of the first ECC FATAL ERROR occurred, and subsequent errors will not update this information until the first error is processed by software.

18.4.1.10 M-mode Counter Write Enable Register (MCOUNTERWEN)

The MCOUNTERWEN is applied to authorize whether S-mode can write to the S-mode event counter. For more details, please refer to *Mcounterwen Register* .

18.4.2 M-mode Extended Register Group 2

18.4.2.1 M-mode Performance Monitor Control Register (MHPMCR)

MHPMCR is Xuantie self-extended register. For detailed information, please refer to *MHPMCR Register* .

18.4.2.2 M-mode Performance Monitor Start Trigger Register (MHPMSR)

MHPMSR is Xuantie self-extended register. For detailed information, please refer to *Start Trigger Register* .

18.4.2.3 M-Mode Performance Monitor End Trigger Register (MHPMER)

MHPMER is Xuantie self-extended register. For detailed information, please refer to *End Trigger Register*

18.4.2.4 M-Mode Profiling/Sampling Enable Register (MSMPR)

MSMPR controls whether a core can process listen requests. Each core has an individual configuration to handle such requests independently. The top-level coherent bus root relies on the listening status of each core to determine and control the transmission of listen requests accordingly. This register is readable and writable in M-mode.

This MRMR register is 64-bit wide, with only bit 0 defined; all other bits are Reserved

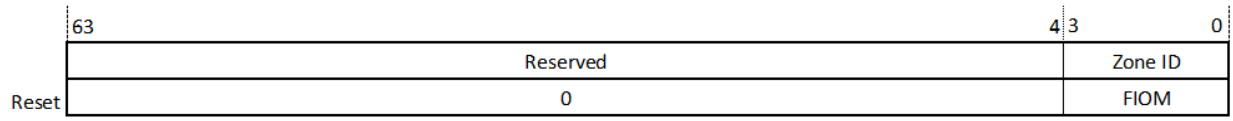
bit 0: SMPEN-Core Listen Enable Bit

- When SMPEN is set to 1' b0, the core is unable to process listen requests, the top-level logic masks sending listen requests to that particular core. (Reset Value)
- When the SMPEN bit is set to 1' b1, the core is enabled to handle listen requests, and the top-level logic will send listen requests to that core. Before powering down a core, it is essential to set the corresponding SMPEN bit for that core to 0, in order to disable its listening functionality. Upon power-up of the core, software must ensure that the SMPEN bit is set to 1 before enabling the D-Cache and MMU. During normal operation of the core, including when it is in WFI mode, the SMPEN must be maintained at 1' b1; otherwise, the results may be unpredictable.

18.4.2.5 Processor ZONE ID Register (MZONEID)

ZoneID

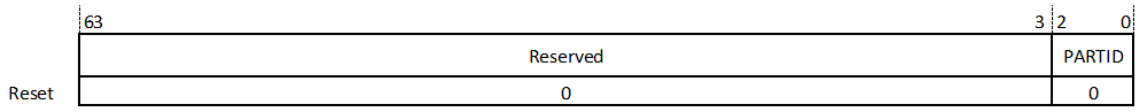
The specified field supports Read, Modify, and Write (MRW) access permissions. The processor can configure the current Zone ID number. Different ZoneIDs correspond to different permissions.



	63 … 4	3 … 0
	Reserved	Zone ID
Reset	0	FIOM

Fig. 18.37: **Processor ZONE ID Register (MZONEID)**

18.4.2.6 Processor Last-Level Cache partition ID Register (ML2PID)



	63 … 3	2 … 0
	Reserved	PARTID
Reset	0	0

Fig. 18.38: **Processor Last-Level Cache partition ID Register (ML2PID)**

PARTID——Processor last-Level cache partition access ID

The specified field supports MRW access permissions. Within the last level cache, the available cache way group for the processor is determined by matching PARTID.

18.4.2.7 Processor L2 Cache Partition Access Configuration Register (ML2WP)

65 … 56	55 … 48	47 … 40	39 … 32	31 … 24	23 … 16	15 … 8	7 … 0
group7	group6	group5	group4	group3	group2	group1	group0
ff	ff	ff	ff	ff	ff	ff	ff
RW	RW	RW	RW	RW	RW	RW	RW

Fig. 18.39: **Processor L2 Cache Partition Access Configuration Register (ML2WP)**

group0[7:0]——group 0 accessible ID

- It includes 8 IDs. And and when the corresponding bit for the processor' s ML2PID number is set to 1, it indicates that this processor has access to the cache way of group0.

group1[7:0]——group 1 accessible ID

- It includes 8 IDs. And and when the corresponding bit for the processor' s ML2PID number is set to 1, it indicates that this processor has access to the cache way of group1.

group2[7:0]——group 2 accessible ID

- It includes 8 IDs. And and when the corresponding bit for the processor' s ML2PID number is set to 1, it indicates that this processor has access to the cache way of group2.

group3[7:0]——group 3 accessible ID

- It includes 8 IDs. And and when the corresponding bit for the processor' s ML2PID number is set to 1, it indicates that this processor has access to the cache way of group3.

group4[7:0]——group 4 accessible ID

- It includes 8 IDs. And and when the corresponding bit for the processor' s ML2PID number is set to 1, it indicates that this processor has access to the cache way of group4.

group5[7:0]——group 5 accessible ID

- It includes 8 IDs. And and when the corresponding bit for the processor' s ML2PID number is set to 1, it indicates that this processor has access to the cache way of group5.

group6[7:0]——group 6 accessible ID

- It includes 8 IDs. And and when the corresponding bit for the processor' s ML2PID number is set to 1, it indicates that this processor has access to the cache way of group6.

group7[7:0]——group 7 accessible ID

- It includes 8 IDs. And and when the corresponding bit for the processor' s ML2PID number is set to 1, it indicates that this processor has access to the cache way of group7.

18.4.2.8 M-mode L1 Cache ECC Single Bit Error Physical Address Register (MSBEPA)

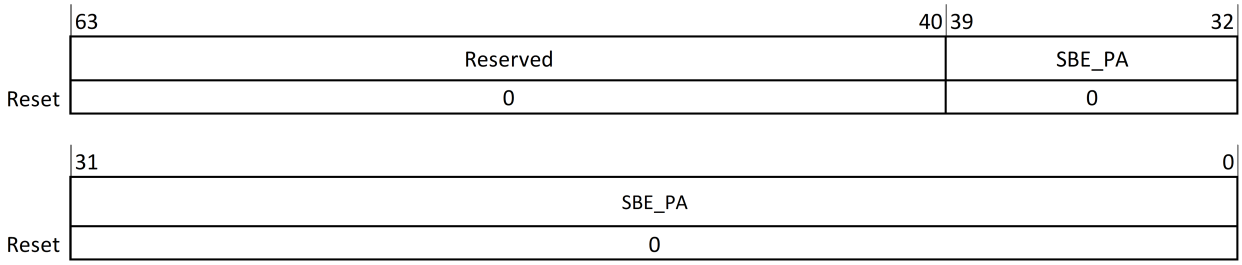


Fig. 18.40: **M-mode L1 Cache ECC Single-bit Error Physical Address Register (MSBEPA)**

SBE_PA - L1 Cache ECC error address

The SBE error physical address is recorded, when DCACHE Single Bit Error (SBE) reports an L1 Cache ECC interrupt.

18.4.2.9 M-mode L2 Cache ECC Single-bit Error Physical Address Register (MSBEPA2)

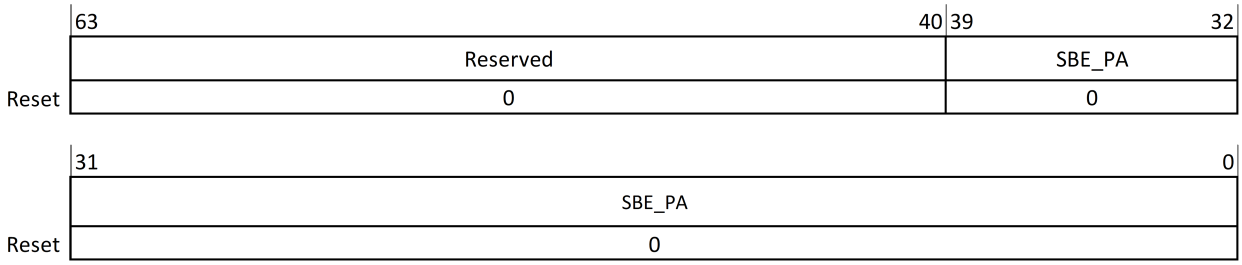


Fig. 18.41: **M-mode L2 Cache ECC Single-bit Error Physical Address Register (MSBEPA2)**

SBE_PA - L2 Cache ECC error address

The SBE error physical address is recorded, when L2 Cache SBE reports an L2 Cache ECC interrupt.

18.4.3 M-mode Cache Access Extension Register Group

M-mode cache access extension registers are designed to directly read L1 and L2 cache, facilitating debugging operations on cache。

18.4.3.1 M-mode Cache Instruction Register (MCINS)

MCINS is applied to is applied to enable a read request to L1 or L2 cache.

This register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

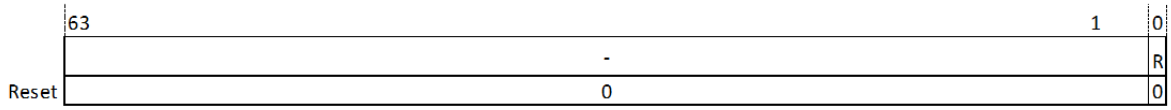


Fig. 18.42: **M-mode Cache Instruction Register (MCINS)**

R-Cache read access:

- When R is set to 0, no read request is enabled.
- When R is set to 0, read request is enabled.

18.4.3.2 M-mode Cache Access Index Register (MCINDEX)

MCINDEX is designed to configure the cache position information of read requests.

This register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

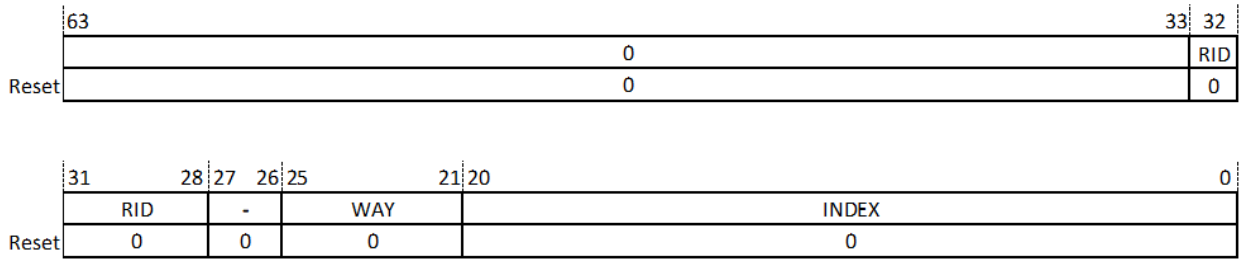


Fig. 18.43: **M-mode Cache Access Index Register (MCINDEX)**

RID-RAM flag:

Indicate the accessed RAM information.

- When RID is set to 0, it indicates ICACHE TAG RAM is accessed.
- When RID is set to 1, it indicates ICACHE DATA RAM is accessed.
- When RID is set to 2, it indicates DCACHE ST TAG RAM is accessed.
- When RID is set to 3, it indicates DCACHE DATA RAM is accessed.
- When RID is set to 4, it indicates L2CACHE TAG RAM is accessed.
- When RID is set to 5, it indicates L2CACHE DATA RAM is accessed.
- When RID is set to 6, it indicates ICACHE TAG ECC RAM is accessed.
- When RID is set to 7, it indicates ICACHE DATA ECC RAM is accessed.
- When RID is set to 8, it indicates DCACHE ST TAG ECC RAM is accessed.
- When RID is set to 9, it indicates DCACHE DATA ECC RAM is accessed.
- When RID is set to 10, it indicates L2CACHE TAG ECC RAM is accessed.
- When RID is set to 11, it indicates L2CACHE DATA ECC RAM is accessed.
- When RID is set to 12, it indicates DCACHE LD TAG RAM is accessed.
- When RID is set to 13, it indicates DCACHE LD TAG ECC RAM is accessed.
- When RID is set to 19, it indicates ICACHE PREDECODE RAM is accessed.
- When RID is set to 20, it indicates SNOOP FILTER RAM is accessed.
- When RID is set to 21, it indicates SNOOP FILTER ECC RAM is accessed.
- When RID is set to 22, it indicates TLB TAG RAM is accessed.
- When RID is set to 23, it indicates TLB DATA RAM is accessed.
- When RID is set to 24, it indicates TLB TAG RAM ECC RAM is accessed.
- When RID is set to 25, it indicates TLB DATA RAM ECC RAM is accessed.

WAY-Cache:

Indicate the RAM access position.

INDEX-Cache:

Indicate the index position of RAM access position.

18.4.3.3 M-mode Cache Data Register (MCDATA0/1)

MCDATA0/1 is designed to record data read from the L1 or L2 cache.

This register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

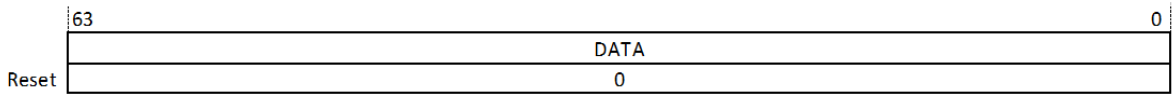


Fig. 18.44: **M-mode Cache Access Data Register (MCDATA)**

Table 18.4: **The Corresponding Relationship of MCDATA and RAM Type**

RAM Type	CDATA
ICACHE TAG	CDATA0[39:12]: TAG CDATA0[0]:VALID
ICACHE DATA	CDATA0~CDATA1: 128 bit DATA
DCACHE ST TAG	CDATA0[39:14]: 26 bit tag CDATA0[13:12] : cindex[13:12] CDATA0[3:0]: {page share, dirty, share, vld}
DCACHE DATA	CDATA0~CDATA1: 128bit DATA
L2CACHE TAG	CDATA0[40]: PAGE_SHARE CDATA0[39:12]: TAG+INDEX CDATA0[7:4]: CP CDATA0[3:0]: {valid, share, dirty, pend}
L2CACHE DATA	CDATA0~CDATA1: 128bit DATA
ICACHE TAG ECC	CDATA0[0]: ECC
ICACHE DATA ECC	CDATA0[3:0]:ECC
DCACHE ST TAG ECC	CDATA0[7:0]: 7bit st tag&dirty ecc info
DCACHE DATA ECC	CDATA0[27:0]: 4 bank * 7bit ecc info
L2CACHE TAG ECC	CDATA0[11:5]: tag ecc CDATA0[4:0]: dirty ecc
L2CACHE DATA ECC	CDATA0[63:0]

Continued on next page

Table 18.4 – continued from previous page

RAM Type	CDATA
DCACHE LD TAG	CDATA0[38:13]:26 bit tag CDATA0[12:11]:cindex[13:12] CDATA0[0]:VALID
DCACHE LD TAG ECC	CDATA0[6:0]:7 bit ecc info(1bit parity + 6 bit ham code)
ICACHE PREDECODE	CDATA0[31:0]:PREDECD
SNOOP FILTER	CDATA0[39:12]:tag+index of this way CDATA0[5:4]:rrpv of this way CDATA0[3:0]:valid
SNOOP FILTER ECC	CDATA0[11:5]:tag ecc of this way CDATA0[3:0]:info ecc of this way
TLB TAG	SV39 CDATA0[27:0]:VPN CDATA0[31:28]:Page Size CDATA0[47:32]:ASID CDATA0[48]:G SV48 CDATA0[35:0]:VPN CDATA0[39:36]:Page Size CDATA0[55:40]:ASID CDATA0[56]:G
TLB DATA	CDATA0[27:0]:PPN CDATA0[40:28]:FLAG Enable and Read Parity:CDATA0[42:41] Parity

Continued on next page

Table 18.4 – continued from previous page

RAM Type	CDATA
TLB TAG RAM ECC	SV39 CDATA0[27:0]:VPN CDATA0[31:28]:Page Size CDATA0[47:32]:ASID CDATA0[48]:G CDATA0[52:49]:4’ b0 Enable and Read Parity:CDATA0[54:53] Parity SV48 CDATA0[35:0]:VPN CDATA0[39:36]:Page Size CDATA0[55:40]:ASID CDATA0[56]:G CDATA0[60:57]:4’ b0 Enable and Read Parity:CDATA0[62:61]:Parity
TLB DATA RAM ECC	CDATA0[27:0] PPN CDATA0[40:28] FLAG Enable and Read Parity:CDATA0[42:41]:Parity

Note: MCINDEX[20:19] = 2’ b00 indicates jTLB;

MCINDEX[20:19] = 2’ b01 indicates iuTLB;

MCINDEX[20:19] = 2’ b10 indicates duTLB。

18.4.3.4 M-mode L1Cache Hardware Error Injection Register (MEICR)

MEICR is applied to injecting ECC errors into the L1 cache.

This register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

INJ_EN-ECC error Injection Enable Bit:

When INJ_EN is set to 1, L1cache ECC error injection is enabled;

When INJ_EN is set to 0, L1cache ECC error injection is disabled.

FATAL_INJ-ECC ERROR Injection Select Bit:

When FATAL_INJ is set to 1, 2-bit error is injected;

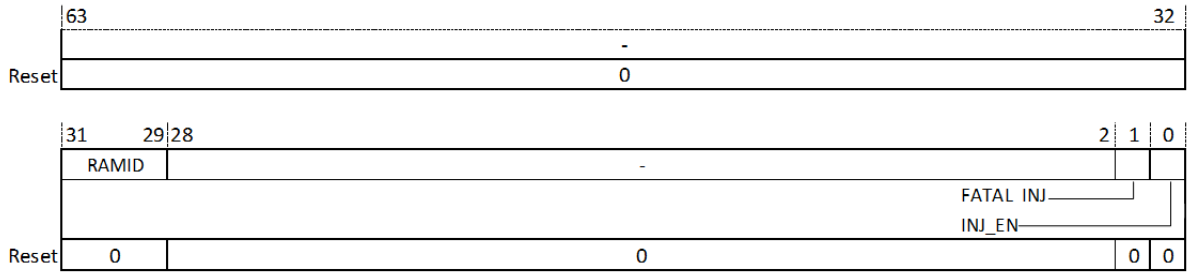


Fig. 18.45: **M-mode L1Cache Hardware Error Injection Register (MEICR)**

When FATAL_INJ is set to 0, 1-bit error is injected.

RAMID-ECC RAM index:

When RAMID is set to 0, ICACHE TAG RAM is injected;

When RAMID is set to 1, ICACHE DATA RAM is injected;

When RAMID is set to 2, DCACHE TAG RAM is injected;

When RAMID is set to 3, DCACHE DATA RAMis injected;

When RAMID is set to 4, JTLB TAG RAM is injected;

When RAMID is set to 5, JTLB DATA RAM is injected。

18.4.3.5 M-mode L2Cache Hardware Error Injection Register (MEICR2)

MEICR2 is applied to injecting ECC errors into the L2 cache.

This register is 64-bit wide and readable and writable in M-mode. The access in non-machine mode will result in an illegal instruction exception.

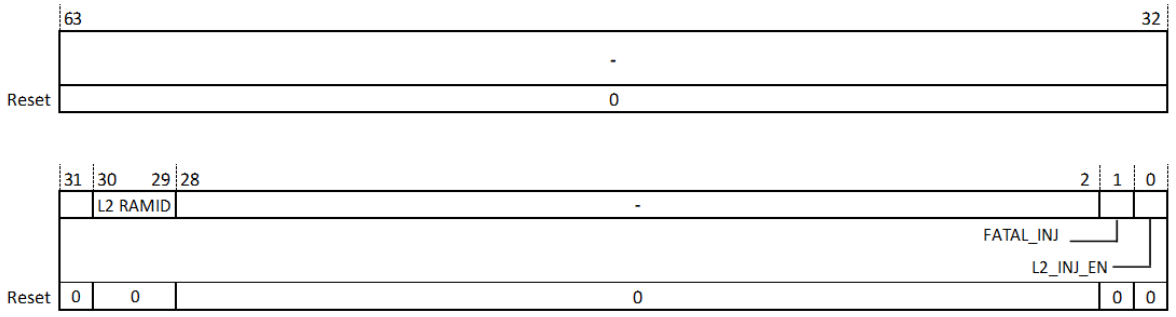


Fig. 18.46: **M-mode L2Cache Hardware Error Injection Register (MEICR2)**

L2_INJ_EN-L2 ECC ERROR Injection Enable Bit:

When L2_INJ_EN is set to 1, L2Cache ECC error injection is enabled;

When L2_INJ_EN is set to 0, L2Cache ECC error injection is disabled.

FATAL_INJ-ECC ERROR Injection Select Bit:

When FATAL_INJ is set to 1, 2-bit error is injected;

When FATAL_INJ is set to 0, 1-bit error is injected.

L2_RAMID-ECC RAM Index:

When RAMID is set to 0, L2 CACHE TAG RAM is injected;

When RAMID is set to 1, L2 CACHE DATA RAM is injected;

When RAMID is set to 2, L2 CACHE DIRTY RAM is injected.

18.4.3.6 L1 LD BUS ERR Address Register (MBEADDR)

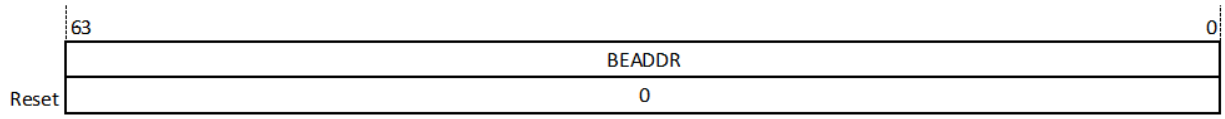


Fig. 18.47: **L1 LD BUS ERR Address Register (MBEADDR)**

BEADDR - Processor Bus Error Address Register

- When there is a bus error in processor, this register stores the physical address of the bus error.

18.4.3.7 Cache Permission Control Register (MCPER)

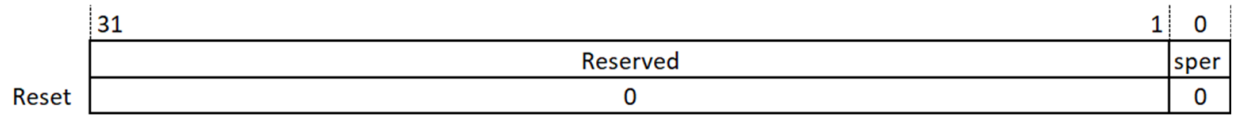


Fig. 18.48: **Cache Permission Control Register (MCPER)**

When SPER is set to 0, the cache extension instructions in the following Table 18.5 can only be executed in M-mode, but can not be executed in S-mode.

When SPER is set to 1, the cache extension instructions in the following table can be executed in both M-mode and S-mode.

Table 18.5: **Cache Extension Instruction**

Instructions	Description
DCACHE.CALL	Write-back data cache write-back instruction
DCACHE.CIALL	Write-back and invalidate DCACHE instruction
DCACHE.IALL	Invalidate all DCACHE instructions
DCACHE.CISW	Invalidate and write-back DCACHE specific way/set instruction
DCACHE.ISW	Invalidate DCACHE specific way/set instruction
ICACHE.IALL	Invalidate all ICACHE instructions
ICACHE.IALLS	Invalidate and brocast all ICACHE instructions

18.4.4 M-mode Processor ID Register Group

18.4.4.1 M-mode Processor ID Register (MCPUID)

MCPUID stores the processor ID, and the reset value is determined by the corresponding product.

18.4.4.2 On-Chip Bus Base Address Register (MAPBADDR)

This register reflects the base address of on-chip registers (CLINT, PLIC) for the processor. The value of this register is determined by the port pad_cpu_apb_base.

18.4.4.3 On-Chip System Interconnect Registers Base Address (MAPBADDR2)

In non-ACE configurations, this register is entirely set to zero. In ACE configurations, the value of this register is determined by the port pad_cpu_apb_base2.

18.4.5 Debug Extension Register Group

18.4.5.1 Xuantie Debug Cause Register (MHALTCAUSE)

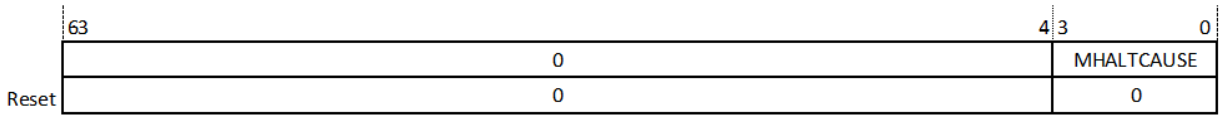


Fig. 18.49: **Xuantie Debug Cause Register (MHALTCAUSE)**

MHALTCAUSE: indicates the cause for entering in debug mode

- When MHALTCAUSE is 1: it indicates that the execution of the ebreak instruction is the cause for entering in debug mode.
- When MHALTCAUSE is 2: it indicates that the triggering of registers is the cause for entering in debug mode.

- When MHALTCAUSE is 3: it indicates that the synchronous debug request is the cause for entering in debug mode.
- When MHALTCAUSE is 4: it indicates that the single-step request is the cause for entering in debug mode.
- When MHALTCAUSE is 5: it indicates that the reset debug request is the cause for entering in debug mode.
- When MHALTCAUSE is 8: it indicates that the asynchronous debug request is the cause for entering in debug mode.

18.4.5.2 Xuantie Debug Information Register (MDBGINFO)

MDBGINFO[63:0]: to record the debug information of the processor core during asynchronous debugging.

18.4.5.3 Xuantie Branch Target Address Record Register (MPCFIFO)

MPCFIFO[63:0]: records the target addresses of branch/jump instructions.

18.4.5.4 Xuantie Debug Information Register 2 (MDBGINFO2)

MDBGINFO2[63:0]: to record CIU and L2-Cache debug during asynchronous debug request.

18.5 Appendix C-5 C920 Extended S-mode Control Registers

This appendix provides a detailed explanation of the C920 Extended S-mode Control Registers.

18.5.1 S-mode Processor Control and Status Extension Registers Group

18.5.1.1 S-mode Extension Status Register Group (SXSTATUS)

SXSTATUS is the mapping of M-mode extension status register (MXSTATUS). For detailed informtion, please refer to *M-Mode Extension Status Register (MXSTATUS)* .

This register is 64-bit wide and readable in S-mode, and only MM, PMDS, and PMDU bits are writeable. The access in U-mode will result in an illegal instruction exception.

18.5.1.2 S-mode Hardware Control Register (SHCR)

SHCR is the mapping of M-mode Hardware Control Register (MHCR). For detailed information, please refer to *M-mode Hardware Configuration Register (MHCR)* .

This register is 64-bit wide and readable in S-mode. The access in U-mode will result in an illegal instruction exception.

18.5.1.3 S-mode L2Cache ECC Register (SCER2)

SCER2 is the mapping of M-mode L2Cache ECC (MCER2). For detailed information, please refer to *M-mode L2 Cache ECC Control Register(MCER2)* .

This register is 64-bit wide and readable in S-mode. The access in U-mode will result in an illegal instruction exception.

18.5.1.4 S-mode L1Cache ECC Register (SCER)

SCER is the mapping of L1 Cache ECC Register (MCER). For detailed information, please refer to *M-mode L1Cache ECC Register (MCER)* .

This register is 64-bit wide and readable in S-mode. The access in U-mode will result in an illegal instruction exception.

18.5.1.5 S-mode Count Inhibit Register (SHPMINHIBIT)

SHPMINHIBIT is the mapping of M-Mode Count Inhibit Register (mcountinhibit). When the mhpmcr.sce is set high, S-mode can control performance monitoring counters by the shpminhibit bit.

In M-mode, shpminhibit register is readable and writeable, regardless of the state of the mcounterwen register. If mcounterwen.bit[n] is set high in S-mode, shpminhibit.bit[n] is readable and writeable; otherwise, any write operation to shpminhibit.bit[n] will be ineffective, and a read operation would return a value of 0.

18.5.1.6 S-mode Performance Monitoring Control Register (SHPMCR)

SHPMCR, Xuantie self-extending register, is the read-write mapping of M-mode Performance Monitoring Control Register (mhpmcr), excluding SCE. When the mhpmcr.sce bit is set high, s-mode can control performance monitoring through this register. In S32 context, only the TS is mapped to BIT[31]. For detailed information, please refer to *SHPMCR Register* .

18.5.1.7 S-mode Performance Monitoring Start Trigger Register (SHPMSR)

SHPMSR is the mapping of M-mode Performance Monitoring Start Trigger Register. When the MHPMCR.sce bit is set high, S-mode can control the starting address of triggered events through this register. For detailed information, please refer to *Trigger Register* .

18.5.1.8 S-mode Performance Monitoring End Trigger Register (SHPMER)

SHPMER is the mapping of M-mode Performance Monitoring End Trigger Register (MHPMER). When MHPMCR.sce bit is set hign, s-mode can control the end address of triggered events through this register. For detailed information, please refer to *Trigger Register* .

18.5.1.9 S-mode Level-2 Cache Partition ID Register (SL2PID)

When the mxstatus.bit[SPCE] is set to 1, S-mode can configure the L2 cache partition access IDs through SL2PID, otherwise S-mode read/write access is illegal.

18.5.1.10 S-mode L2 Cache Partition Access Configure Register (SL2WP)

When mxstatus.bit[SPCE] is set to 1, S-mode can configure the L2 cache partition access by SL2WP, otherwise read/write access is illegal.

18.5.1.11 S-mode L1 LD BUS ERR Address Register (SBEADDR)

The fields definition and feature of this register are the same as that of the MBEADDR register

18.5.1.12 S-mode L1 Cache ECC Single-bit Error Physcal Address Register (SSBEPA)

SSBEPA is the mapping of M-mode L1 Cache ECC Single-bit Error Physcal Address Register (MSBEPA). For detailed information, please refer to *M-mode L1 Cache ECC Single Bit Error Physical Address Register (MSBEPA)*.

This register is 64-bit wide, readable and writeable in S-mode. The access in U-mode will result in an illegal instruction exception.

18.5.1.13 S-mode L2 Cache ECC Single-bit Error Physcal Address Register (SSBEPA2)

SSBEPA2 is the maping of L2 Cache ECC Single-bit Error Physcal Address Register (MSBEPA2). For detailed information, please refer to *M-mode L2 Cache ECC Single-bit Error Physical Address Register (MSBEPA2)*.

This register is 64-bit wide, readable and writeable in S-mode. The access in U-mode will result in an illegal instruction exception.

18.5.1.14 S-mode Cycle Counter (SCYCLE)

The SCYCLE register stores the number of cycles executed by the processor. While the processor is in an active execution state (i.e., not in a low-power state), the SCYCLE register increments its count on every clock cycle. The cycle counter is 64-bit wide and can be reset to zero.

For detailed information, please refer to *Event Counters* .

18.5.1.15 S-mode Instruction Retired Counter (SINSTRET)

The SINSTRET register is designed to store the numbers of retired instructions. And the SINSTRET register increments its count on every instruction retirement

The SINSTRET register is 64-bit wide and can be reset to zero.

For detailed information, please refer to *Event Counters* .

18.5.1.16 S-mode Event Counter (SHPMCOUNTERn)

SHPMCOUNTERn is the mapping of M-mode Event Counter (MHPMCOUNTERn) .

For detailed information, please refer to *Event Counters* .

18.6 Appendix C-6 C920 Extended U-mode Control Registers

This appendix provides a detailed explanation of the C920 Extended U-mode Control Registers.

18.6.1 U-mode Extended Floating Point Control Register Group

18.6.1.1 U-mode Floating Point Extended Control Register (FXCR)

U-mode Floating Point Extended Control Register (FXCR) is applied to floating-point extended switch and floating-point exception accumulation bit(s).

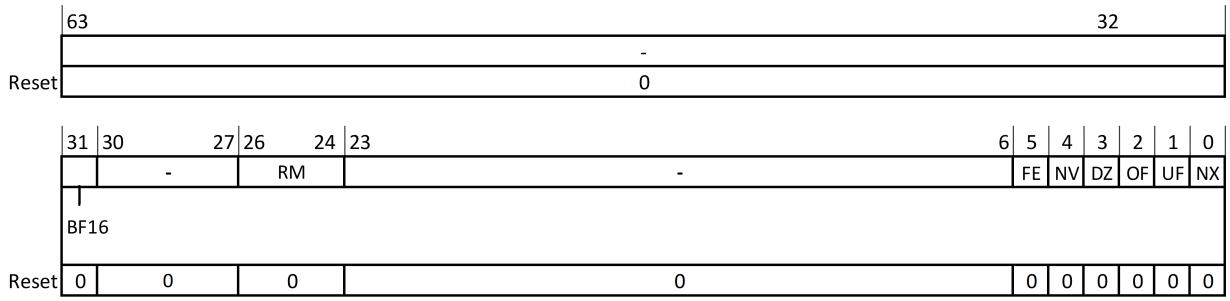


Fig. 18.50: **Floating Point Extended Control Register (FXCR)**

NX - Imprecise Exception:

Mapping to the corresponding bit in FCSR.

UF - Underflow Exception:

Mapping to the corresponding bit in FCSR.

OF - Overflow Exception:

Mapping to the corresponding bit in FCSR.

DZ - Division by Zero Exception:

Mapping to the corresponding bit in FCSR.

NV - Invalid Operation Exception:

Mapping to the corresponding bit in FCSR.

FE - Floating-Point Exception Sticky Bit:

This bit will be set to 1 when any floating-point exception occurs.

RM - Rounding Mode:

Mapping to the corresponding bit(s) in FCSR.

BF16 - Bfloat16 Switch Bit:

When this bit is set high, the processor ceases to support half precision floating-point numbers and instead processes 16-bit floating-point numbers according to the BFloat16 format.

CHAPTER 19

Apendix D Xuantie C900 Multi-core Synchronization Related Instructions and Program Implementations

19.1 Overview

The multi-core synchronization of XuanTie C900 is based on the RISC-V architecture, and complies with the definitions about instruction synchronization (fence.i), Translation Lookaside Buffer (TLB) maintenance (sfence.vma), and atomic instruction set extension in RISC-V privileged spec.

To improve maintenance efficiency in the scenario of multi-core and non-uniform bus, XuanTie C900 further enhances instruction synchronization, TLB maintenance, and DMA synchronization to meet different market requirements.

19.2 RISC-V Standard Instructions

19.2.1 fence Instruction

The basic RISC-V instruction set includes the fence instruction, which explicitly ensures the order of program instructions.

FENCE IORW, IORW

The fence instruction distinguishes the IO address space and memory address space. IO represents input/output, and RW represents read/write.

FENCE RW ensures that preceding read/write instructions are not executed later than the fence instruction.

FENCE RW ensures that subsequent read/write instructions are not executed before the fence instruction.

Similarly, the following instructions can be independently formed: FENCE R, RW / FENCE R, R / FENCE R / FENCE RW / FENCE RW, W ···

IO is equal to RW, and the following instruction can be formed: FENCE I, IO / FENCE I, I / FENCE I / FENCE IO / FENCE IO, O ···

Instructions can even be designed to mix IO and RW, such as FENCE RI, IORW / FENCE IORW, IORW ···

In summary, the FENCE instruction allows programmers to clearly and explicitly specify the required order of load/store operations with respect to memory or IO accesses, by defining a combination of its 8 bits representing preceding and subsequent R (read), W (write), I (input), and O (output).

19.2.2 fence.i Instruction

This instruction clears I-Cache to ensure that all the data access results before this instruction can be accessed by the fetch operations after the instruction.

19.2.3 sfence.vma Instruction

sfence.vma rs1,rs2 is applied to invalidation and synchronization of virtual memories. rs1 indicates the virtual address and rs2 indicates the Address Space Identifier (ASID).

- rs1=x0, rs2=x0: all Translation Lookaside Buffer (TLB) entries are invalidated
- rs1!=x0, rs2=x0: all TLB entries that hit the virtual address specified by rs1 are invalidated.
- rs1=x0, rs2!=x0: all TLB entries that hit the process ID specified by rs2 are invalidated.
- rs1!=x0, rs2!=x0: all TLB entries that hit the virtual address specified by rs1 and the process ID specified by rs2 are invalidated.

19.2.4 AMO Instruction

An atomic operation indicates the exclusive consecutive read, modify, and writeback operations on a shared memory address by multiple threads.

In a single-core system, exclusive operations are applicable if not being interrupted by interrupts/exceptions. Furthermore, in a single-core system, the memory model is relatively straightforward and aligns with programmers' intuition: a read operation will always return a value from the most recent write to the same address. Regardless of the design specifics of the Load-Store Unit (LSU) in a single-core CPU, it consistently ensures that programmers' expectations regarding memory accesses are met.

In a multi-core system, the memory model becomes significantly complex, and situations may no longer be intuitive. Questions like "Which write was the last one? Is the order of this read sequence guaranteed? And will this be the next write operation?" that do not require concern in single-core scenarios can become intricately tangled in a multi-core environment.

Currently, multiple memory ordering models are defined by different hardware implementations:

- Sequential consistency

- Processor consistency
- Weak consistency
- Release consistency (RISC-V)

As a result, the definition of atomic operation varies according to the architecture.

In the RISC-V architecture, Atomic Memory Operations (AMO) instructions cover a broad range of Arithmetic Logic Unit (ALU) operations, including addition, bitwise AND/OR/XOR, MIN/MAX, and so forth. These essentially meet Linux' s requirements for atomic operation primitives. However, there is an issue that the supported data types are relatively limited. In RV32 architectures, only word-sized operations are supported, while RV64 extends support to both word and double-word sized operations. The lack of support for half-word operations, though, poses a problem. Specifically, qspinlock has a strong requirement for an xchg operation on half words, which currently prevents RISC-V from effectively supporting qspinlock.

19.2.5 Load-Reserved/Store-Conditional Instruction

The Load-Reserved/Store-Conditional (LR/SC) instructions are widely applied in the ARM architecture. And the compare-and-swap (CAS) instruction in the x86 architecture is equivalent to the LR/SC instruction.

The definitions of LR/SC instructions in RISC-V are as follow:

LR is similar to load. It obtains data from a specified memory and monitors subsequent write operations of this address. After performing ALU calculation for the obtained data, the CPU uses the SC instruction to write a new value into the memory address of the previous LR operation. If no CPU write operation is performed on this memory address, the SC instruction writes the new value into the memory and sets rd to 0 (indicating success), like a common store instruction. Otherwise, the SC instruction does not write the new value into the memory, and sets rd to a non-zero value (indicating failure).

RISC-V lists the following advantages of LR/SC against CAS:

1. CAS suffers from the ABA problem, which means it only cares about the final state rather than the intermediate steps. If the value loaded previously matches the one fetched by CAS, the new value is successfully written. However, this might deviate from programmer expectations and can undermine atomicity since even if someone else has written to the same address or made two writes, reverting back to the initial value in the second write, CAS would still consider it a match. In contrast, LR/SC instructions monitor any write operations; even writing the same value would damage the SC instruction.
2. The hardware implementation of CAS is relatively complex, requiring three source registers and one destination register (to hold the result).
3. To address the ABA problem, certain systems provide a DW-CAS (Double-word Compare and Swap) instruction, which is more complex to implement, requiring five source registers and two destination registers.
4. LR/SC demonstrates better efficiency compared to CAS because CAS inherently involves an extra load instruction (i.e., load + CAS instructions), whereas LR + SC achieves the same functionality with just a single load instruction.

The above presents the reasons why RISC-V choose LR/SC over CAS instructions. In reality, however, software APIs do not provide robust support for LR/SC instructions. For instance, Linux only directly maps the cmpxchg primitive to the CAS instruction without providing a load_reserved/store_conditional primitive.

This results in the practical necessity of implementing cmpxchg using LR/SC operations:

```
# a0 holds address of memory location
# a1 holds expected value
# a2 holds desired value
# a0 holds return value, 0 if successful, !0 otherwise
cas:
lr.w t0, (a0) # Load original value.
bne t0, a1, fail # Doesn' t match, so fail.
sc.w t0, a2, (a0) # Try to update.
bnez t0, cas # Retry if store-conditional failed.
li a0, 0 # Set return to success.
jr ra # Return.
fail:
li a0, 1 # Set return to failure.
jr ra # Return.
```

Based on the loop structure of cmpxchg, double-loop implementation is formed:

```
c = v->counter;
while ((old = cmpxchg(&v->counter, c, c c_op i)) != c)
        c = old;
```

If this is the case, the advantages outlined in RISC-V' s reasons 1, 3, and 4 are negated, thus abandoning CAS has an negative impact on software compatibility. The presence of CAS support in arm64 serves as a good example.

The livelock problem of LR/SC is more complex. More problems may exist for Non Uniform Memory Access (NUMA) systems with more than 128 harts. (which will not be expanded upon in this article).

Compared to arm64, RISC-V' s LR/SC lacks the paired usage with LR/wfe, which prevents the implementation of a load_cond primitive (When a single core has multiple threads, the load_cond primitive instruction is required to stop occupying the pipeline).

19.3 Xuantie Enhancement Instruction

19.3.1 sync.is

This instruction ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction. When this instruction retires, the pipeline is cleared and the request is broadcast to other cores. This instruction can be used as the sync.s instruction (only for flush).

19.3.2 dcache.cipa rs1

This instruction writes the D-Cache/L2 Cache entry that hits the physical address specified by rs1 back to the lower-level store and invalidates this entry. This instruction can also be used as the dcache.cpa (only for flush) or dcache.ipa

(only for invalidation) instruction.

19.3.3 icache.iva rs1

This instruction invalidates the I-Cache entries corresponding to the virtual address specified by rs1.

19.4 Software Examples

The following presents examples of MMU (Memory Management Unit) and CACHE maintenance implementations for the Linux RISC-V architecture, along with software demonstrations of the relevant operations within OS.

19.4.1 TLB Maintenance

19.4.1.1 TLB flush

```
static inline void local_flush_tlb(unsigned long asid)
{
       __asm__ __volatile__ ("sfence.vma" : : : "memory");
}
```

19.4.1.2 Flush TLB Entries Associated with a Process Based on ASID

```
static inline void local_flush_tlb(unsigned long asid)
{
       __asm__ __volatile__ ("sfence.vma , %0" : : "r" (asid) : "memory");
}
```

19.4.1.3 Flush TLB Entries Based on VA

```
static inline void local_flush_tlb_range(unsigned long start, unsigned long size)
{
       unsigned long page_add = PAGE_DOWN(start);
       unsigned long page_end = PAGE_UP(start + size);

       while(page_add < page_end) {
               __asm__ __volatile__ ("sfence.vma %0, zero"
                                     :
                                     : "r" (page_add), "r" (asid)
                                     : "memory");
               page_add += PAGE_SIZE;
```

(continues on next page)

(continued from previous page)

```
        }
}
```

19.4.1.4 Flush TLB Entries Based on VA and ASID

```
static inline void local_flush_tlb_range_asid(unsigned long start, unsigned long size, unsigned␣
↪long asid)
{
        unsigned long page_add = PAGE_DOWN(start);
        unsigned long page_end = PAGE_UP(start + size);

        while(page_add < page_end) {
                __asm__ __volatile__ ("sfence.vma %0, %1"
                                      :
                                      : "r" (page_add), "r" (asid)
                                      : "memory");
                page_add += PAGE_SIZE;
        }
}
```

19.4.2 Instruction Area Synchronization

19.4.2.1 In-Core Global Instruction Area Synchronization

```
static inline void local_flush_icache_all(void)
{
        asm volatile ("fence.i" ::: "memory");
}
```

19.4.2.2 Multi-Core Global Instruction Area Synchronization

```
static void ipi_remote_fence_i(void *info)
{
        asm volatile ("fence.i" ::: "memory");
}

void flush_icache_all(void)
{
        on_each_cpu(ipi_remote_fence_i, NULL, 1);
}
```


19.4.2.3 Xuantie Multi-Core Precise Instruction Area Synchronization

```
static inline void flush_icache_range(unsigned long va_start, unsigned long size)
{
       register unsigned long i asm("a0") = va_start & ~(L1_CACHE_BYTES - 1);

       for (; i < (start + size); i += L1_CACHE_BYTES)
               __asm__ __volatile__ ("icache.iva" : : "r" (asid) : "memory");

       __asm__ __volatile__("sync.is");
}
```

19.4.3 DMA Synchronization

19.4.3.1 Xuantie Multi-Core Precise DMA Synchronization with Three Directions

```
void dma_sync_from_cpu_to_dev(unsigned long pa_start, unsigned long size)
{
       register unsigned long i asm("a0") = pa_start & ~(L1_CACHE_BYTES - 1);

       for (; i < (start + size); i += L1_CACHE_BYTES)
               __asm__ __volatile__ ("dcache.cpa" : : "r" (asid) : "memory");

       __asm__ __volatile__("sync.s");
}

void dma_sync_from_dev_to_cpu(unsigned long pa_start, unsigned long size)
{
       register unsigned long i asm("a0") = pa_start & ~(L1_CACHE_BYTES - 1);

       for (; i < (start + size); i += L1_CACHE_BYTES)
               __asm__ __volatile__ ("dcache.ipa" : : "r" (asid) : "memory");

       __asm__ __volatile__("sync.s");
}

void dma_sync_all(unsigned long pa_start, unsigned long size)
{
       register unsigned long i asm("a0") = pa_start & ~(L1_CACHE_BYTES - 1);

       for (; i < (start + size); i += L1_CACHE_BYTES)
               __asm__ __volatile__ ("dcache.cipa" : : "r" (asid) : "memory");
```

(continues on next page)

(continued from previous page)

```
        __asm__ __volatile__("sync.s");
}
```

19.4.4 AMO Implementations for Reference

The following content comes from the official Linux RISC-V architecture implementation of arch_atomic and cmpxchg.

```
/*
 * First, the atomic ops that have no ordering constraints and therefor don't
 * have the AQ or RL bits set.  These don't return anything, so there's only
 * one version to worry about.
 */
#define ATOMIC_OP(op, asm_op, I, asm_type, c_type, prefix)              \
static __always_inline                                                  \
void atomic##prefix##_##op(c_type i, atomic##prefix##_t *v)             \
{                                                                       \
        __asm__ __volatile__ (                                          \
                "       amo" #asm_op "." #asm_type " zero, %1, %0"      \
                : "+A" (v->counter)                                     \
                : "r" (I)                                               \
                : "memory");                                            \
}                                                                       \

#ifdef CONFIG_GENERIC_ATOMIC64
#define ATOMIC_OPS(op, asm_op, I)                                       \
        ATOMIC_OP (op, asm_op, I, w, int,   )
#else
#define ATOMIC_OPS(op, asm_op, I)                                       \
        ATOMIC_OP (op, asm_op, I, w, int,   )                           \
        ATOMIC_OP (op, asm_op, I, d, s64, 64)
#endif

ATOMIC_OPS(add, add,  i)
ATOMIC_OPS(sub, add, -i)
ATOMIC_OPS(and, and,  i)
ATOMIC_OPS( or,  or,  i)
ATOMIC_OPS(xor, xor,  i)

#undef ATOMIC_OP
#undef ATOMIC_OPS

/*
```

(continues on next page)

(continued from previous page)

```
 * Atomic ops that have ordered, relaxed, acquire, and release variants.
 * There's two flavors of these: the arithmatic ops have both fetch and return
 * versions, while the logical ops only have fetch versions.
 */
#define ATOMIC_FETCH_OP(op, asm_op, I, asm_type, c_type, prefix)        \
static __always_inline                                                  \
c_type atomic##prefix##_fetch_##op##_relaxed(c_type i,                  \
                                             atomic##prefix##_t *v)     \
{                                                                       \
        register c_type ret;                                            \
        __asm__ __volatile__ (                                          \
                "       amo" #asm_op "." #asm_type " %1, %2, %0"        \
                : "+A" (v->counter), "=r" (ret)                         \
                : "r" (I)                                               \
                : "memory");                                            \
        return ret;                                                     \
}                                                                       \
static __always_inline                                                  \
c_type atomic##prefix##_fetch_##op(c_type i, atomic##prefix##_t *v)     \
{                                                                       \
        register c_type ret;                                            \
        __asm__ __volatile__ (                                          \
                "       amo" #asm_op "." #asm_type ".aqrl  %1, %2, %0"  \
                : "+A" (v->counter), "=r" (ret)                         \
                : "r" (I)                                               \
                : "memory");                                            \
        return ret;                                                     \
}

#define ATOMIC_OP_RETURN(op, asm_op, c_op, I, asm_type, c_type, prefix) \
static __always_inline                                                  \
c_type atomic##prefix##_##op##_return_relaxed(c_type i,                 \
                                              atomic##prefix##_t *v)    \
{                                                                       \
        return atomic##prefix##_fetch_##op##_relaxed(i, v) c_op I;      \
}                                                                       \
static __always_inline                                                  \
c_type atomic##prefix##_##op##_return(c_type i, atomic##prefix##_t *v)  \
{                                                                       \
        return atomic##prefix##_fetch_##op(i, v) c_op I;                \
}

#ifdef CONFIG_GENERIC_ATOMIC64
```

(continues on next page)

(continued from previous page)

```
#define ATOMIC_OPS(op, asm_op, c_op, I)                                 \
        ATOMIC_FETCH_OP( op, asm_op,       I, w, int,   )               \
        ATOMIC_OP_RETURN(op, asm_op, c_op, I, w, int,   )
#else
#define ATOMIC_OPS(op, asm_op, c_op, I)                                 \
        ATOMIC_FETCH_OP( op, asm_op,       I, w, int,   )               \
        ATOMIC_OP_RETURN(op, asm_op, c_op, I, w, int,   )               \
        ATOMIC_FETCH_OP( op, asm_op,       I, d, s64, 64)               \
        ATOMIC_OP_RETURN(op, asm_op, c_op, I, d, s64, 64)
#endif

ATOMIC_OPS(add, add, +,  i)
ATOMIC_OPS(sub, add, +, -i)

#define atomic_add_return_relaxed       atomic_add_return_relaxed
#define atomic_sub_return_relaxed       atomic_sub_return_relaxed
#define atomic_add_return               atomic_add_return
#define atomic_sub_return               atomic_sub_return

#define atomic_fetch_add_relaxed        atomic_fetch_add_relaxed
#define atomic_fetch_sub_relaxed        atomic_fetch_sub_relaxed
#define atomic_fetch_add                atomic_fetch_add
#define atomic_fetch_sub                atomic_fetch_sub

#ifndef CONFIG_GENERIC_ATOMIC64
#define atomic64_add_return_relaxed     atomic64_add_return_relaxed
#define atomic64_sub_return_relaxed     atomic64_sub_return_relaxed
#define atomic64_add_return             atomic64_add_return
#define atomic64_sub_return             atomic64_sub_return

#define atomic64_fetch_add_relaxed      atomic64_fetch_add_relaxed
#define atomic64_fetch_sub_relaxed      atomic64_fetch_sub_relaxed
#define atomic64_fetch_add              atomic64_fetch_add
#define atomic64_fetch_sub              atomic64_fetch_sub
#endif

#undef ATOMIC_OPS

#ifdef CONFIG_GENERIC_ATOMIC64
#define ATOMIC_OPS(op, asm_op, I)                                       \
        ATOMIC_FETCH_OP(op, asm_op, I, w, int,   )
#else
#define ATOMIC_OPS(op, asm_op, I)                                       \
```

(continues on next page)

(continued from previous page)

```
        ATOMIC_FETCH_OP(op, asm_op, I, w, int,   )                      \
        ATOMIC_FETCH_OP(op, asm_op, I, d, s64, 64)
#endif

ATOMIC_OPS(and, and, i)
ATOMIC_OPS( or,  or, i)
ATOMIC_OPS(xor, xor, i)

#define atomic_fetch_and_relaxed        atomic_fetch_and_relaxed
#define atomic_fetch_or_relaxed         atomic_fetch_or_relaxed
#define atomic_fetch_xor_relaxed        atomic_fetch_xor_relaxed
#define atomic_fetch_and                atomic_fetch_and
#define atomic_fetch_or                 atomic_fetch_or
#define atomic_fetch_xor                atomic_fetch_xor

#ifndef CONFIG_GENERIC_ATOMIC64
#define atomic64_fetch_and_relaxed      atomic64_fetch_and_relaxed
#define atomic64_fetch_or_relaxed       atomic64_fetch_or_relaxed
#define atomic64_fetch_xor_relaxed      atomic64_fetch_xor_relaxed
#define atomic64_fetch_and              atomic64_fetch_and
#define atomic64_fetch_or               atomic64_fetch_or
#define atomic64_fetch_xor              atomic64_fetch_xor
#endif

#undef ATOMIC_OPS

#undef ATOMIC_FETCH_OP
#undef ATOMIC_OP_RETURN

/* This is required to provide a full barrier on success. */
static __always_inline int atomic_fetch_add_unless(atomic_t *v, int a, int u)
{
       int prev, rc;

        __asm__ __volatile__ (
                "0:     lr.w     %[p],  %[c]\n"
                "       beq      %[p],  %[u], 1f\n"
                "       add      %[rc], %[p], %[a]\n"
                "       sc.w.rl  %[rc], %[rc], %[c]\n"
                "       bnez     %[rc], 0b\n"
                "       fence    rw, rw\n"
                "1:\n"
                : [p]"=&r" (prev), [rc]"=&r" (rc), [c]"+A" (v->counter)
```

(continues on next page)

(continued from previous page)

```
                : [a]"r" (a), [u]"r" (u)
                : "memory");
        return prev;
}
#define atomic_fetch_add_unless atomic_fetch_add_unless

#ifndef CONFIG_GENERIC_ATOMIC64
static __always_inline s64 atomic64_fetch_add_unless(atomic64_t *v, s64 a, s64 u)
{
       s64 prev;
       long rc;

        __asm__ __volatile__ (
                "0:     lr.d     %[p],  %[c]\n"
                "       beq      %[p],  %[u], 1f\n"
                "       add      %[rc], %[p], %[a]\n"
                "       sc.d.rl  %[rc], %[rc], %[c]\n"
                "       bnez     %[rc], 0b\n"
                "       fence    rw, rw\n"
                "1:\n"
                : [p]"=&r" (prev), [rc]"=&r" (rc), [c]"+A" (v->counter)
                : [a]"r" (a), [u]"r" (u)
                : "memory");
        return prev;
}
#define atomic64_fetch_add_unless atomic64_fetch_add_unless
#endif

/*
 * atomic_{cmp,}xchg is required to have exactly the same ordering semantics as
 * {cmp,}xchg and the operations that return, so they need a full barrier.
 */
#define ATOMIC_OP(c_t, prefix, size)                                    \
static __always_inline                                                  \
c_t atomic##prefix##_xchg_relaxed(atomic##prefix##_t *v, c_t n)          \
{                                                                       \
        return __xchg_relaxed(&(v->counter), n, size);                  \
}                                                                       \
static __always_inline                                                  \
c_t atomic##prefix##_xchg_acquire(atomic##prefix##_t *v, c_t n)          \
{                                                                       \
        return __xchg_acquire(&(v->counter), n, size);                  \
}                                                                       \
```

(continues on next page)

(continued from previous page)

```
static __always_inline                                                  \
c_t atomic##prefix##_xchg_release(atomic##prefix##_t *v, c_t n)         \
{                                                                       \
        return __xchg_release(&(v->counter), n, size);                  \
}                                                                       \
static __always_inline                                                  \
c_t atomic##prefix##_xchg(atomic##prefix##_t *v, c_t n)                 \
{                                                                       \
        return __xchg(&(v->counter), n, size);                          \
}                                                                       \
static __always_inline                                                  \
c_t atomic##prefix##_cmpxchg_relaxed(atomic##prefix##_t *v,             \
                                     c_t o, c_t n)                      \
{                                                                       \
        return __cmpxchg_relaxed(&(v->counter), o, n, size);            \
}                                                                       \
static __always_inline                                                  \
c_t atomic##prefix##_cmpxchg_acquire(atomic##prefix##_t *v,             \
                                     c_t o, c_t n)                      \
{                                                                       \
        return __cmpxchg_acquire(&(v->counter), o, n, size);            \
}                                                                       \
static __always_inline                                                  \
c_t atomic##prefix##_cmpxchg_release(atomic##prefix##_t *v,             \
                                     c_t o, c_t n)                      \
{                                                                       \
        return __cmpxchg_release(&(v->counter), o, n, size);            \
}                                                                       \
static __always_inline                                                  \
c_t atomic##prefix##_cmpxchg(atomic##prefix##_t *v, c_t o, c_t n)        \
{                                                                       \
        return __cmpxchg(&(v->counter), o, n, size);                    \
}

#ifdef CONFIG_GENERIC_ATOMIC64
#define ATOMIC_OPS()                                                    \
        ATOMIC_OP(int,   , 4)
#else
#define ATOMIC_OPS()                                                    \
        ATOMIC_OP(int,   , 4)                                           \
        ATOMIC_OP(s64, 64, 8)
#endif
```

(continues on next page)

(continued from previous page)

```
ATOMIC_OPS()

#define atomic_xchg_relaxed atomic_xchg_relaxed
#define atomic_xchg_acquire atomic_xchg_acquire
#define atomic_xchg_release atomic_xchg_release
#define atomic_xchg atomic_xchg
#define atomic_cmpxchg_relaxed atomic_cmpxchg_relaxed
#define atomic_cmpxchg_acquire atomic_cmpxchg_acquire
#define atomic_cmpxchg_release atomic_cmpxchg_release
#define atomic_cmpxchg atomic_cmpxchg

#undef ATOMIC_OPS
#undef ATOMIC_OP

static __always_inline int atomic_sub_if_positive(atomic_t *v, int offset)
{
       int prev, rc;

        __asm__ __volatile__ (
                "0:     lr.w     %[p],  %[c]\n"
                "       sub      %[rc], %[p], %[o]\n"
                "       bltz     %[rc], 1f\n"
                "       sc.w.rl  %[rc], %[rc], %[c]\n"
                "       bnez     %[rc], 0b\n"
                "       fence    rw, rw\n"
                "1:\n"
                : [p]"=&r" (prev), [rc]"=&r" (rc), [c]"+A" (v->counter)
                : [o]"r" (offset)
                : "memory");
        return prev - offset;
}

#define atomic_dec_if_positive(v)       atomic_sub_if_positive(v, 1)

#ifndef CONFIG_GENERIC_ATOMIC64
static __always_inline s64 atomic64_sub_if_positive(atomic64_t *v, s64 offset)
{
       s64 prev;
       long rc;

        __asm__ __volatile__ (
                "0:     lr.d     %[p],  %[c]\n"
                "       sub      %[rc], %[p], %[o]\n"
```

(continues on next page)

(continued from previous page)

```
                "       bltz    %[rc], 1f\n"
                "       sc.d.rl %[rc], %[rc], %[c]\n"
                "       bnez    %[rc], 0b\n"
                "       fence   rw, rw\n"
                "1:\n"
                : [p]"=&r" (prev), [rc]"=&r" (rc), [c]"+A" (v->counter)
                : [o]"r" (offset)
                : "memory");
        return prev - offset;
}

#define __xchg_relaxed(ptr, new, size)                                  \
({                                                                      \
        __typeof__(ptr) __ptr = (ptr);                                  \
        __typeof__(new) __new = (new);                                  \
        __typeof__(*(ptr)) __ret;                                       \
        switch (size) {                                                 \
        case 4:                                                         \
                __asm__ __volatile__ (                                  \
                        "       amoswap.w %0, %2, %1\n"                 \
                        : "=r" (__ret), "+A" (*__ptr)                   \
                        : "r" (__new)                                   \
                        : "memory");                                    \
                break;                                                  \
        case 8:                                                         \
                __asm__ __volatile__ (                                  \
                        "       amoswap.d %0, %2, %1\n"                 \
                        : "=r" (__ret), "+A" (*__ptr)                   \
                        : "r" (__new)                                   \
                        : "memory");                                    \
                break;                                                  \
        default:                                                        \
                BUILD_BUG();                                            \
        }                                                               \
        __ret;                                                          \
})

#define xchg_relaxed(ptr, x)                                            \
({                                                                      \
        __typeof__(*(ptr)) _x_ = (x);                                   \
        (__typeof__(*(ptr))) __xchg_relaxed((ptr),                      \
                                        _x_, sizeof(*(ptr)));           \
})
```

(continues on next page)

(continued from previous page)

```
#define __xchg_acquire(ptr, new, size)                                  \
({                                                                      \
        __typeof__(ptr) __ptr = (ptr);                                  \
        __typeof__(new) __new = (new);                                  \
        __typeof__(*(ptr)) __ret;                                       \
        switch (size) {                                                 \
        case 4:                                                         \
                __asm__ __volatile__ (                                  \
                        "       amoswap.w %0, %2, %1\n"                 \
                        RISCV_ACQUIRE_BARRIER                           \
                        : "=r" (__ret), "+A" (*__ptr)                   \
                        : "r" (__new)                                   \
                        : "memory");                                    \
                break;                                                  \
        case 8:                                                         \
                __asm__ __volatile__ (                                  \
                        "       amoswap.d %0, %2, %1\n"                 \
                        RISCV_ACQUIRE_BARRIER                           \
                        : "=r" (__ret), "+A" (*__ptr)                   \
                        : "r" (__new)                                   \
                        : "memory");                                    \
                break;                                                  \
        default:                                                        \
                BUILD_BUG();                                            \
        }                                                               \
        __ret;                                                          \
})

#define xchg_acquire(ptr, x)                                            \
({                                                                      \
        __typeof__(*(ptr)) _x_ = (x);                                   \
        (__typeof__(*(ptr))) __xchg_acquire((ptr),                      \
                                            _x_, sizeof(*(ptr)));       \
})

#define __xchg_release(ptr, new, size)                                  \
({                                                                      \
        __typeof__(ptr) __ptr = (ptr);                                  \
        __typeof__(new) __new = (new);                                  \
        __typeof__(*(ptr)) __ret;                                       \
        switch (size) {                                                 \
        case 4:                                                         \
```

(continues on next page)

(continued from previous page)

```
                __asm__ __volatile__ (                                  \
                        RISCV_RELEASE_BARRIER                           \
                        "       amoswap.w %0, %2, %1\n"                 \
                        : "=r" (__ret), "+A" (*__ptr)                   \
                        : "r" (__new)                                   \
                        : "memory");                                    \
                break;                                                  \
        case 8:                                                         \
                __asm__ __volatile__ (                                  \
                        RISCV_RELEASE_BARRIER                           \
                        "       amoswap.d %0, %2, %1\n"                 \
                        : "=r" (__ret), "+A" (*__ptr)                   \
                        : "r" (__new)                                   \
                        : "memory");                                    \
                break;                                                  \
        default:                                                        \
                BUILD_BUG();                                            \
        }                                                               \
        __ret;                                                          \
})

#define xchg_release(ptr, x)                                            \
({                                                                      \
        __typeof__(*(ptr)) _x_ = (x);                                   \
        (__typeof__(*(ptr))) __xchg_release((ptr),                      \
                                            _x_, sizeof(*(ptr)));       \
})

#define __xchg(ptr, new, size)                                          \
({                                                                      \
        __typeof__(ptr) __ptr = (ptr);                                  \
        __typeof__(new) __new = (new);                                  \
        __typeof__(*(ptr)) __ret;                                       \
        switch (size) {                                                 \
        case 4:                                                         \
                __asm__ __volatile__ (                                  \
                        "       amoswap.w.aqrl %0, %2, %1\n"            \
                        : "=r" (__ret), "+A" (*__ptr)                   \
                        : "r" (__new)                                   \
                        : "memory");                                    \
                break;                                                  \
        case 8:                                                         \
                __asm__ __volatile__ (                                  \
```

(continues on next page)

(continued from previous page)

```
                        "       amoswap.d.aqrl %0, %2, %1\n"            \
                        : "=r" (__ret), "+A" (*__ptr)                   \
                        : "r" (__new)                                   \
                        : "memory");                                    \
                break;                                                  \
        default:                                                        \
                BUILD_BUG();                                            \
        }                                                               \
        __ret;                                                          \
})

#define xchg(ptr, x)                                                    \
({                                                                      \
        __typeof__(*(ptr)) _x_ = (x);                                   \
        (__typeof__(*(ptr))) __xchg((ptr), _x_, sizeof(*(ptr)));        \
})

#define xchg32(ptr, x)                                                  \
({                                                                      \
        BUILD_BUG_ON(sizeof(*(ptr)) != 4);                              \
        xchg((ptr), (x));                                               \
})

#define xchg64(ptr, x)                                                  \
({                                                                      \
        BUILD_BUG_ON(sizeof(*(ptr)) != 8);                              \
        xchg((ptr), (x));                                               \
})

/*
 * Atomic compare and exchange.  Compare OLD with MEM, if identical,
 * store NEW in MEM.  Return the initial value in MEM.  Success is
 * indicated by comparing RETURN with OLD.
 */
#define __cmpxchg_relaxed(ptr, old, new, size)                          \
({                                                                      \
        __typeof__(ptr) __ptr = (ptr);                                  \
        __typeof__(*(ptr)) __old = (old);                               \
        __typeof__(*(ptr)) __new = (new);                               \
        __typeof__(*(ptr)) __ret;                                       \
        register unsigned int __rc;                                     \
        switch (size) {                                                 \
        case 4:                                                         \
```

(continues on next page)

(continued from previous page)

```
                __asm__ __volatile__ (                                  \
                        "0:     lr.w %0, %2\n"                          \
                        "       bne  %0, %z3, 1f\n"                     \
                        "       sc.w %1, %z4, %2\n"                     \
                        "       bnez %1, 0b\n"                          \
                        "1:\n"                                          \
                        : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)     \
                        : "rJ" ((long)__old), "rJ" (__new)              \
                        : "memory");                                    \
                break;                                                  \
        case 8:                                                         \
                __asm__ __volatile__ (                                  \
                        "0:     lr.d %0, %2\n"                          \
                        "       bne %0, %z3, 1f\n"                      \
                        "       sc.d %1, %z4, %2\n"                     \
                        "       bnez %1, 0b\n"                          \
                        "1:\n"                                          \
                        : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)     \
                        : "rJ" (__old), "rJ" (__new)                    \
                        : "memory");                                    \
                break;                                                  \
        default:                                                        \
                BUILD_BUG();                                            \
        }                                                               \
        __ret;                                                          \
})

#define cmpxchg_relaxed(ptr, o, n)                                      \
({                                                                      \
        __typeof__(*(ptr)) _o_ = (o);                                   \
        __typeof__(*(ptr)) _n_ = (n);                                   \
        (__typeof__(*(ptr))) __cmpxchg_relaxed((ptr),                   \
                                       _o_, _n_, sizeof(*(ptr)));       \
})

#define __cmpxchg_acquire(ptr, old, new, size)                          \
({                                                                      \
        __typeof__(ptr) __ptr = (ptr);                                  \
        __typeof__(*(ptr)) __old = (old);                               \
        __typeof__(*(ptr)) __new = (new);                               \
        __typeof__(*(ptr)) __ret;                                       \
        register unsigned int __rc;                                     \
        switch (size) {                                                 \
```

(continues on next page)

(continued from previous page)

```
        case 4:                                                         \
                __asm__ __volatile__ (                                  \
                        "0:     lr.w %0, %2\n"                          \
                        "       bne  %0, %z3, 1f\n"                     \
                        "       sc.w %1, %z4, %2\n"                     \
                        "       bnez %1, 0b\n"                          \
                        RISCV_ACQUIRE_BARRIER                           \
                        "1:\n"                                          \
                        : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)     \
                        : "rJ" ((long)__old), "rJ" (__new)              \
                        : "memory");                                    \
                break;                                                  \
        case 8:                                                         \
                __asm__ __volatile__ (                                  \
                        "0:     lr.d %0, %2\n"                          \
                        "       bne %0, %z3, 1f\n"                      \
                        "       sc.d %1, %z4, %2\n"                     \
                        "       bnez %1, 0b\n"                          \
                        RISCV_ACQUIRE_BARRIER                           \
                        "1:\n"                                          \
                        : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)     \
                        : "rJ" (__old), "rJ" (__new)                    \
                        : "memory");                                    \
                break;                                                  \
        default:                                                        \
                BUILD_BUG();                                            \
        }                                                               \
        __ret;                                                          \
})

#define cmpxchg_acquire(ptr, o, n)                                      \
({                                                                      \
        __typeof__(*(ptr)) _o_ = (o);                                   \
        __typeof__(*(ptr)) _n_ = (n);                                   \
        (__typeof__(*(ptr))) __cmpxchg_acquire((ptr),                   \
                                       _o_, _n_, sizeof(*(ptr)));       \
})

#define __cmpxchg_release(ptr, old, new, size)                          \
({                                                                      \
        __typeof__(ptr) __ptr = (ptr);                                  \
        __typeof__(*(ptr)) __old = (old);                               \
        __typeof__(*(ptr)) __new = (new);                               \
```

(continues on next page)

(continued from previous page)

```
        __typeof__(*(ptr)) __ret;                                       \
        register unsigned int __rc;                                     \
        switch (size) {                                                 \
        case 4:                                                         \
                __asm__ __volatile__ (                                  \
                        RISCV_RELEASE_BARRIER                           \
                        "0:     lr.w %0, %2\n"                          \
                        "       bne  %0, %z3, 1f\n"                     \
                        "       sc.w %1, %z4, %2\n"                     \
                        "       bnez %1, 0b\n"                          \
                        "1:\n"                                          \
                        : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)    \
                        : "rJ" ((long)__old), "rJ" (__new)              \
                        : "memory");                                    \
                break;                                                  \
        case 8:                                                         \
                __asm__ __volatile__ (                                  \
                        RISCV_RELEASE_BARRIER                           \
                        "0:     lr.d %0, %2\n"                          \
                        "       bne %0, %z3, 1f\n"                      \
                        "       sc.d %1, %z4, %2\n"                     \
                        "       bnez %1, 0b\n"                          \
                        "1:\n"                                          \
                        : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)    \
                        : "rJ" (__old), "rJ" (__new)                    \
                        : "memory");                                    \
                break;                                                  \
        default:                                                        \
                BUILD_BUG();                                            \
        }                                                               \
        __ret;                                                          \
})

#define cmpxchg_release(ptr, o, n)                                      \
({                                                                      \
        __typeof__(*(ptr)) _o_ = (o);                                   \
        __typeof__(*(ptr)) _n_ = (n);                                   \
        (__typeof__(*(ptr))) __cmpxchg_release((ptr),                   \
                                       _o_, _n_, sizeof(*(ptr)));       \
})

#define __cmpxchg(ptr, old, new, size)                                  \
({                                                                      \
```

(continues on next page)

(continued from previous page)

```
        __typeof__(ptr) __ptr = (ptr);                                  \
        __typeof__(*(ptr)) __old = (old);                               \
        __typeof__(*(ptr)) __new = (new);                               \
        __typeof__(*(ptr)) __ret;                                       \
        register unsigned int __rc;                                     \
        switch (size) {                                                 \
        case 4:                                                         \
                __asm__ __volatile__ (                                  \
                        "0:     lr.w %0, %2\n"                          \
                        "       bne  %0, %z3, 1f\n"                     \
                        "       sc.w.rl %1, %z4, %2\n"                  \
                        "       bnez %1, 0b\n"                          \
                        "       fence rw, rw\n"                         \
                        "1:\n"                                          \
                        : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)    \
                        : "rJ" ((long)__old), "rJ" (__new)              \
                        : "memory");                                    \
                break;                                                  \
        case 8:                                                         \
                __asm__ __volatile__ (                                  \
                        "0:     lr.d %0, %2\n"                          \
                        "       bne %0, %z3, 1f\n"                      \
                        "       sc.d.rl %1, %z4, %2\n"                  \
                        "       bnez %1, 0b\n"                          \
                        "       fence rw, rw\n"                         \
                        "1:\n"                                          \
                        : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)    \
                        : "rJ" (__old), "rJ" (__new)                    \
                        : "memory");                                    \
                break;                                                  \
        default:                                                        \
                BUILD_BUG();                                            \
        }                                                               \
        __ret;                                                          \
})
```

