Nuclei® RISC-V Instruction Set Architecture Specification

Nucleisys

Contents

1 Copyright Notice

The product described herein is subject to continuous development and improvement; information herein is given by Nuclei in good faith but without warranties.

This document is intended only to assist the reader in the use of the product. Nuclei do not assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation indirect, incidental, special, exemplary, or consequential damages. incorrect use of the product.

2 Contact Information

Should you have any problems with the information contained herein or any suggestions, please contact Nuclei System Technology by email support@nucleisys.com, or visit "Nuclei User Center" website http://user.nucleisys.com for supports or online discussion.

List of Figures

List of Tables

3 Revision History

Rev.	Revision Date	Revised Section	Revised Content
1.1.0	2020/1/20	N/A	1.First version as the full English
2.0.0	2020/4/28	15	1.Add some cfg_info CSR registers 2.Refer to Nuclei_DSP_QuickStart 2.0.0 version 3.Refer to Nuclei_CCM_Mechanism 1.0 version
2.0.1	2020/5/27	15.5.7	1.Change the mcache_ctl[1]/scrathpad mode default value to be 1 after reset
2.0.2	2020/6/29	15, 16	1.Add ECC related CSRs 2.ADD ECC introduction
2.0.3	2020/8/6	8	1.Update TIMER to be compatiable with CLINT mode for UX class to run Linux
2.0.3	2020/11/24	9	1.Update the PLIC registers memory map
2.1.0	2021/1/20	15.4	1.Add marchid and mimpid to indicate the corresponding RTL hardware version, Core ID, databook and ISA Spec version.
2.2.0	2021/2/4	12.2.3	1.Add PMP TOR supported
2.3.0	2021/5/18	17	1.Add Performance Monitor descriptions
2.4.0	2021/8/2	15.4.5	1.Add support hartid to be non-zero value in non-SMP system
2.5.0	2021/10/17	11,21,22,24	1.Add CIDU chapter 2.Add Bit Manipulation chapter 3.Add Vector chapter 4.Add SMP and Cluster Cache chapter
2.6.0	2021/12/18	19,27	1.Add CLM and CLM Slave Port 2.Bus Error can be configured to exception or interrupt. 3.BPU enable/disable
2.6.1	2022/02/24	19	1.Add mode of Instruction Trace in related CSR.
2.6.2	2022/03/01	19	1.Add mode of BF16 in related CSR.
2.6.3	2022/05/16	19	1.Add IREGION in related CSR.
2.7.0	2022/11/22	5,19	1.Nuclei Core supports RISC-V K Ext. 2.Nuclei Core supports secure/non-secure mode.
2.8.0	2023/02/08	6,19	1.Nuclei Processor core's privilege spec is updated to RISC-V Privilege Spec v.20211203. 2.Nuclei 900 Series v2.8.0 follows latest RISC-V Profile. 3.Nuclei UX900 v2.8.0 or later supports CSR shartid.
3.0.0	2023/04/20		1.Fix some syntax problems 2.Add some ecc register description

continues on next page

Table 3.1 – continued from previous page

Rev.	Revision Date	Revised Section	Revised Content
3.1.0	2023/06/09		1.Remove two level exception related CSRs. 2.Secure Mode updated to RISC-V Smwg Extension. 3.Device & Noncacheable region CSR name changed and updated.
4.0.0	2024/8/1		1. Merge CCM doc into this doc. 2. Merge TEE doc into this doc. 3. Merge NICE doc into this doc. 4. Merge DSP doc into this doc. 5. Merge XLCZ doc into this doc. 6. Add U class Core 7. Add some RISCV extensions CPU support. 8. Change riscv isa and privileged spec version from 20191213 to 20240411. 9. Merge worldguard spec into this doc. 10. Merge aclint spec into this doc. 11. Merge Riscv smepmp extension v1.0 into this doc. 12. Add Secure states description.
4.1.0	2024/8/25	17,19	1. Add PMA chapter. 2. Add MMU U32 feature description.
4.2.0	2024/9/1	24,32	1. Update the hpmevent register. 2. Add some register into CC.
4.3.0	2024/9/26	22	1. Update the mcfg_info register.

4 Introduction

This document describes the ISA (Instruction Set Architecture) implemented in Nuclei processor core, including the instruction set and privileged architecture features.

Basically, Nuclei processor core are following and compatible to RISC-V standard architecture, but there might be some additions and enhancements to the original standard spec.

To respect the RISC-V standard, this document may not repeat the contents of original RISC-V standard, but will highlight the additions and enhancements of Nuclei defined.

5 Nuclei RISC-V Instruction Set Overview

5.1 RISC-V ISA Extensions supported by Nuclei Core

Nuclei processor core follows the RISC-V instruction set standard (riscv-spec-20240411.pdf).

RISC-V is the configurable modular instruction set. Nuclei processor core support the following RISC-V Extensions:

- Rv32E, v1.9: 32bits architecture, with 16 general purpose registers
- RV32I, v2.1: 32bits architecture, with 32 general purpose registers
- RV64I, v2.1: 64bits architecture, with 32 general purpose registers
- Zicsr, v2.0: Control and Status register Extension.
- Zicntr/Zihpm, v2.0: Counters Extension.
- Zihintntl, v1.0: Non-Temporal Locality Hints Extension.
- Zihintpause, v1.0: Pause Hint Extension.
- M, v2.0: Integer Multiplication and Division instructions.
- Zmmul, v1.0: Integer Multiplication instructions.
- C, v2.0: Compressed Instructions as 16bits Encoding to reduce code size.
- A, v2.1: Atomic Instructions.
- B, v1.0: Bit Manipulation Instructions.
- F, v2.2: Single-Precision Floating-Point Instructions.
- D, v2.2: Double-Precision Floating-Point Instructions.
- Zfh, v1.0: Half-Precision Floating-Point Instructions.
- Zfhmin, v1.0: Mini Half-Precision Floating-Point Instructions.
- Zfa, v1.0: Additional Floating-Point Instructions.
- BF16, v1.0: BF16 Format Floating-Point Instructions.
- P, v0.5.4: Packed-SIMD Instructions.
- V, v1.0: Vector Instructions.
- Zvb, v1.0.0: Vector bit-manipulation Instructions.
- K, v1.0: Scalar & Entropy Source Instructions.
- Zvk, v1.0.0: Vector Scalar Instructions.
- Zc, v1.0: Group of extensions which define subsets of the existing C extension (Zca, Zcd, Zcf) and new extensions which only contain 16-bit encodings.

- Zicond, v1.0: Integer Conditional Operations Extension.
- CMO, v1.0: Base Cache Management Operation ISA Extension.
- Etrace, v1.0: Processor Trace.
- Sstc, v1.0.0: S-mode timer interrupt Extension.
- Svnapot, v1.0: NAPOT Translation Extension.
- Svpbmt, v1.0: Page-Based Memory Types Extension.
- Svinval, v1.0: Fine-Grained Address-Translation Cache Invalidation Extension.
- Smepmp, v1.0: PMP Enhancements for memory access and execution prevention on Machine mode Extension.
- Smwg, v0.4: WorldGuard Extension.
- Sscofpmf, v1.0.0: Count Overflow and Mode-Based Filtering Extension.
- Smpu, v0.9.0: S-mode Memory Protection Extension.
- Smclic,Ssclic,Smclicshv,Smclicconfig, v0.9: Core-Local Interrupt Controller(CLIC) Extension.
- Advanced Core Local interrupt, v1.0: MTIMER, MSWI and SSWI.
- Platform-Level Interrupt Controller(PLIC), v1.0.0_rc5.

According to the naming rule from RISC-V standard, the above-mentioned instruction set module can be combined, e.g., RV32IMAC, RV32IMFC, RV32IMAFDC, RV32IMAFDCP, etc. RISC-V standard also use the abbreviation G for the IMAFD combination, hence, RV32IMAFDC or RV64IMAFDC can be also abbreviated as RV32GC or RV64GC.

5.2 Nuclei Processor Core Classes: N, U, NX and UX

To differentiate the IP products with different positions, Nuclei divides the core products into 4 classes:

- Nuclei N class: support RV32I or RV32E, for the 32bits microcontroller applications.
  - Nuclei N200 series core can be configured to support RV32E or RV32I.
  - Nuclei N300, N600, N900 series core only support RV32I.
- Nuclei U class: support RV32I with MMU, for the 32bits Linux capable applications.
  - The MMU can also be turned off by software, in this mode, U class core can also work as microcontroller, i.e., Nuclei U class core is downward-compatible to Nuclei N class core.
- Nuclei NX class: support RV64I without MMU, for the 64bits microcontroller applications.
- Nuclei UX class: support RV64I with MMU, for the 64bits Linux capable applications.
  - The MMU can also be turned off by software, in this mode, UX class core can also work as microcontroller, i.e., Nuclei UX class core is downward-compatible to Nuclei NX class core.

6 Nuclei RISC-V Privileged Architecture

6.1 RISC-V Privileged Architecture supported by Nuclei Core

Nuclei processor core follows the RISC-V privileged architecture standard (riscv-privileged-20240411.pdf).

And Nuclei 900 series cores v2.8.0 or later follow latest RISC-V Profile , N900 and NX900 is compatible to RVI22 Profile and UX900 is compatible to RVA22 Profile.

Basically, Nuclei processor core are following and compatible to RISC-V standard privileged architecture, but there might be some additions and enhancements to the original standard spec.

To respect the RISC-V standard, this document may not repeat the contents of original RISC-V standard, but will highlight the additions and enhancements of Nuclei defined.

6.2 Privileged Modes

Following the RISC-V privileged architecture standard, Nuclei processor core support following Privilege Modes:

- Machine Mode
- Supervisor Mode
- User Mode

Note: According to the RISC-V standard privileged architecture, there is no way for the software to check current privileged mode (e.g., machine mode or user mode).

Please refer to RISC-V standard privileged architecture for more details.

6.3 Debug Modes

Nuclei processor core also support debug mode to support off-chip debugging. Nuclei processor core follows the RISC-V debug standard (riscv-debug-spec-v.0.13.2.pdf).

6.4 Machine Sub-Mode added by Nuclei

Besides the above-mentioned standard Privilege Modes, Nuclei processor core further defined 4 types of sub-mode, to differentiate the exact machine mode status, called Machine Sub-Mode:

- Normal Machine Mode
  - The processor core will be under this default sub-mode when out of reset.
  - If the processor core does not encounter exception, NMI, interrupt, debug request, or does not switch mode explicitly, then it will remain in this sub-mode.
- Exception Handling Mode
  - The processor core will be under this sub-mode when the core encountered exception trap. Please refer to *Exception Handling in Nuclei processor core* (page 16) for more details.
- NMI Handling Mode
  - The processor core will be under this sub-mode when the core encountered NMI trap. Please refer to *NMI Handling in Nuclei processor core* (page 19) for more details.
- Interrupt Handling Mode
  - The processor core will be under this sub-mode when the core encountered interrupt trap. Please refer to *Interrupt Handling in Nuclei processor core* (page 23) for more details.

Nuclei defined a CSR register msubm to reflect processor core's current machine sub-mode (msubm.TYP) and previous machine sub-mode (msubm.PTYP). Please refer to *msubm* (page 115) for more details of CSR register msubm.

7 Exception Handling in Nuclei processor core

7.1 Exception Overview

Exception is that the processor core suddenly encounters an abnormal situation when executing the program instruction stream, and aborts execution of the current program, and turns to handle the exception instead. The key points are as follows:

- The "abnormal event" which the core encounters is called an exception. An exception is caused by an internal event in the core or an event during the execution of the program, such as a hardware failure, a program failure, or the execution of a special system call instruction. In short, it is a core-internal issue.
- When the exception is taken, the core will enter the exception handler program.

7.2 Exception Masking

According to the RISC-V architecture, exception cannot be masked, which means if the core encounters an exception, it must stop current execution and turns to handle the exception.

7.3 Priority of Exception

It is possible that the core encounters multiple exceptions at the same time, so exceptions also have priority. The priority of the exception is defined in RISC-V standard privileged architecture, please refer to RISC-V standard privileged architecture for more details.

7.4 Entering Exception Handling Mode

Taking an exception, hardware behaviors of the Nuclei processor core are shown in *The overall process of exception* (page 17). Note that the following operations are done simultaneously in one cycle:

- Stop the execution of the current program, and start from the PC address defined by the CSR mtvec. Update the CSR registers: mcause, mepc, mtval, and mstatus. Update the Privilege Mode.
  - These behaviors are following RISC-V standard privileged architecture specification. This document will not repeat its content, please refer to RISC-V standard privileged architecture specification for more details.
- Update the Machine Sub-Mode of the core.
  - This is unique in Nuclei processor core, and will be detailed in the following section.

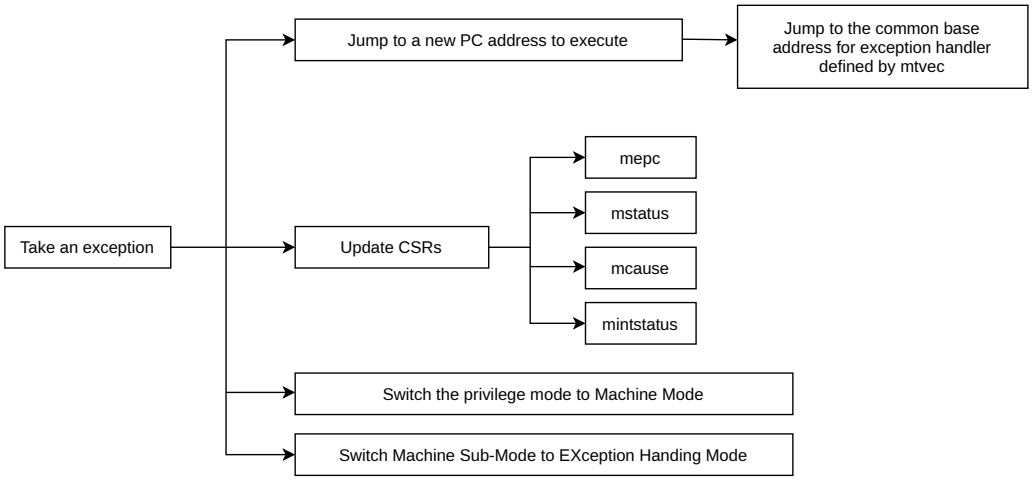


Fig. 7.1: The overall process of exception

7.4.1 Update the Machine Sub-Mode

The Machine Sub-Mode of the Nuclei processor core is indicated in the msubm.TYP filed in real time. When the core takes an exception, the Machine Sub-Mode will be updated to exception handling mode, so:

- The filed msubm.TYP is updated to exception handling mode, as shown in *The CSR mstatus and msubm updating when enter/exit the exception* (page 17), to reflect the current Machine Sub-Mode is "Exception Handling Mode".
- The value of msubm.PTYP will be updated to the value of msub.TYP before taking the exception, as shown in *The CSR mstatus and msubm updating when enter/exit the exception* (page 17). The value of msubm.PTYP will be used to restore the value of msubm.PTYP after exiting the exception handler.

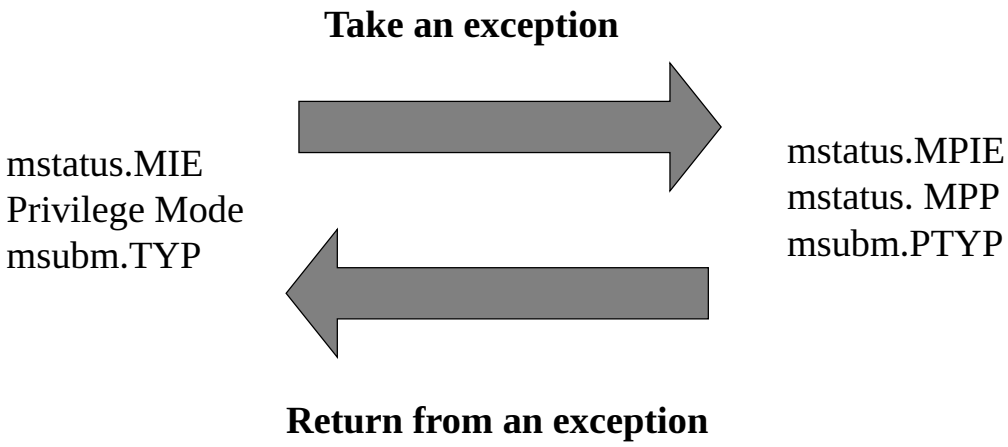


Fig. 7.2: The CSR mstatus and msubm updating when enter/exit the exception

7.5 Exit the Exception Handling Mode

After handling the exception, the core needs to exit from the exception handler eventually. Since the exception is handling in Machine Mode, the software has to execute mret to exit the exception handler.

The hardware behavior of the processor after executing mret instruction is as shown in *The overall process of exiting an exception* (page 18). Note that the following hardware behaviors are done simultaneously in one cycle:

- Stop the execution of the current program, and start from the PC address defined by the CSR mepc. Update the CSR mstatus. And update the Privilege Mode.
  - These behaviors are following RISC-V standard privileged architecture specification. This document will not repeat its content, please refer to RISC-V standard privileged architecture specification for more details.

- Update the Machine Sub-Mode of the core.
  - This is unique in Nuclei processor core, and will be detailed in the following section.

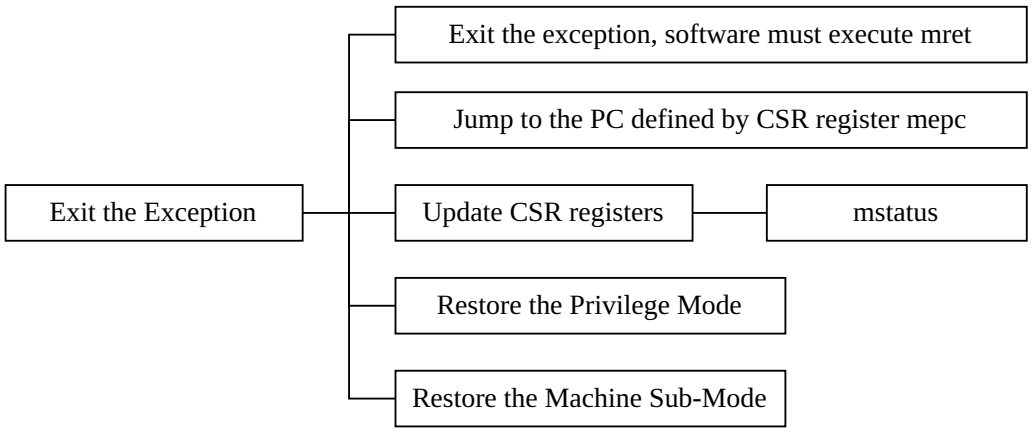


Fig. 7.3: The overall process of exiting an exception

7.5.1 Update the Machine Sub-Mode

The value of msubm.TYP indicates the Machine Sub-Mode of the Nuclei processor core in real time. After executing the mret instruction, the hardware will automatically restore the core's Machine Sub-Mode by the value of msubm.PTYP:

- Taking an exception, the value of msubm.PTYP is updated to the Machine Sub-Mode before taking the exception. After executing the mret instruction, the hardware will automatically restore the Machine Sub-Mode using the value of msubm.PTYP, as shown in *The CSR mstatus and msubm updating when enter/exit the exception* (page 17). Through this mechanism, the Machine Sub-Mode of the core is restored to the same mode before taking the exception.

7.6 Exception Service Routine

When the core takes one exception, it starts to execute the program starting at the address defined by mtvec, and this program is usually an exception service routine. The program can decide to jump further to the specified exception service routine by querying the exception code in the CSR mcause. For example, if the exception code in mcause is 0x2, which indicates that this exception is caused by an illegal instruction, then it can jump to the specific handler for illegal instruction fault.

Note: Since there is no hardware to save and restore the execution context automatically when take or exit an exception, so the software needs to explicitly use the instruction (in assembly language) for context saving and restoring.

---

8 NMI Handling in Nuclei processor core

8.1 NMI Overview

NMI (Non-Maskable Interrupt) is a special input signal of the processor core, often used to indicate system-level emergency errors (such as external hardware failures, etc.). After encountering the NMI, the processor should abort execution of the current program immediately and process the NMI error instead.

8.2 NMI Masking

In the RISC-V architecture, NMI cannot be masked, which means if the core encounters an NMI, it must stop current execution and turns to handle the NMI.

8.3 Entering NMI Handling Mode

Taking an NMI, hardware behaviors of the Nuclei processor core are described as shown in *The overall process of NMI* (page 20). Note that the following operations are done simultaneously in one cycle:

- Update the CSR registers: mepc and mstatus.
  - These behaviors are following RISC-V standard privileged architecture specification. This document will not repeat its content, please refer to RISC-V standard privileged architecture specification for more details.
- Update the CSR registers: mcause.
  - The value of mcause for NMI is unique in Nuclei processor core, and will be detailed in the following section.
- Stop the execution of the current program, and start from the PC address defined by the CSR mnvec.
  - The value of mnvec is unique in Nuclei processor core, and will be detailed in the following section.
- Update the Privilege Mode and Machine Sub-Mode of the core.
  - This is unique in Nuclei processor core, and will be detailed in the following section.

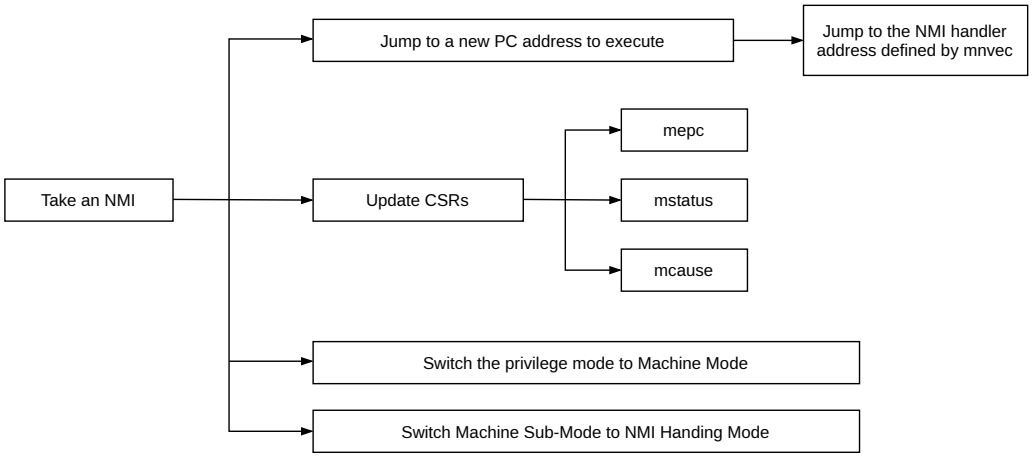


Fig. 8.1: The overall process of NMI

8.3.1 Execute from the PC Defined by mnvec

The Nuclei processor core jumps to the PC defined by the CSR mnvec after encountering an NMI. The CSR mnvec has two potential values controlled by CSR register mmisc_ctl:

- When mmisc_ctl[9]=1, the value of mnvec is equal to the value of mtvec, which means NMIs and exceptions share the same trap entry address.
- When mmisc_ctl[9]=0, the value of mnvec equals to the value of reset_vector which is the PC value after a reset. The reset_vector is the core's input signal. Please refer to the specific databook of the Nuclei processor core for details about this signal.

8.3.2 Update the CSR mcause

The Nuclei processor core will save the NMI code into the CSR mcause.EXCCODE by the hardware automatically when take a NMI. Interrupts, exceptions and NMIs all have their own specified Trap ID. The Trap ID of NMI has two potential values controlled by CSR register mmisc_ctl:

- When mmisc_ctl[9]=1,the Trap ID of NMI is 0xfff.
- When mmisc_ctl[9]=0,the Trap ID of NMI is 0x1.

The software can recognize the Trap reason querying the Trap ID, and build the corresponding trap handler program for different types of traps.

8.3.3 Update the Privilege Mode

NMI is handed in Machine Mode, so the privilege mode will be switched to Machine Mode when the core takes an NMI.

8.3.4 Update the Machine Sub-Mode

The Machine Sub-Mode of the Nuclei processor core is indicated in the msubm.TYP filed in real time. When the core takes an NMI, the Machine Sub-Mode will be updated to NMI handling mode, so:

- The filed msubm.TYP is updated to NMI handling mode, as described in *The CSR mstatus and msubm updating when enter/exit the NMI* (page 21), to reflect the current Machine Sub-Mode is "NMI handling mode".
- The value of msubm.PTYP will be updated to the value of msub.TYP before taking the NMI, as shown in *The CSR mstatus and msubm updating when enter/exit the NMI* (page 21). The value of msubm.PTYP will be used to restore the value of msubm.PTYP after exiting the NMI handler.

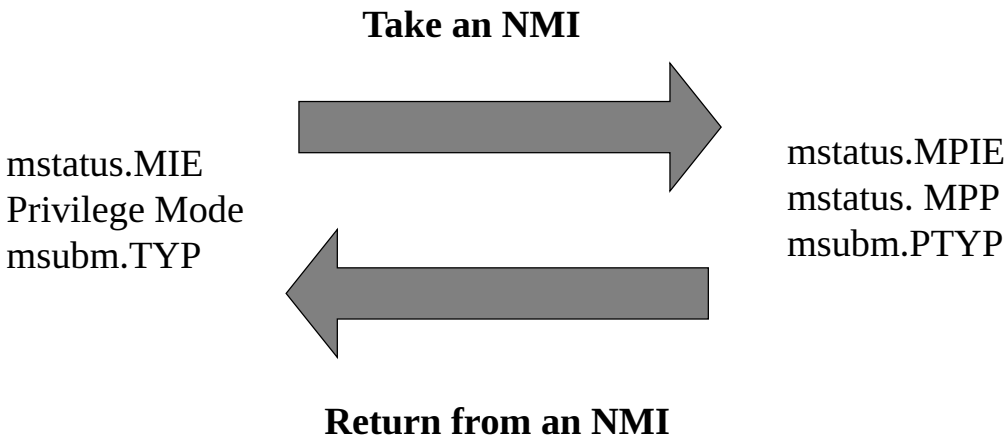


Fig. 8.2: The CSR mstatus and msubm updating when enter/exit the NMI

8.4 Exit the NMI Handling Mode

After handling the NMI, the core needs to exit from the NMI handler eventually, and return to execute the main program. Since the NMI is handling in Machine Mode, the software has to execute mret to exit the NMI handler.

The hardware behavior of the processor after executing mret instruction is as shown in *The overall process of exiting an NMI* (page 21). Note that the following hardware behaviors are done simultaneously in one cycle:

- Stop the execution of the current program, and start from the PC address defined by the CSR mepc. Update the CSR mstatus.Update the Privilege Mode.
  - These behaviors are following RISC-V standard privileged architecture specification. And the behaviors are exactly same as the behaviors "Exit the Exception Handling Mode", this document will not repeat its content here, please refer to RISC-V standard privileged architecture specification for more details.
- Update the Machine Sub-Mode.
  - This is unique in Nuclei processor core, and will be detailed in the following section.

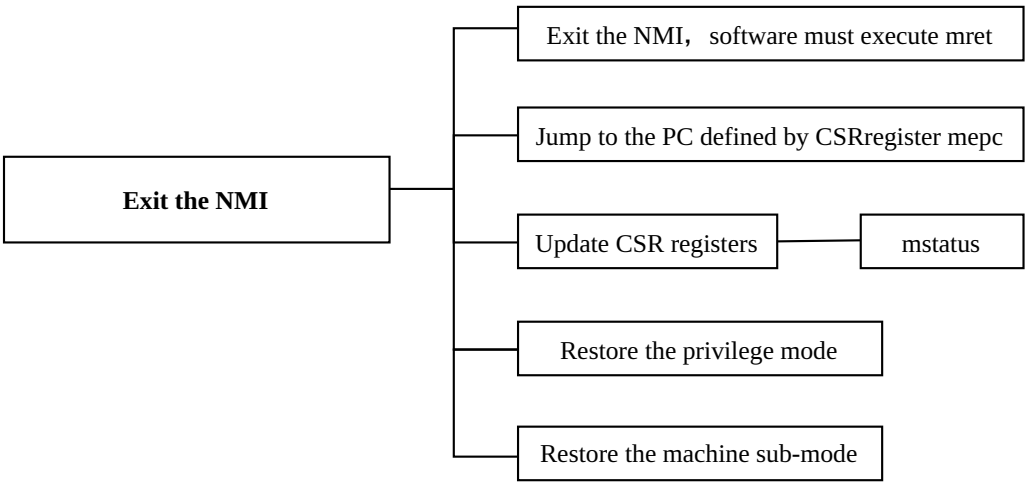


Fig. 8.3: The overall process of exiting an NMI

8.4.1 Update the Machine Sub-Mode

The value of msubm.TYP indicates the Machine Sub-Mode of the Nuclei processor core in real time. After executing the mret instruction, the hardware will automatically restore the core's Machine Sub-Mode by the value of msubm.PTYP:

- Taking an NMI, the value of msubm.PTYP is updated to the Machine Sub-Mode before taking the NMI. After executing the mret instruction, the hardware will automatically restore the Machine Sub-Mode using the value of msubm.PTYP, as shown in *The CSR mstatus and msubm updating when enter/exit the NMI* (page 21). Through this mechanism, the Machine Sub-Mode of the core is restored to the same mode before taking the NMI.

8.5 NMI Service Routine

When the core takes an NMI, it will jump to execute the program at the address defined by mnvec, which is usually the NMI service routine.

Note: Since there is no hardware to save and restore the execution context automatically when take or exit an NMI, so the software needs to explicitly use the instruction (in assembly language) for context saving and restoring.

9 Interrupt Handling in Nuclei processor core

9.1 Interrupt Overview

Interrupt, that is, the core is suddenly interrupted by other requests during the execution of the current program, and the current program is stopped, and then the core turns to handle other requests. After handling other requests, the core goes back and continues to execute the previous program.

The key points of interrupts are the followings:

- The "other request" interrupts the processor core is called Interrupt Request. The source of this request is called the Interrupt Source. The interrupt source is usually comes from outside the core which is called the External Interrupt Source, but some of the interrupt sources are core-internal, which are called the Internal Interrupt Sources.
- The program used to handle the "other request" is called the Interrupt Service Routine (ISR).
- Interrupt mechanism is a normal mechanism, not an error situation. Once the core receives an interrupt request, it needs to save the context of the current execution status, which is referred as "context saving". After processing the request, the core needs to restore the previous status, referred to "context restoring", thereby continuing to execute the previously interrupted program.
- There may be multiple interrupt sources that simultaneously initiate requests to the core, and an arbitration is needed to select one from these sources to determine which interrupt source is prioritized. This scenario is called "interrupt arbitration", and different interrupts can be assigned with different levels and priorities to facilitate the arbitration, so there is a concept of "interrupt level" and "interrupt priority".

9.2 CLIC mode and CLINT mode

9.2.1 Setting CLINT or CLIC mode

The Nuclei processor core supports the "CLINT interrupt mode (CLINT mode in short)" and "CLIC interrupt mode (CLIC mode in short)". Software can set the different mode by writing least significant bits of mtvec, please refer to *mtvec* (page 112) for more details.

Please refer to following sections for the recommendations of when to set the CLIC mode or CLINT mode.

9.2.2 CLINT mode

CLINT mode is the default mode after reset, it is a simple interrupt handling scheme.

The CLINT mode relying on the PLIC (Platform Level Interrupt Controller) in conjunction with the CSR register mie and mip, which is part of RISC-V standard privileged architecture specification. Please refer to RISC-V standard privileged architecture specification for more details.

The CLINT mode is recommended to be used in Linux capable applications or symmetric multi-processor (SMP) applications, please refer to *PLIC Unit Introduction* (page 44) for more details.

9.2.3 CLIC mode

CLIC mode is not the default mode after reset, hence need software to explicitly turn it on.

CLIC mode is a relevantly complicate interrupt handling scheme. The CLIC mode relying on the ECLIC (Enhanced Core Local Interrupt Controller), but the CSR register mie and mip are functionally bypassed in this mode.

The CLIC mode is recommended to be used in real-time or microcontroller applications, please refer to *ECLIC Unit Introduction* (page 46) for more details.

9.3 Interrupt Type

The types of interrupts supported by the Nuclei processor core are shown in *Interrupt Types* (page 24).

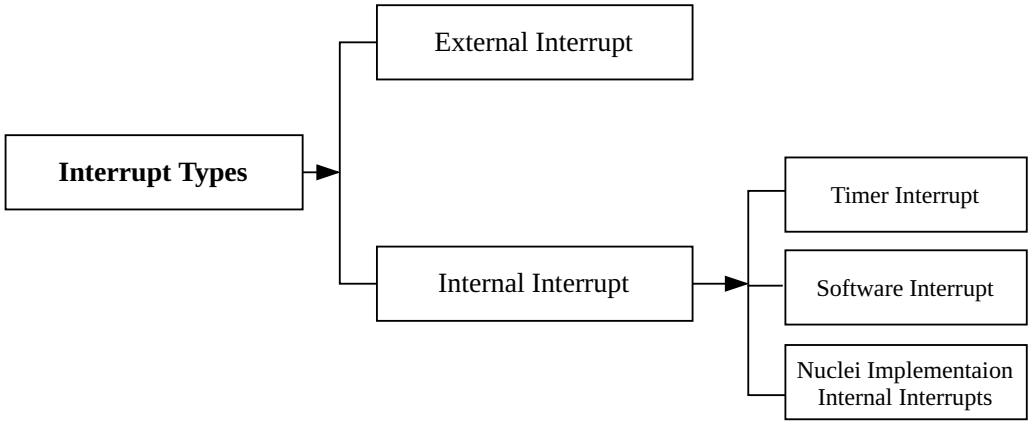


Fig. 9.1: Interrupt Types

These will be detailed in the following sections.

9.3.1 External Interrupt

An external interrupt is an interrupt initiated from outside the core. External interrupts allow user to connect to an external interrupt source, such as an interrupt generated by an external device like UART, GPIO and so on.

The Nuclei processor core supports multiple external interrupt sources.

Note:

- In CLINT mode, all of external interrupts are managed by the PLIC, as depicted in *Single-core with PLIC/ECLIC configured and PLIC enabled* (page 61).
- In CLIC mode, all of external interrupts are managed by the ECLIC, as depicted in *Single-core with PLIC/ECLIC configured and ECLIC enabled* (page 62).

9.3.2 Internal Interrupt

The Nuclei processor core has several core-internal private interrupts as the followings:

- Software Interrupt
  - The Nuclei processor core implements a TIMER unit, and msip/ssip register is defined in the TIMER unit, through which machine/supervisor software interrupts can be generated. Please see *Generating the Software Interrupt through msip/ssip* (page 43) for details.
- Timer Interrupt
  - The Nuclei processor core implements a TIMER unit, and a counter is defined in the TIMER unit, through which machine time interrupts can be generated. Please see *Generate the Machine Timer Interrupt through mtime and mtimecmp* (page 42) for details. For Supervisor timer interrupt, please refer <RISC-V SSTC> for more details.
- Nuclei Implementation Internal Interrupts
  - Some Nuclei processor cores implement more Internal Interrupts, the later sections of this document show more details.

Note:

- In CLINT mode, the internal interrupts of the Nuclei processor core are managed by CSR register mie and mip, as depicted in *mie* (page 111) and *mip* (page 111).
- In CLIC mode, the internal interrupts of the Nuclei processor core are also managed by the ECLIC, as depicted in *Single-core with PLIC/ECLIC configured and ECLIC enabled* (page 62).

9.4 Interrupt Masking

9.4.1 Global Interrupt Masking

Interrupts in machine mode can be masked globally by the control bit of CSR mstatus.MIE in Nuclei processor core. Please refer to RISC-V standard privileged architecture specification for more details.

9.4.2 Individual Interrupt Masking

It can also be masked individually for different interrupt sources:

- In CLINT mode:
  - In machine mode, the CSR register mie.MSIE/MTIE can be used to disable software interrupt, timer interrupt individually respectively. If there are other implementation internal interrupts, the CSR mie related bits will also be implemented. The mie.MEIE can be used to disable all the external interrupts managed by PLIC. Please refer to RISC-V standard privileged architecture specification for more details.
  - And PLIC unit also have memory mapped registers to enable/disable each interrupt source managed by PLIC. Please see *PLIC Registers* (page 44) for details.
- In CLIC mode, ECLIC have memory mapped register to enable/disable each interrupt source managed by ECLIC. Users can program the corresponding ECLIC register to manage some specified interrupt sources. Please see *ECLIC Registers* (page 49) for details.

9.5 Interrupt Levels, Priorities and Arbitration

When multiple interrupts are initiated at the same time, the arbitration is required:

- In CLINT mode:
  - The PLIC manages all external interrupts. PLIC assigns its own interrupt priority registers to each external interrupt source. Users can program the PLIC registers to manage the priority of the specified interrupt sources. When multiple interrupts occur simultaneously, the PLIC will select the one that has the highest priority and sent interrupt request to the core (as meip). Please see *PLIC Registers* (page 44) for details.
- In CLIC mode:
  - The ECLIC manages all interrupts. ECLIC assigns its own interrupt level and priority registers to each interrupt source. Users can program the ECLIC registers to manage the level and priority of the specified interrupt sources. When multiple interrupts occur simultaneously, the ECLIC will select the one that has the highest level/priority to be taken. Please see *ECLIC Registers* (page 49) for more details.

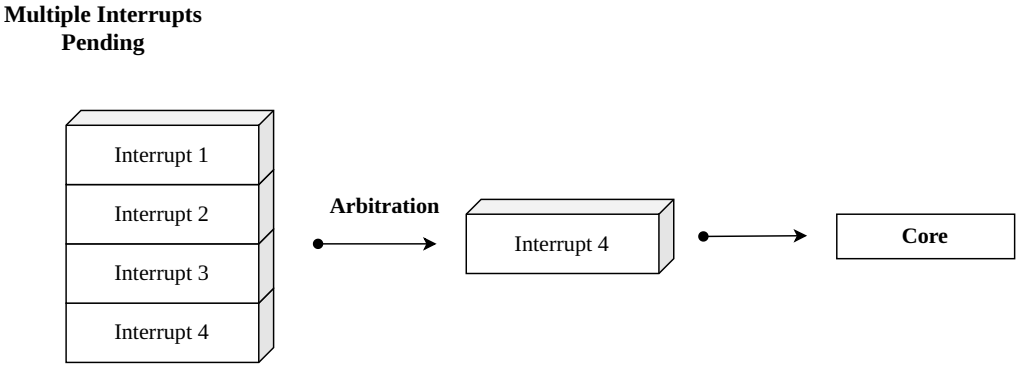


Fig. 9.2: Arbitration among Multiple Interrupts

9.6 (CLIC mode) Entering Interrupt Handling Mode

If it is in CLINT mode, taking an interrupt, hardware behaviors of the Nuclei processor core are following RISC-V standard privileged architecture specification. This document will not repeat its content, please refer to RISC-V standard privileged architecture specification for more details.

If it is in CLIC mode, taking an interrupt, hardware behaviors of the Nuclei processor core are described as below. Note that the following operations are done simultaneously in one cycle:

- Stop the execution of the current program, and jump to another PC to execute.
  - Update the following CSR registers:
    - mcause
    - mepc
    - mstatus
    - mintstatus
- Update the Privilege Mode and Machine Sub-Mode of the core.
- The overall process of interrupt is shown in *The Overall Process of Interrupt for CLIC mode* (page 27).

These will be detailed in the following sections.

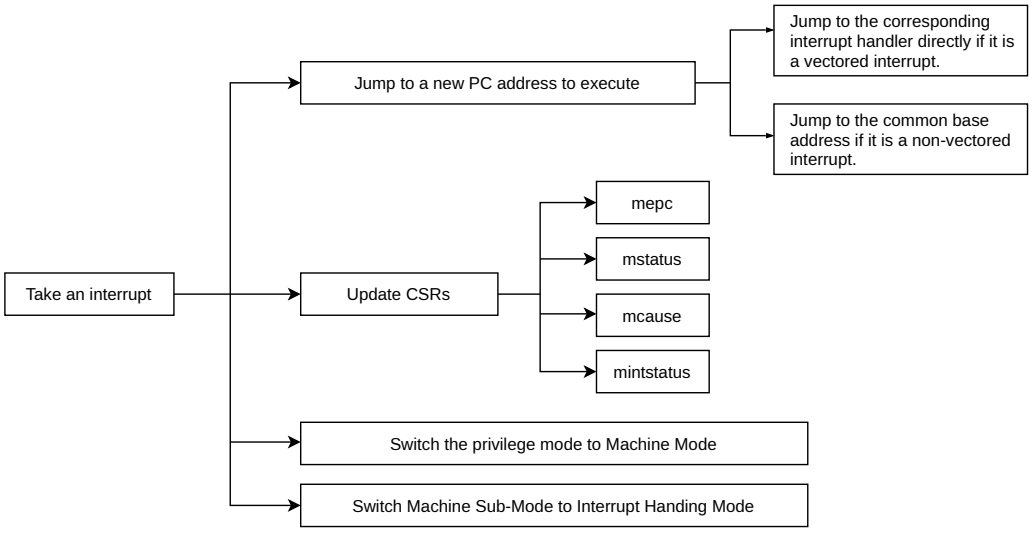


Fig. 9.3: The Overall Process of Interrupt for CLIC mode

9.6.1 Execute from a new PC

In CLIC mode, each interrupt source of the ECLIC can be set to vectored or non-vectored interrupt (via the shv filed of the register clicintattr[i]). The key points are as follows:

- If the interrupt is configured as a vectored interrupt, then the core will jump to the corresponding target address of this interrupt in the Vector Table Entry when this interrupt is taken. For details about the Interrupt Vector Table, please refer to *(CLIC mode) Interrupt Vector Table* (page 31). For details of the vectored processing mode, please refer to *Vectored Processing Mode* (page 37).
- If the interrupt is configured as a non-vectored interrupt, then the core will jump to a common base address shared by all interrupts. For details of the non-vectored processing mode, please refer to *Non-Vectored Processing Mode* (page 34).

9.6.2 Update the Privilege Mode

The privilege mode will be switched to Machine Mode when the core takes an Interrupt.

9.6.3 Update the Machine Sub-Mode

The Machine Sub-Mode of the Nuclei processor core is indicated in the msubm.TYP filed in real time. When the core takes an interrupt, the Machine Sub-Mode will be updated to interrupt handling mode, so:

- The value of msubm.PTYP will be updated to the value of msub.TYP before taking the interrupt as shown in Figure 6-4. The value of msubm.PTYP will be used to restore the value of msub.TYP after exiting the interrupt handler.
- The filed msubm.TYP is updated to interrupt handling mode, as described in Figure 6-4, to reflect the current Machine Sub-Mode is “interrupt handling mode”.

9.6.4 Update the CSR mepc

The return address when the Nuclei processor core exits the interrupt handler is stored in the CSR mepc. When the core takes an interrupt, the hardware will update the CSR mepc automatically, and the value in this CSR will be the return address when exit the interrupt handler. After handling the interrupt, the PC value is restored from this CSR mepc to return to the execution point that was previously stopped.

Note: When an interrupt is taken, the CSR mepc is updated to the PC of the instruction that encounters the interrupt. Then after exiting the interrupt, the program will continue to execute from the instruction that encounters the interrupt.

Although the CSR mepc can be updated automatically encountering an interrupt, it is a both readable and writeable register, so the software can modify it explicitly.

9.6.5 Update the CSRs mcause/mstatus/mintstatus

The Nuclei processor core will update the CSR mcause by the hardware automatically, as described in *The CSR updating when enter/exit the Interrupt* (page 29), explained as follows:

- A mechanism is required to record the ID of the interrupt being taken.
  - When an interrupt is taken by the Nuclei processor core, the field mcause.EXCCODE is updated to the ID of the taken interrupt by the ECLIC, so the software can query the ID of this selected interrupt by reading CSR mcause.
- When the current interrupt is taken, a mechanism is required to record the global interrupt enable bit and the Privilege Mode before taking the interrupt.
  - When the Nuclei processor core takes an interrupt, the filed mstatus.MPIE will be updated to the value of mstatus.MIE, and the filed mstatus.MIE will be set to 0, which means interrupts are globally masked, and all interrupts will not be taken.
  - When the Nuclei processor core takes a M-mode interrupt, the Privilege Mode of the core will be switched to Machine Mode, and the field mstatus.MPP will be set to the Privilege Mode before taking the interrupt.
- When the current interrupt is taken, possibly it is preempting the interrupt who was previously being processed (whose interrupt level is relatively lower, so it can be preempted), and a mechanism is needed to record the interrupt level of the preempted interrupt.
  - When an interrupt is taken by the Nuclei processor core, the field mcause.MPIL is updated to the value of minstatus.MIL, and the mintstatus.MIL records current interrupt's level vaule. The value of mcause.MPIL is used to restore the value of mintstatus.MIL after handling the interrupt.
- If the taken interrupt is a vectored interrupt, the core will jump to the corresponding target address stored in the Vector Table Entry. For a detailed description of the vectored interrupt processing mode, please see *Vectored Processing Mode* (page 37). In terms of the hardware implementation, the processing of an interrupt needs to be divided into two steps. The first step is to query the target address from the Vector Table, and then jump to the target address in the second step. Then, it is possible that a memory access occurs in the first step, querying the target address from the Vector Table, so a mechanism is required to record such a special memory access error.
  - When the Nuclei processor core takes an interrupt, if the interrupt is a vectored mode interrupt, the value of mcause.minhv will be updated to 1, and then cleared to 0 when the above "two-step" operation is completed. Assuming a memory access error occurs midway, it will raise an Instruction Access Fault exception, and the value of mcause.minhv will be 1 assuming this bit is not cleared.

Note: The mcause.MPIE and mcause.MPP are mirrored with the field of mstatus.MPIE and mstatus.MPP. Which means normally the value of mstatus.MPIE is always the same as the value of mcause.MPIE and the value of mstatus.MPP is the same as the value of mcasue.MPP.

If the value mintstatus.MIL is not 0, it means that core is handing an interrupt which level is MIL. If new interrupt is comming from ECLIC, the level of new interrupt should be greater than mintstatus.MIL so the new interrupt can preempt current interrupt, or it has waiting current interrupt finishing (mintstatus.MIL will be updated when an interrupt finishes in mret mechanism).

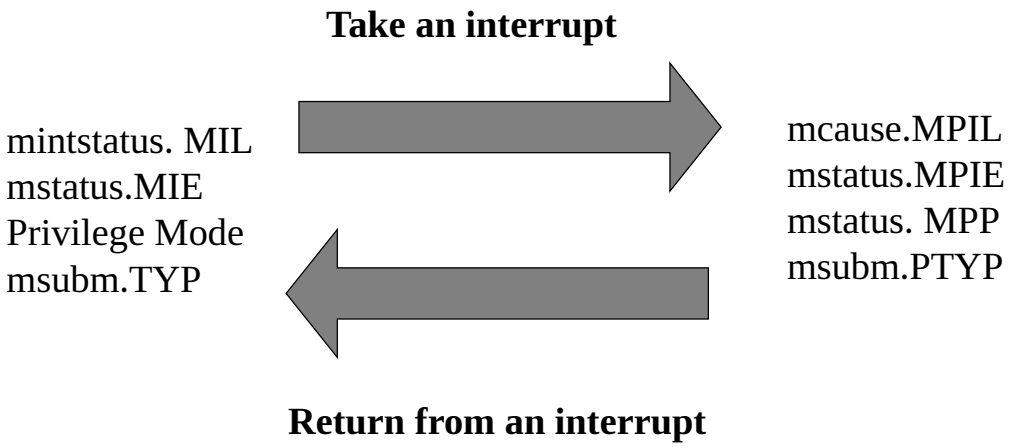


Fig. 9.4: The CSR updating when enter/exit the Interrupt

9.7 (CLIC mode) Exit the Interrupt Handling Mode

If it is in CLINT mode, after handling the interrupt, hardware behaviors of the Nuclei processor core are following RISC-V standard privileged architecture specification. This document will not repeat its content, please refer to RISC-V standard privileged architecture specification for more details.

If it is in CLIC mode, after handling the interrupt, the core needs to exit from the interrupt handler eventually, and return to execute the main program. Since the interrupt is handling in Machine Mode, the software must execute mret to exit the interrupt handler. The hardware behavior of the processor after executing mret instruction is as depicted in *The overall process of exiting an interrupt* (page 29). Note that the following hardware behaviors are done simultaneously in one cycle:

- Stop the execution of the current program, and start from the PC address defined by the CSR mepc.
- Update the following CSRs:
  - mstatus
  - mcause
  - mintstatus
- Update the Privilege Mode and the Machine Sub-Mode.

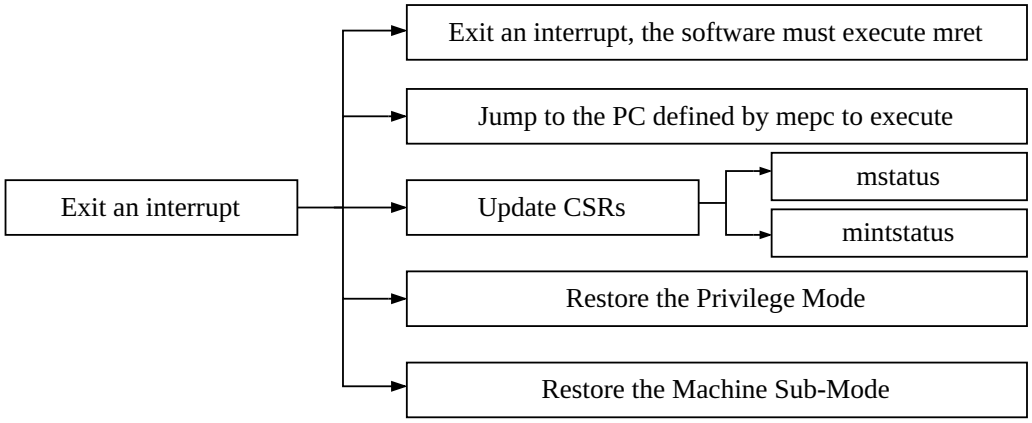


Fig. 9.5: The overall process of exiting an interrupt

These will be detailed in the following sections.

9.7.1 Executing from the Address Defined by mepc

When an interrupt is taking, the mepc is updated to the PC value of the instruction encountered the interrupt. Through this mechanism, executing the mret instruction, the core will return to the instruction encountered the interrupt, and continues to execute the program.

9.7.2 Update the CSRs mcause and mstatus

The Nuclei processor core will update the CSR mcause when executes one mret instruction, explained as follows:

- When an interrupt is taken, the value of mcause.MPIL will be updated to the value of mintstatus.MIL before taking the interrupt, while the mintstatus.MIL records current interrupt's level vaule. The hardware will restore the value of minstatus.MIL using the value of mcause.MPIL when executes the mret instruction to exit the interrupt handler. Through this mechanism, the value of mintstatus.MIL is restored to the previous value before taking the interrupt.
- When an interrupt is taken, the value of mcause.MPIE will be updated to the value of mstatus.MIE before taking the interrupt. The hardware will restore the value of mstatus.MIE using the value of mcause.MPIE when executes the mret instruction to exit the interrupt handler. Through this mechanism, the value of mstatus.MIE is restored to the previous value before taking the interrupt.
- When an interrupt is taken, the value of mcause.MPP will be updated to the Privilege Mode before taking the interrupt. The hardware will restore the Privilege Mode using the value of mcause.MPP when executes the mret instruction to exit the interrupt handler. Through this mechanism, the Privilege Mode is restored to the previous value before taking the interrupt.

Note: The mcause.MPIE and mcause.MPP are mirrored with the field of mstatus.MPIE and mstatus.MPP. Which means normally the value of mstatus.MPIE is always the same as the value of mcause.MPIE and the value of mstatus.MPP is the same as the value of mcasue.MPP.

9.7.3 Update the Privilege Mode

The hardware will update the Privilege Mode using the value of mcause.MPP automatically after the execution of the mret instruction:

- Taking an interrupt, the value of mstatus.MPP was updated to the Privilege Mode of the core before taking the interrupt, and after executing the mret instruction, the value of Privilege Mode is restored by the value of mstatus.MPP. Through this mechanism, the core is guaranteed to return to the Privilege Mode before taking the interrupt.

9.7.4 Update the Machine Sub-Mode

The value of msubm.TYP indicates the Machine Sub-Mode of the Nuclei processor core in real time. After executing the mret instruction, the hardware will automatically restore the core's Machine Sub-Mode by the value of msubm.PTYP:

- Taking an interrupt, the value of msubm.PTYP is updated to the Machine Sub-Mode before taking the interrupt. After executing the mret instruction, the hardware will automatically restore the Machine Sub-Mode using the value of msubm.PTYP. Through this mechanism, the Machine Sub-Mode of the core is restored to the same mode before taking the interrupt.

9.8 (CLIC mode) Interrupt Vector Table

If in CLINT mode, Nuclei processor core does not support the vector mode. Hence, there is no vector table relevant. Herein this section only introduces the CLIC mode interrupt vector table.

If in CLIC mode, as shown in *Interrupt Vector Table* (page 31), the interrupt vector table is a contiguous address space in the memory, and each word of this address space is used to store the address of the interrupt service routine corresponding to each interrupt source of the ECLIC.

The base address of the interrupt vector table is defined by the CSR mtvt.

The role of the interrupt vector table is very important. When the core takes an interrupt, no matter a vectored or non-vectored interrupt, the hardware will eventually jump to the corresponding PC of the interrupt service routine by querying the interrupt vector table. Please see *(CLIC mode) Vectored and Non-Vectored Processing Mode of Interrupts* (page 33) for more details.

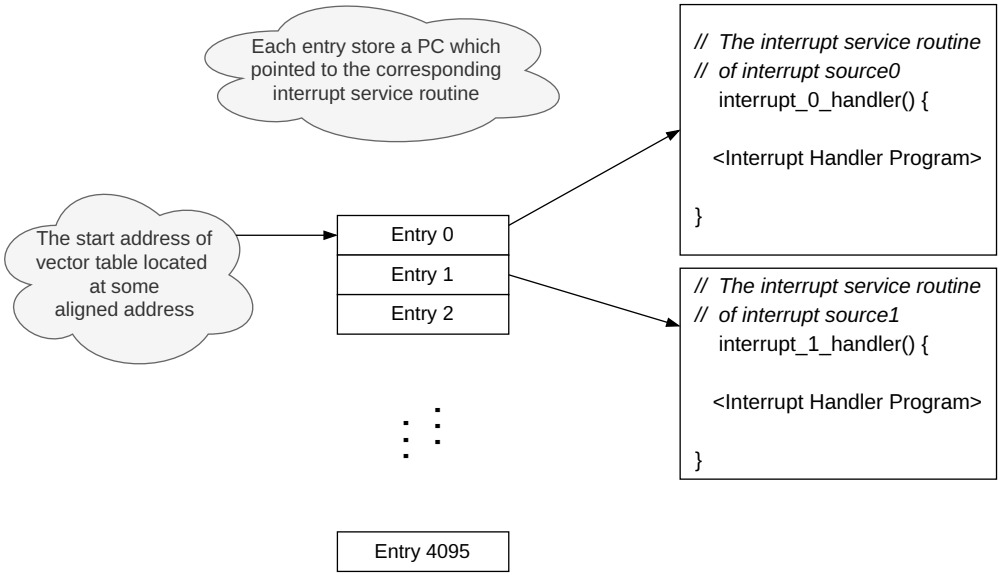


Fig. 9.6: Interrupt Vector Table

9.9 Context Saving and Restoring

Nuclei processor core based on the RISC-V architecture do not support the hardware automatic context saving and restoring when take or exit an interrupt. So the software is required to write the instructions (in assembly language) for context saving and restoring.

For CLIC mode, depending on whether the interrupt is a vectored or non-vectored, the context requiring saving and restoring will vary. Please see *(CLIC mode) Vectored and Non-Vectored Processing Mode of Interrupts* (page 33) for more details.

9.10 Interrupt Response Latency

The concept of interrupt response latency usually refers to the cycle consumed from the time point "external interrupt source asserting" to the time point "the first instruction in the corresponding interrupt service routine of C function is executed". Therefore, the interrupt latency usually includes the following aspects of the cycle overhead:

- The overhead of jumping to the target PC
- The overhead of context saving
- The overhead of jumping to the Interrupt Service Routine of C function

For CLIC mode, interrupt response latency varies depending on whether the interrupt is a vectored or non-vectored. Please see *(CLIC mode) Vectored and Non-Vectored Processing Mode of Interrupts* (page 33) for more details.

9.11 (CLIC mode) Interrupt Preemption

If in CLINT mode, Nuclei processor core does not support the interrupt preemption. Herein this section only introduces the CLIC mode interrupt preemption.

If in CLIC mode, while the core is handling an interrupt, there may be another new interrupt request of a higher level, and then the core can stop the current interrupt service routine and start to taken the new one and execute its "Interrupt Service Routine". Hence, the interrupt preemption is formed (that is, the previous interrupt has not returned yet, and the new interrupt is taken), and there could be multi-level of nesting.

Take the case in *Interrupt Preemption* (page 32) as an example:

- Assuming that the core is handling one timer interrupt and suddenly an interrupt is initiated by button 1 and this interrupt has a higher level than the timer interrupt. The core will stop processing the timer interrupt and start to handle the interrupt initiated by button 1.
- Then another interrupt is initiated by button 2, which has a higher level than the interrupt initiated by button 1, so the core will stop processing the interrupt of button 1 and start to handle the interrupt of button 2.
- After that no other higher-level interrupts arrive, the button 2 interrupt will not be preempted, and the core can successfully complete the interrupt service routine of the button 2 interrupt, and then return to process the button 1 interrupt.
- Completing the interrupt service routine of button 1 interrupt, the core will return to execute the timer interrupt service routine to handle the timer interrupt.

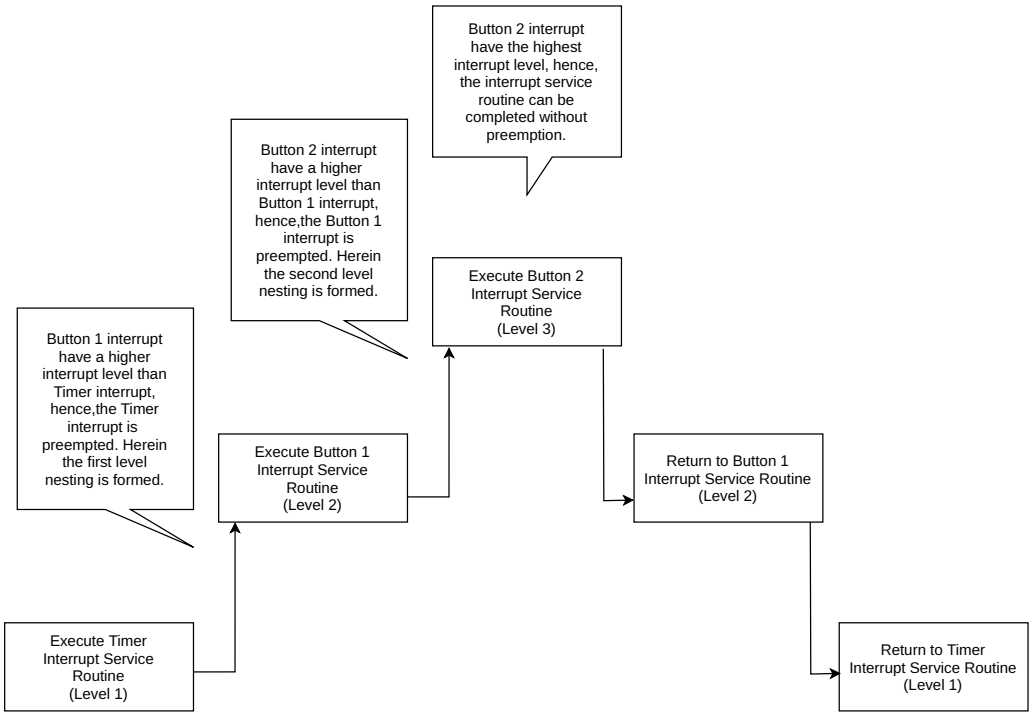


Fig. 9.7: Interrupt Preemption

In the Nuclei processor core, the supported methods for interrupt preemption depending on whether the interrupt is a vectored interrupt or a non-vectored interrupt. Please see *(CLIC mode) Vectored and Non-Vectored Processing Mode of Interrupts* (page 33) for more details.

9.12 (CLIC mode) Interrupt Tail-Chaining

If in CLINT mode, Nuclei processor core does not support the interrupt tail-chaining. Herein this section only introduces the CLIC mode interrupt tail-chaining.

If in CLIC mode, while the core is processing one interrupt, a new interrupt request is initiated, but the level of the new request is not higher than the handling one, so the new interrupt request cannot preempt the handling one.

After handling the current interrupt, theoretically it is necessary to restore the context, then exit the interrupt service routine, return to the main program, and then take the new interrupt. To take the new interrupt, it is necessary to save the context again. Therefore, there is a back-to-back "context saving" and "context restoring". The "tail-chaining" can save the cost of this back-to-back "context saving" and "context-restoring", as shown in the *Interrupt tail-chaining* (page 33).

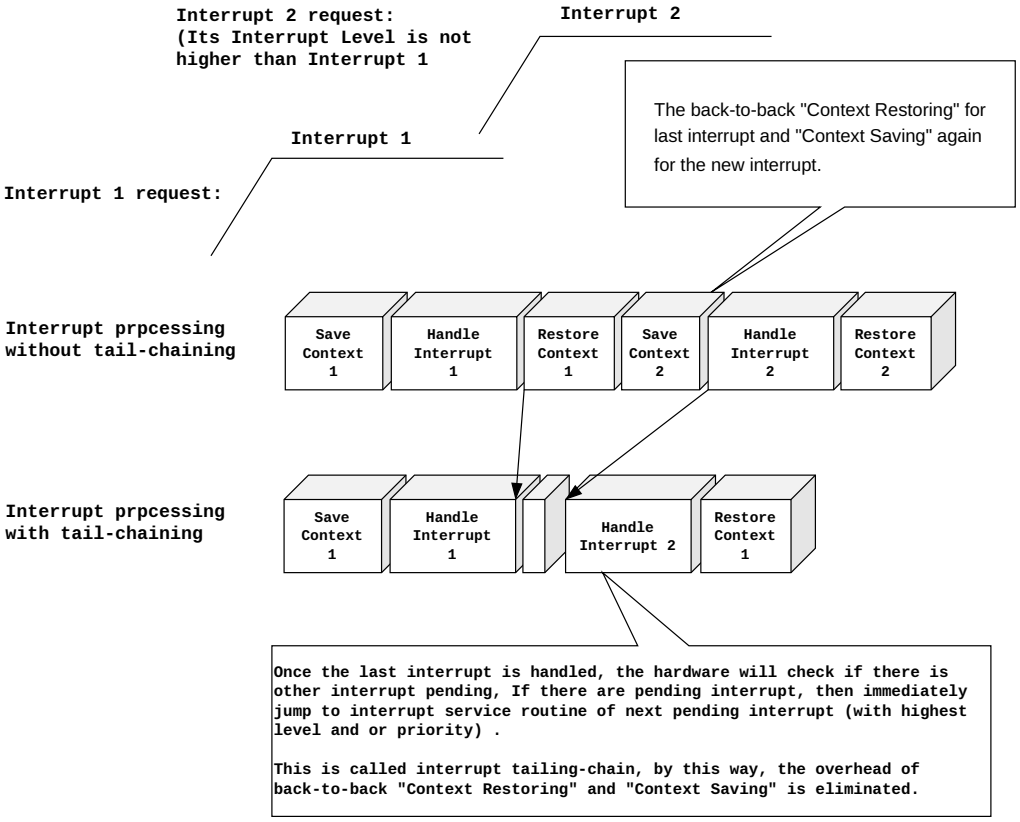


Fig. 9.8: Interrupt tail-chaining

As for the Nuclei processor core, only non-vectored interrupts (CLIC mode) support the feature of tail-chaining. Please see *Non-Vectored Interrupt Tail-Chaining* (page 36) for more details.

9.13 (CLIC mode) Vectored and Non-Vectored Processing Mode of Interrupts

In CLIC mode, each interrupt source can be configured to vectored or non-vectored processing mode (via the shv field of the ECLIC register clicintattr[i]). There is obvious difference between the vectored and non-vector processing mode, which are described in the following sections.

9.13.1 Non-Vectored Processing Mode

9.13.1.1 Feature and Latency of Non-Vectored Processing Mode

If the interrupt is non-vectored, once it is taken, the core will jump to the common base entry shared by all non-vectored interrupts, and the address of this entry can be set by software:

- If the least significant bit of the CSR mtvt2 is 0 (default value after reset), the common base address shared by all non-vectored interrupts is specified by the CSR mtvec (ignoring the value of the lowest 6 bits, please refer *mtvec* (page 112) for details). Since the CSR mtvec also indicates the entry address of exceptions, which means exceptions and all non-vector interrupts share the entry address.
- If the least significant bit of the CSR mtvt2 is 1, the common entry address of all non-vectored interrupts is defined by the CSR mtvt2 (ignoring the value of the lowest 2 bits). In order to handle the interrupt as fast as possible, it is recommended to set the least significant bit of the CSR mtvt2 to 1, which means the entry address for all non-vectored interrupts is separated from the entry of exceptions (exception entry is defined by the CSR mtvec).

After entering the common base entry of non-vectored interrupts, the core will start to execute a common program, as the example shown in *Example for non-vectored interrupt* (page 35), the program is typically as follows:

- Firstly, save the CSR mepc, mcause, msubm into the stack. These CSR registers are saved to ensure that subsequent preempted interruption can be handled correctly, because taken the new preempted interrupt will overwrite the values of mepc, mcause, msubm, so they need to be saved into the stack first.
- Save several general-purpose registers (the execution context) into the stack.
- Then execute a Nuclei self-defined instruction "csrrw ra, CSR_JALMNXTI, ra". If there is no pending interrupt, then this instruction will be regarded as a Nop. If there is a pending interrupt, the core will take the following operations:
  - Jump to the target address stored in Vector Table Entry and execute the corresponding Interrupt Service Routine.
  - The hardware will set the global interrupt enable bit mstatus.MIE while the core jumps to the interrupt service routine. Setting the mstatus.MIE bit, new interrupt will be taken and form an interrupt preemption.
  - In addition to jump to the Interrupt Service Routine, the instruction "csrrw ra, CSR_JALMNXTI, ra" also have the effect of a JAL (Jump and Link) instruction. The hardware will update the value of the link register to the PC of this instruction as the return address of the function. Therefore, returning from the interrupt handler, the core will return to the instruction "csrrw ra, CSR_JALMNXTI, ra", and re-judge whether there is still an interrupt pending to implement the operation of the tail-chaining. See more description of tail-chaining from *Non-Vectored Interrupt Tail-Chaining* (page 36).
- At the end of the interrupt service routine, the software also needs to add the corresponding context restoring operation. Before restoring the CSR mepc, mcause, msubm, and the global interrupt enable bit mstatus.MIE needs to be cleared again to ensure the atomicity of the recovery operations of mepc, mcause, and msubm.

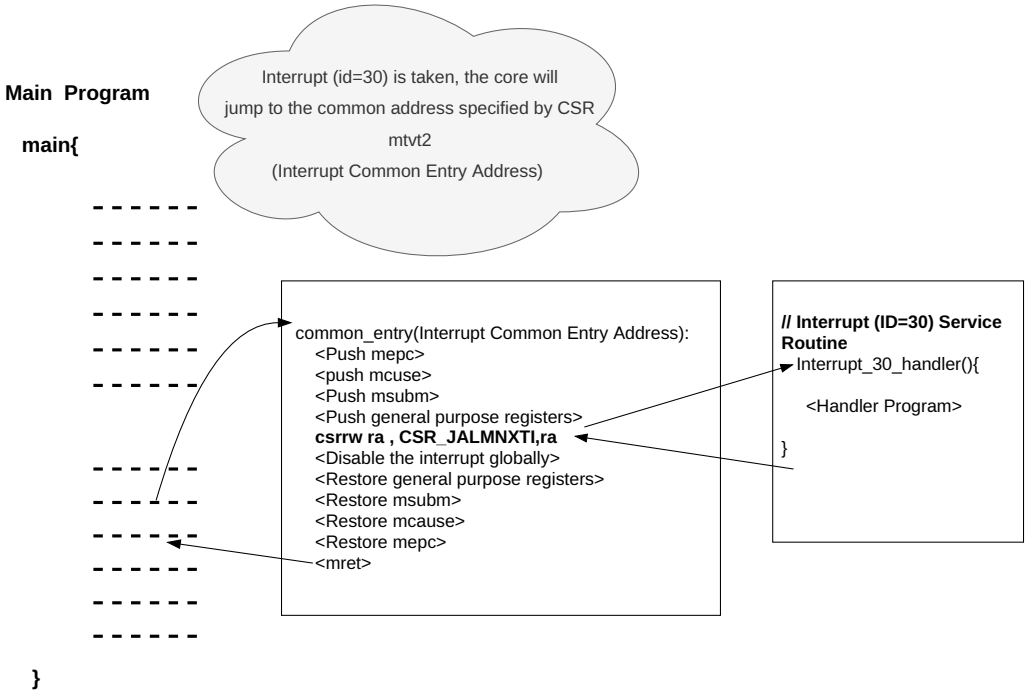


Fig. 9.9: Example for non-vectored interrupt

Since the core needs to execute a common handler before jump to the specified interrupt service routine of the corresponding non-vector interrupt. Therefore, the total cycle overhead from the interrupt initiation to the first instruction in the interrupt service routine (C function) is executed are as below:

- The overhead caused by jumping to the interrupt handler which is about 4 cycles ideally.
- The overhead caused by saving CSRs mepc, mcause, msubm into the stack is about 3 cycles ideally.
- The overhead caused by saving the context. If the architecture is RV32E, then it only takes 8 cycles to save 8 general purpose registers; if it is RV32I architecture, then there are 16 general purpose registers required to be saved.
- The overhead caused by jumping to the Interrupt Service Routine which is about 5 cycles ideally.

9.13.1.2 Preemption of Non-Vectored Interrupt

As mentioned above, non-vectored interrupt processing mode can always support interrupt preemption as the example shown in *Interrupt preemptions caused by three sequential non-vectored interrupts* (page 36): assuming that the three interrupts 30, 31, 32 come sequentially, and the level of interrupt 32 is greater than the level of interrupt 31 which is greater than the level of interrupt 30. Since then, the subsequent interrupts will preempt interrupts that were previously processed to form interrupt preemptions.

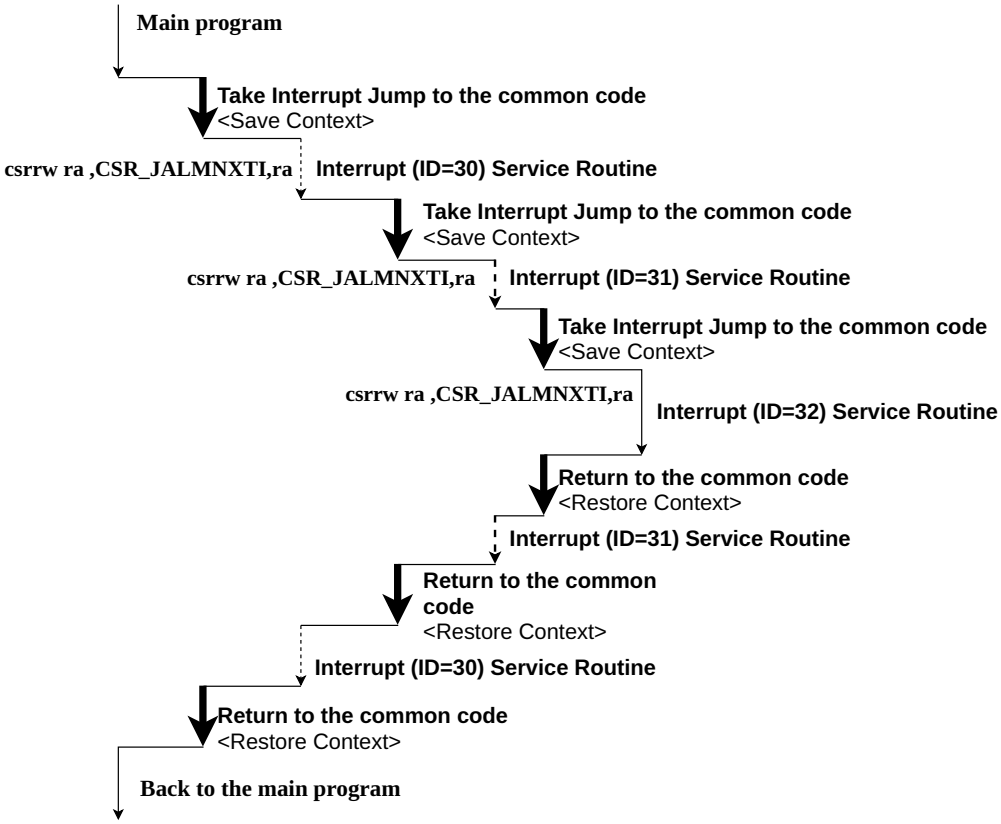


Fig. 9.10: Interrupt preemptions caused by three sequential non-vectored interrupts

9.13.1.3 Non-Vectored Interrupt Tail-Chaining

As mentioned in *(CLIC mode) Interrupt Tail-Chaining* (page 33), the tail-chaining can save cycles overhead significantly (reduced a back-to-back context saving and restoring).

For non-vectored interrupts (CLIC mode), as mentioned in *Feature and Latency of Non-Vectored Processing Mode* (page 34), the instruction "csrrw ra, CSR_JALMNXTI, ra" in the common base handler also achieves the effect of JAL (Jump and Link), which means the hardware will update the value of the Link register to the PC of this instruction as the return address. Therefore, the core will execute the instruction "csrrw ra, CSR_JALMNXTI, ra" again when it return from the interrupt service handler (C function) and re-execute "csrrw ra, CSR_JALMNXTI, ra", i.e., re-judge if there is a pending interrupt to perform the tail-chaining operation.

Note: At this time, the pending interrupt's level should be lower than MPIL of csr mstatus, or it will not be handled by Tail-Chaining.

As the example shown in *Interrupt tail-chaining* (page 37): assuming the interrupts 30, 29, 28 come successively, and "the level of interrupt 30 " >= "the level of interrupt 29" >= "the level of interrupt 28", then the subsequent interrupt will not preempt the interrupt that was taken before, which means no preemption will happen, but all these subsequent interrupt will be marked as "pending". When the interrupt 30 has been already handled, the core will handle the interrupt 29 directly without the intermediate "context restoring" and "context saving" procedures.

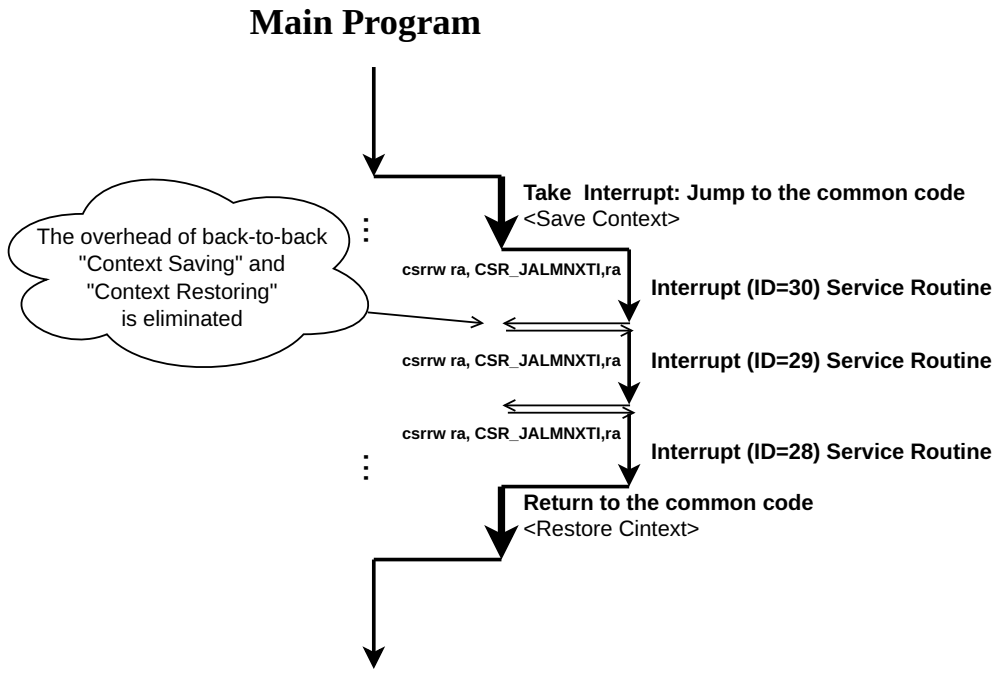


Fig. 9.11: Interrupt tail-chaining

9.13.2 Vectored Processing Mode

9.13.2.1 Feature and Latency of Vectored Processing Mode

If the interrupt is vectored, once it is taken, the core will jump to the target address saved in the Vector Table Entry directly, which is the corresponding interrupt service routine (C function) of the interrupt, as shown in *Example for vectored interrupt* (page 37).

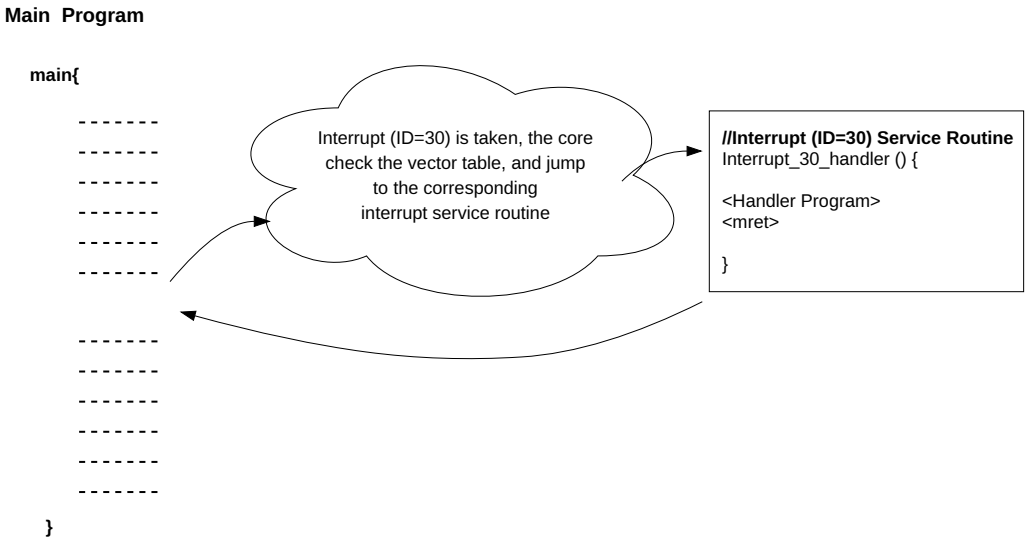


Fig. 9.12: Example for vectored interrupt

Vectored Processing Mode has the following features:

- The core will jump directly to the interrupt service routine without context saving. Therefore, the latency of the vectored interrupt is very short. Ideally, it only takes 6 cycles from the interrupt initiation to the execution of the first instruction of the interrupt service routine (C function), because the hardware only need to perform one lookup and jump.

- For an interrupt service routine of a vectored interrupt, the indication" __attribute__ ((interrupt))" is required to indicate compiler this C function is an interrupt service routine. Why this attribute is needed? Explained as below:
  - In the vector processing mode, since the core does not save the context before jumping to the interrupt service routine, theoretically the interrupt handler cannot call any sub-function which means the handler must be a leaf function.
  - If the interrupt service routine accidentally calls another sub-function, which means the routine is not a leaf function, it will cause a function error because of context corruption.
  - In order to avoid this accidental error, as long as the "__attribute__ ((interrupt))" is used to indicate this function is an interrupt handler, the compiler will automatically detect if this function calls any sub-function. If it calls any sub-function, the compiler will automatically insert a piece of code to save the context. Note: in this case, although the function correctness is guaranteed, the overhead caused by context saving will actually increase the latency of the response of the interrupt (equivalent to the non-vectored interrupt processing) and cause the expansion of the code size. Hence, in practice, it is not recommended to call other sub-functions in the interrupt service routine of a vectored interrupt.

9.13.2.2 Preemption of Vectored Interrupt

In vectored processing mode, the core does not perform any special operation before jumping to the interrupt service routine, and the value of mstatus.MIE is updated to 0 by the hardware, which means the interrupt is global disabled and no new interrupt will be taken once the core is handling the interrupt. Therefore, the vectored processing mode does not support interrupt preemption by default. In order to support vectored interrupt preemption, a special stack-push operation is necessary at the beginning of the interrupt service routine as shown in *Example for vectored interrupt supported preemption* (page 38):

- First save the CSRs mepc, mcause, msubm to the stack. These CSRs are saved to ensure that subsequent interrupt preemption can perform correctly, because the new taken interrupt will overwrite the values of mepc, mcause, and msubm, so they need to be saved to the stack first.
- Re-enable the global interrupt enable bit, that is, set the mstatus.MIE to 1. After the global interrupt enable bit is set, the new interrupt can be taken to allow the mechanism of interrupt preemption.
- At the end of the interrupt service routine, it is necessary to add the operation of context restoring. And before CSRs mepc, mcause, and msubm are restored from the stack, the global interrupt enable bit must be set as 0 to guarantee the atomicity of the restoring operation of CSRs mepc, mcause, and msubm (not interrupted by the new interrupt).

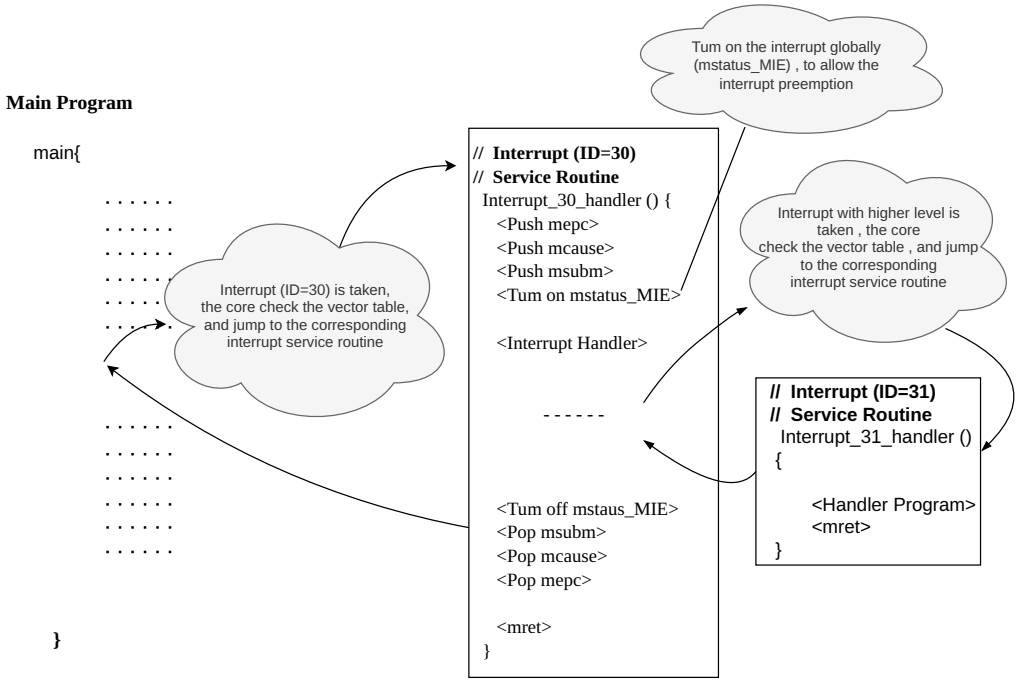


Fig. 9.13: Example for vectored interrupt supported preemption

As described above, with the special processing, the vectored processing mode can support interrupt preemption, as shown in *Interrupt preemptions caused by three sequential vectored interrupts* (page 39): assuming that the three interrupts 30, 31, 32 come sequentially, and the level of interrupt 32 is greater than the level of interrupt 31 which is greater than the level of interrupt 30. Since then, the subsequent interrupts will preempt interrupts that were previously processed to form interrupt preemptions.

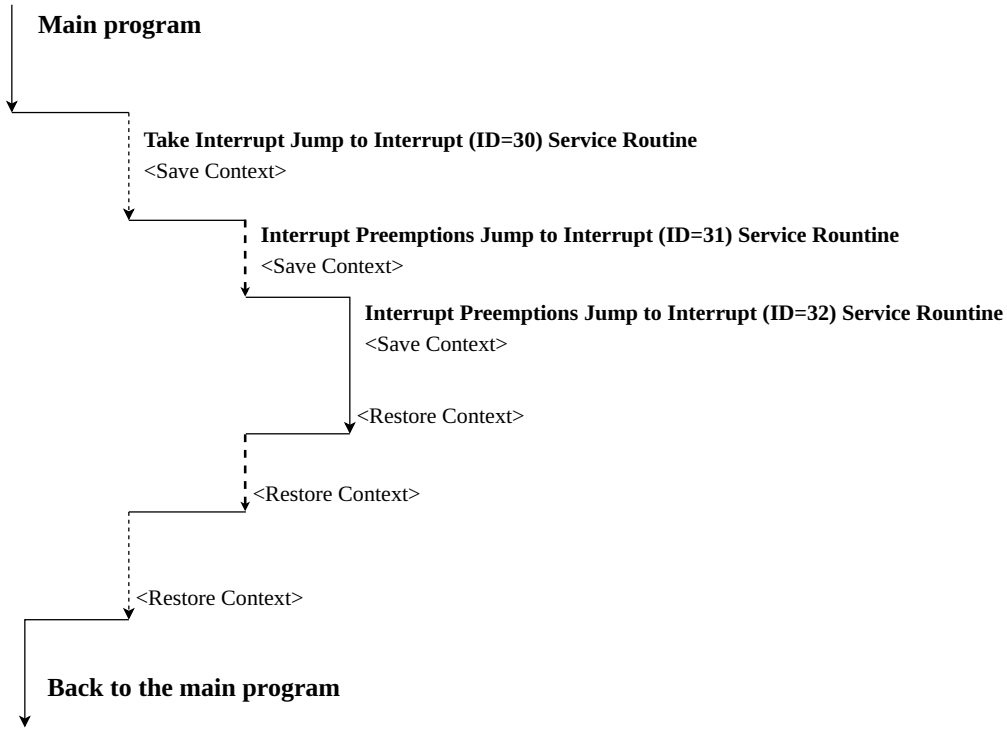


Fig. 9.14: Interrupt preemptions caused by three sequential vectored interrupts

9.13.2.3 Vectored Interrupt Tail-Chaining

For the vectored processing mode, the core does not save the context before jumping to the interrupt service routine, so the meaning of "interrupt tail-chaining" is not significant. Therefore, the vectored processing mode does not support the features of "interrupt tail-chaining".

10 TIMER Unit Introduction

10.1 TIMER Overview

The Timer Unit (TIMER for short) is used to generate the Timer Interrupt and Software Interrupt in Nuclei processor core.

10.2 TIMER Registers

The TIMER is a memory-mapped unit:

- For the base address of the TIMER unit, please refer to the specific databook of the Nuclei processor core.
- Registers and the corresponding offset in the TIMER unit are shown in *The addresses offset of registers in the TIMER unit* (page 40)

Table 10.1: The addresses offset of registers in the TIMER unit

Offset	Permission/Width	Register Name	Default Value	Function Description
0x0	MRW/4B	mtime_lo	0x00000000	Reflect the lower 32-bit value of mtime. Shadow copy of MTIME in CLINT mode
0x4	MRW/4B	mtime_hi	0x00000000	Reflect the upper 32-bit value of mtime. Shadow copy of MTIME in CLINT mode
0x8	MRW/4B	mtimecmp_lo	0xFFFFFFFF	Set the lower 32-bit value of mtimecmp. Shadow copy of MTIMECMP for 1st hart in CLINT mode
0xC	MRW/4B	mtimecmp_hi	0xFFFFFFFF	Set the upper 32-bit value of mtimecmp. Shadow copy of MTIMECMP for 1st hart in CLINT mode
0xFEC	MRW/4B	mtime_srw_ctrl	0x00000000	Control S-mode can access this timer or not.
0xFF0	MRW/4B	msftrst	0x00000000	Generate soft-reset request.
0xFF8	MRW/4B	mtimectl	0x00000000	Control some features of the time counter.
0xFFC	MRW/4B	msip	0x00000000	Generate the Software Interrupt. Shadow copy of MSIP for 1st hart in CLINT mode

continues on next page

Table 10.1 – continued from previous page

Offset	Permission/Width	Register Name	Default Value	Function Description
0x1000	MRW/4B	MSIP for Hart-0	0x00000000	Software Interrupt for 1st hart.
0x1004	MRW/4B	MSIP for Hart-1	0x00000000	Software Interrupt for 2nd hart.
0x1008	MRW/4B	MSIP for Hart-2	0x00000000	Software Interrupt for 3rd hart.
0x100c	MRW/4B	MSIP for Hart-3	0x00000000	Software Interrupt for 4th hart.
…	…	…	…	…
0x5000	MRW/8B	MTIMECMP for Hart-0	0x00000000	M-mode timer compare register for 1st hart.
0x5008	MRW/8B	MTIMECMP for Hart-1	0x00000000	M-mode timer compare register for 2nd hart.
0x5010	MRW/8B	MTIMECMP for Hart-2	0x00000000	M-mode timer compare register for 3rd hart.
0x5018	MRW/8B	MTIMECMP for Hart-3	0x00000000	M-mode timer compare register for 4th hart.
…	…	…	…	…
0xCFF8	MRW/8B	MTIME	0x00000000	MTIME in CLINT mode
0xD000	SRW/4B	SSIP for Hart-0	0x00000000	Supervisor Software Interrupt for 1st hart.
0xD004	SRW/4B	SSIP for Hart-1	0x00000000	Supervisor Software Interrupt for 2nd hart.
0xD008	SRW/4B	SSIP for Hart-2	0x00000000	Supervisor Software Interrupt for 3rd hart.
0xD00c	SRW/4B	SSIP for Hart-3	0x00000000	Supervisor Software Interrupt for 4th hart.
…	…	…	…	…

Note:

- Registers in the TIMER unit only support aligned read and write access with WORD size.
- In a design with multi cluster/core of Nuclei Core, each cluster/core has an independent input hartid signal, so the hartid of 1st hart of this cluster/core is (0+OFFSET), OFFSET means the input number.
- The hardware can guarantee that the registers can only be accessed under M-mode.or S-Mode

The functionality of each register is described in the following sections.

10.2.1 Control S-mode can access timer or not through mtime_srw_ctrl

The register mtime_srw_ctrl is implemented to control S-mode can access mtime or not , as shown in *mtime_srw_ctrl bit fields* (page 41).

Table 10.2: mtime_srw_ctrl bit fields

Field	Bits	Permission	Default Value	Description
Reserved	31:1	N/A	N/A	Reserved, ties to 0
SRW	0	RW	0	Control S-mode can read or write timer registers or not. 0: S-Mode can read/write all timer registers. 1: S-Mode can not read/write timer registers

10.2.2 Time Counter Register mtime

The key points of TIMER unit are as follows:

- The TIMER implements a 64-bit register mtime, which is composed of {mtime_hi, mtime_lo}. This register reflects the value of the 64-bit timer. The timer is turned on by default, so it will always count after reset.
- The increment frequency of the counter is controlled by the core's input signal mtime_toggle_a or core's always-on clock input core_aon_clk. Please refer to the specific databook of the Nuclei processor core for details about this signal.

10.2.3 Generate the Machine Timer Interrupt through mtime and mtimecmp

The TIMER unit can be used to generate the timer interrupt. The key points are as follows:

- The TIMER implements a 64-bit register mtimecmp, which is composed of {mtimecmp_hi, mtimecmp_lo}. This register is used as the comparison value of the timer. If the value of mtime is greater than the value of mtimecmp, then a timer interrupt is generated.
- If mtimectl.CMPCLREN is set as 1, then the mtime will be automatically cleared to zero when the value of mtime is greater than the value of mtimecmp, and then restart counting from zero.
- If mtimectl.CMPCLREN is set as 0, then the mtime will always increments normally. The software can clear the timer interrupt by overwriting the value of mtimecmp or mtime (so that the value of mtimecmp is greater than the value of mtime).

Note: The timer interrupt is connected to the ECLIC unit as unified interrupt management. Please see *ECLIC Unit Introduction* (page 46) for details of ECLIC.

For Supervisor timer interrupt, please refer <RISC-V SSTC> for more details.

10.2.4 Control the Timer Counter through mtimectl

The register mtimectl is implemented to control the behaviors of timer counting , as shown in *mtimectl bit fields* (page 42).

Table 10.3: mtimectl bit fields

Field	Bits	Permission	Default Value	Description
Reserved	31:4	N/A	N/A	Reserved, ties to 0
HDBG	3	RW	0	Halt-on-debug Controls whether core's stoptime bit in debug CSR *dcsr* stops the timer counter: If the files is 0, Timer counter ignores stoptime signal; if it is 1, assert stoptime in *dcsr* halts the timer counter when core is in debug mode. Note: this bit is implemented from 900 v3.8.2.
CLKSRC	2	RW	0	Select the source of increment frequency. If this field is 1, then the increment frequency is frequency of core_aon_clk, otherwise the increment frequency is controlled by mtime_toggle_a, please refer to the specific databook of the Nuclei processor core for details about this signal.
CMPCLREN	1	RW	0	Control the timer count to clear-to-zero or not. If this field is 1, then the mtime register will be cleared to zero after generated timer interrupt, otherwise it increments normally.
TIMESTOP	0	RW	0	Control the timer count or pause. If this field is 1, then the timer is paused, otherwise it increments normally.

Note: If CMPCLREN is enabled, the timer interrupt request will be a pulse request. In this case, timer interrupt should be set to edge-trigger mode.

10.2.5 Generating the Software Interrupt through msip/ssip

The TIMER unit can be used to generate the Software Interrupt. The register msip/ssip is implemented in the TIMER unit as shown in *msip/ssip bit fields* (page 43), only the least significant bit of msip is an effective bit. This bit is used to generate the software interrupt directly:

- The software generates the software interrupt by writing 1 to the msip/ssip register;
- The software clears the software interrupt by writing 0 to the msip/ssip register.

Note: the soft interrupt is connected to the ECLIC unit as unified interrupt management. Please see *ECLIC Unit Introduction* (page 46) for details of ECLIC.

Table 10.4: msip/ssip bit fields

Field	Bits	Permission	Default Value	Description
Reserved	31:1	N/A	N/A	Reserved, ties to 0
MSIP/SSIP	0	RW	0	This bit is used to generate the software interrupt

10.2.6 Generating the Soft-Reset Request

The TIMER unit can be used to generate the Soft-Reset request. The register msftrst is implemented in the TIMER unit as shown in *msftrst bit fields* (page 43), only the least significant bit of msftrst is an effective bit. This bit is used to generate the Soft-Reset request directly:

- The software generates the Soft-Reset request by writing 0x80000a5f to the msftrst register. Requiring to write such a complicate number is to avoid the random mis-operation of software writing.
- The most significant bit of msftrst can only be clear by reset,so if the SoC reset the core in respond to Soft-Reset request, then msftrst register will be reset (and cleared to zero).

Note: The core's output signal sysrstreq (active high) is used to carry out the Soft-Reset request, the SoC should reset the core (assert core_reset_n, not por_reset_n) in respond to the request. Please refer to the specific databook of the Nuclei processor core for details about the signals sysrstreq, core_reset_n and por_reset_n.

Table 10.5: msftrst bit fields

Field	Bits	Permission	Default Value	Description
MSFTRST	31	RW	0	This bit is used to generate the Soft-Reset Request
Reserved	30:0	N/A	N/A	Reserved, ties to 0

11 PLIC Unit Introduction

11.1 PLIC Overview

For the Linux capable applications or symmetric multi-processor (SMP) applications, Nuclei processor core have been equipped with a Platform-Level Interrupt Controller (PLIC), which is part of RISC-V standard privileged architecture specification (riscv-privileged-20240411.pdf).

User can easily get the documents from Nuclei released doc-package or https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc.

Note:

- The PLIC unit arbitrates the interrupt sources to the processor core (as the interrupt target) by a line as shown in *Interrupt Connection (for single-core with PLIC/ECLIC configured and PLIC enabled)* (page 61).
- The PLIC need to be enabled by setting the LSB bits of CSR registers mtvec as CLINT mode. Please refer to *Setting CLINT or CLIC mode* (page 23) for the details.
- The PLIC is functionally exclusive to ECLIC. The ECLIC and PLIC connection diagrams are as described in *ECLIC, PLIC and CIDU Connection Diagram* (page 60).

11.2 PLIC Registers

The RISC-V standard privileged architecture specification does not specify the exact register offset for PLIC. The Nuclei processor core implements PLIC as a memory-mapped unit:

- The base address of the PLIC unit, please refer to the specific databook of the Nuclei processor core.
- Registers and the corresponding offset in the PLIC unit are shown in *The addresses offset of registers in the PLIC unit* (page 44).

Table 11.1: The addresses offset of registers in the PLIC unit

OFFSET	WIDTH	PERMISSION	DESCRIPTION	DEFAULT VALUE
0x00_0000			Reserved (source 0 does not exist)	
0x00_0004	4B	SRW	Source 1 priority	0x0
0x00_0008	4B	SRW	Source 2 priority	0x0
……		……	……	……
0x00_0FFC	4B	SRW	Source 1023 priority	0x0
……		……	……	……
0x00_1000	4B	SR	Start of pending array (bit 0-31)	0x0
……		……	……	……

continues on next page

Table 11.1 – continued from previous page

OFFSET	WIDTH	PERMISSION	DESCRIPTION	DEFAULT VALUE
0x00_107C	4B	SR	Last word of pending array (bit 992-1023)	0x0
.....		.....	.....	.....
0x00_2000	4B	MRW	Start of Hart 0 M-mode interrupt enables (source 0-31)	0x0
.....		.....	.....	.....
0x00_207C	4B	MRW	Last word of Hart 0 M-mode interrupt enables (source 992-1023)	
0x00_2080	4B	SRW	Start of Hart 0 S-mode interrupt enables (source 0-31)	
.....		.....	.....	
0x00_20FC	4B	SRW	Last word of Hart 0 S-mode interrupt enables (source 992-1023)	
0x20_0000	4B	MRW	Hart 0 M-mode Priority threshold	0x0
0x20_0004	4B	MRW	Hart 0 M-mode Claim/Complete	0x0
.....			Reserved	
0x20_1000	4B	SRW	Hart 0 S-mode Priority threshold	0x0
0x20_1004	4B	SRW	Hart 0 S-mode Claim/Complete	0x0
.....		.....	.....	
0x3FF_FFFC	4B	MRW	S-mode Priority/Pending Access 0x0 means S-mode can access all priority/pending registers. 0x1 means S-mode can not access all priority/pending registers.	0x0

Note:

- PLIC registers only support aligned access which is the size of word.
- The above "R" means read-only, and any write to this read-only register will be ignored without generating bus error.
- The PLIC unit may not be configured to support 1023 interrupt sources. If an interrupt is not present in the hardware, the corresponding registers of memory locations appear hardwired to zero.
- The PLIC unit has M-mode and S-mode dedicated registers, which can be configured to trigger M-mode interrupt or S-mode interrupt, by default, both M-mode and S-mode interrupts are all handled in M-mode, when Mideleg is configured, S-mode interrupts will be delegated to be handled in S-mode, but the M-mode interrupts will still be handled in M-mode regardless of the Mideleg.
- The hardware can guarantee that the registers can only be accessed under the corresponding privilege mode.

12 ECLIC Unit Introduction

12.1 ECLIC Overview

For the real-time or microcontroller applications, the fast interrupt handling scheme is very important, hence, Nuclei processor core have been equipped with an Enhanced Core Local Interrupt Controller (ECLIC), which is optimized based on the RISC-V standard CLIC, to manage all interrupt sources.

Note:

- The ECLIC unit only serves one core and is private inside the core, as shown in *ECLIC Connection (when ECLIC is enabled)* (page 47).
- The ECLIC need to be enabled by setting the LSB bits of CSR registers mtvec as ECLIC mode. Please refer to *Setting CLINT or CLIC mode* (page 23) for the details.
- The ECLIC is functionally exclusive to PLIC. The ECLIC and PLIC connection diagrams are as described in *ECLIC, PLIC and CIDU Connection Diagram* (page 60).

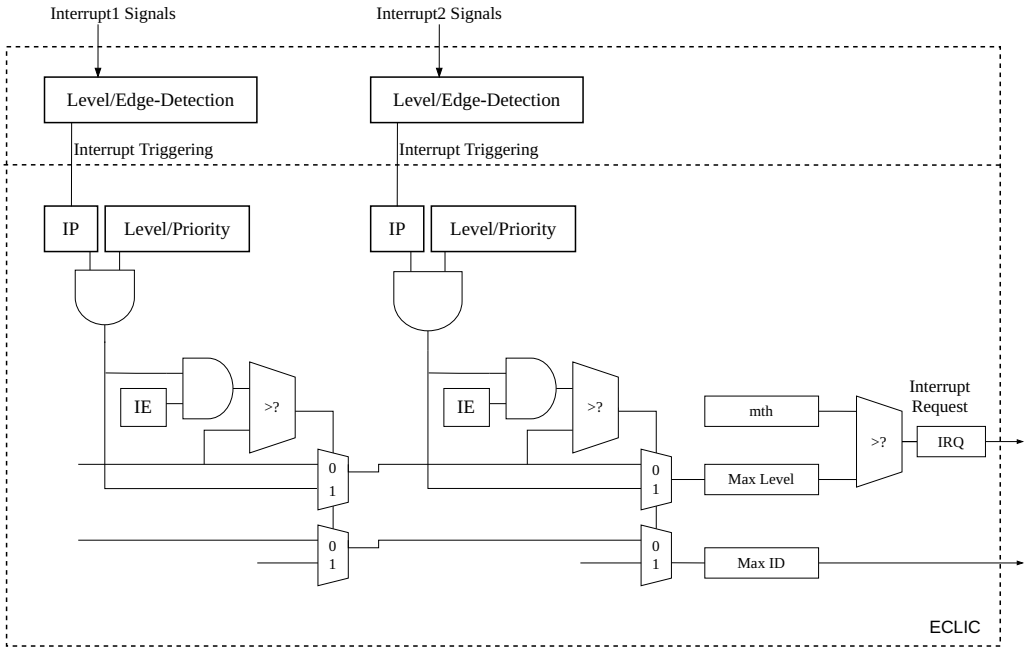


Fig. 12.1: The logic structure of the ECLIC unit

The ECLIC unit is used to arbitrate multiple internal and external interrupts, send interrupt request to core, and support the interrupt preemption. The registers of the ECLIC are described in *The addresses offset of registers in the ECLIC unit* (page 49), and its structure is shown in *The logic structure of the ECLIC unit* (page 46) and the related concepts are as follows:

- ECLIC interrupt target
- ECLIC interrupt source
- ECLIC interrupt source ID
- ECLIC registers
- ECLIC interrupt enable bits
- ECLIC interrupt pending bits
- ECLIC interrupt level or edge triggered attribute
- ECLIC interrupt level and priority
- ECLIC interrupt vectored or non-vectored processing mode
- ECLIC interrupt threshold level
- ECLIC interrupt arbitration mechanism
- ECLIC interrupt response, preemption, tail-chaining mechanism

These will be detailed at next sections.

12.2 ECLIC interrupt target

The ECLIC unit arbitrates the interrupt sources to the processor core (as the interrupt target) by a line as shown in Figure 10-2.

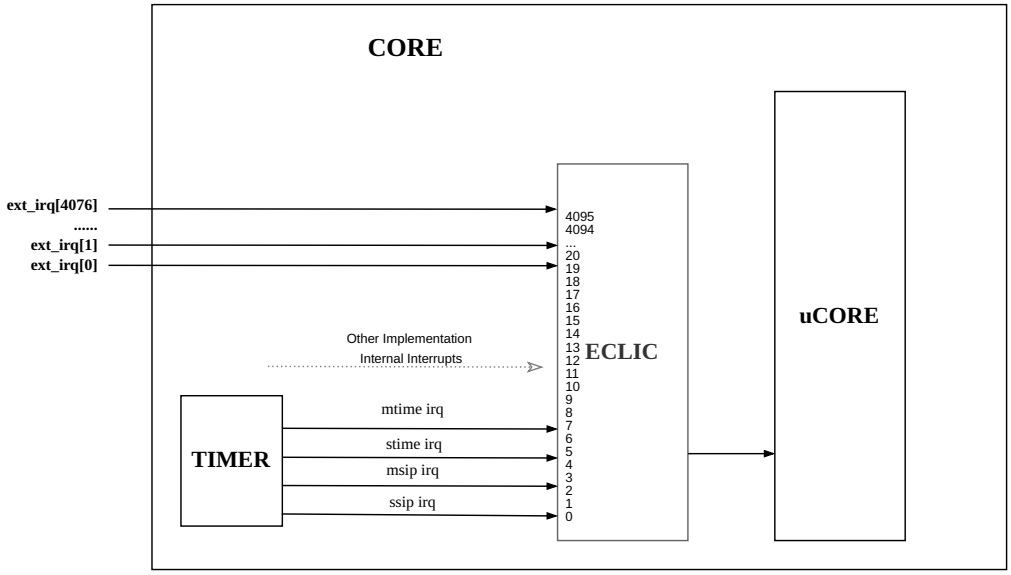


Fig. 12.2: ECLIC Connection (when ECLIC is enabled)

12.3 ECLIC Interrupt Source

As shown in *ECLIC Connection (when ECLIC is enabled)* (page 47), the ECLIC unit can support up to 4096 interrupt sources. The ECLIC unit has defined the following concepts for each interrupt source:

- ID
- IE
- IP
- Level or Edge-Triggered
- Level and Priority

- Vector or Non-Vector Mode

These concepts will be detailed at next sections.

12.4 ECLIC Interrupt Source ID

The ECLIC unit has assigned a unique ID to each interrupt source. For example, if a hardware implementation of the ECLIC unit really configured to support 4096 interrupts, then the ID should be 0 to 4095. Note:

- In the Nuclei processor core, the interrupt IDs ranged from 0 to 18 are reserved for the core-specified internal interrupts. 3 & 7 are fixed for Software Interrupt and Timer Interrupt in all Nuclei processor cores. Some Nuclei core may have more internal interrupts, this table dose not show them, please check related chapter for more details.
- The interrupt source ID greater than 18 can be used by the user to connect external interrupt sources.

The details are shown in *ECLIC interrupt sources and assignment* (page 48).

Table 12.1: ECLIC interrupt sources and assignment

ECLIC interrupt ID	Function	Interrupt Source Description
0	Reserved	This source is not used
1	Supervisor Software interrupt	The s-mode software interrupt generated by the TIMER
2	Reserved	This source is not used
3	Machine Software interrupt	The m-mode software interrupt generated by the TIMER
4	Reserved	This source is not used
5	Supervisor Timer interrupt	The s-mode timer interrupt generated by the Sstc
6	Reserved	This source is not used
7	Machine Timer interrupt	The m-mode timer interrupt generated by the TIMER
8	Reserved	This source is not used
9	Reserved	This source is not used
10	Reserved	This source is not used
11	Reserved	This source is not used
12	Reserved	This source is not used
13	Reserved	This source is not used
14	Reserved	This source is not used
15	Reserved	This source is not used
16	Reserved	This source is not used
17	Reserved	This source is not used
18	Reserved	This source is not used
19 ~ 4095	External interrupt	Normal external interrupt defined by users. Note: • Although the ECLIC unit can support up to 4096 interrupt sources per programming mode, the actual number of supported interrupt sources is indicated in the field clicinfo.NUM_INTERRUPT.

12.5 ECLIC Registers

The ECLIC is a memory-mapped unit.

- The base address of the ECLIC unit, please refer to the specific databook of the Nuclei processor core.
- Registers and the corresponding offset in the ECLIC unit are shown in *The addresses offset of registers in the ECLIC unit* (page 49)

Table 12.2: The addresses offset of registers in the ECLIC unit

Offset	Permission	Register	Width
0x0000	RW	cliccfg	8-bit
0x0004	R	clicinfo	32-bit
0x000b	RW	mth	8-bit
0x1000+4*i	RW	clicintip[i]	8-bit
0x1001+4*i	RW	clicintie[i]	8-bit
0x1002+4*i	RW	clicintattr[i]	8-bit
0x1003+4*i	RW	clicintctl[i]	8-bit

Note:

- The above "i" indicates the interrupt ID, an interrupt i has its own corresponding clicintip[i], clicintie[i], clicintattr[i], and clicintctl[i] registers.
- ECLIC registers only support aligned access which is the size of byte, half-word or word.
- The above "R" means read-only, and any write to this read-only register will be ignored without generating bus error.
- The ECLIC unit may not be configured to support 4096 interrupt sources. If an index i is not present in the hardware, the corresponding clicintip[i], clicintie[i], clicintctl[i] memory locations appear hardwired to zero.
- The address space of ECLIC registers is the range from 0x0000 to 0xFFFF. The value in an address other than the address listed in the above table is constant 0.

These registers are detailed in the next sections.

12.5.1 cliccfg

This cliccfg register is a global configuration register. The software can set global configurations by write this register. *cliccfg bit fields* (page 49) describes the bit fields of this register.

Table 12.3: cliccfg bit fields

Field	Bits	Permission	Default Value	Description
Reserved	7:5	R	N/A	Reserved, ties to 0.
nlbits	4:1	RW	0	Used to specified the bit-width of level and priority in the register clicintctl[i]. Please see *ECLIC Interrupt Level and Priority* (page 52) for more details.
Reserved	0	R	N/A	Reserved, ties to 1.

12.5.2 clicinfo

The clicinfo register is a global info register. The software can query the global parameters by reading this register. *clicinfo bit fields* (page 50) describes the bit fields of this register.

Table 12.4: clicinfo bit fields

Field	Bits	Permission	Default Value	Description
Reserved	31:25	R	N/A	Reserved, ties to 0.
CLICINTCTLBITS	24:21	R	N/A	Used to specify the effective bit-width the register clicintctl[i]. Please see *ECLIC Interrupt Level and Priority* (page 52) for more details.
VERSION	20:13	R	N/A	Hardware implementation version number.
NUM_INTERRUPT	12:0	R	N/A	Number of interrupt sources supported by the hardware.

12.5.3 mth

The mth register is used the set the target interrupt threshold level. The software can set the target interrupt threshold level by writing this register. *mth bit fields* (page 50) describes the bit fields of this register.

Table 12.5: mth bit fields

Field	Bits	Permission	Default Value	Description
mth	7:0	RW	N/A	Target threshold level register. Please see *ECLIC Interrupt Threshold Level* (page 54) for more details.

12.5.4 clicintip[i]

The clicintip[i] register is the pending flag register for the interrupt source. *clicintip[i] bit fields* (page 50) describes the bit fields of this register.

Table 12.6: clicintip[i] bit fields

Field	Bits	Permission	Default Value	Description
Reserved	7:1	RO	N/A	Reserved, ties to 0
IP	0	RW	0	Interrupt source pending flag. Please see *ECLIC Interrupt Pending Bit (IP)* (page 51) for more details.

12.5.5 clicintie[i]

The clicintie[i] register is the enable bit register for the interrupt source. *clicintie[i] bits fields* (page 50) describes the bit fields of this register.

Table 12.7: clicintie[i] bits fields

Field	Bits	Permission	Default Value	Description
Reserved	7:1	R	N/A	Reserved, ties to 0.
IE	0	RW	0	Interrupt enable bit. Please see *ECLIC Interrupt Enable Bit (IE)* (page 51) for more details.

12.5.6 clicintattr[i]

The clicintattr[i] register is used to indicate the attribute of the interrupt source. The software can set the attribute of the interrupt source by writing this register.:ref:*table-clicintattr[i]_bit_fields* describes the bit fields of this register.

Table 12.8: clicintattr[i] bits fields

Field	Bits	Accessibility	Default Value	Description
Reserved	7:6	R	N/A	Reserved, ties to 2'b11
Reserved	5:3	R	N/A	Reserved, ties to 0
trig	2:1	RW	0	Used to set the level or edge triggered attribute of the interrupt source. Please see *ECLIC Interrupt Source Level or Edge-Triggered Attribute* (page 52) for more details.
shv	0	RW	0	Used to set whether the interrupt is vectored or non-vectored. Please see *ECLIC Interrupt Vectored and Non-Vectored Processing Mode* (page 54) for more details.

12.5.7 clicintctl[i]

The clicintctl[i] register is the control register of the interrupt source. The software can set the level and priority by writing this register. The level and priority field are dynamically allocated based on the value of cliccfg.nlbits. Please see *ECLIC Interrupt Level and Priority* (page 52) for more details.

12.6 ECLIC Interrupt Enable Bit (IE)

As shown in *The logic structure of the ECLIC unit* (page 46), the ECLIC unit has allocated an interrupt enable bit (IE) for each interrupt source which is the field clicintie[i].IE whose function are the follows:

- The clicintie[i] register of each interrupt source is a both readable and writeable memory-mapped register. Hence the software can program it.
- If the clicintie[i] register is programmed to 0, it means that this interrupt source is masked.
- If the clicintie[i] register is programmed to 1, it means that this interrupt is enabled.

12.7 ECLIC Interrupt Pending Bit (IP)

As shown in *The logic structure of the ECLIC unit* (page 46), the ECLIC unit has allocated an interrupt pending bit (IP) for each interrupt source which is the field clicintip[i].IP whose function are the follows:

- If the IP bit of one interrupt source is 1, it means this interrupt is triggered. The trigger condition of the interrupt source depends on whether this interrupt is level-triggered or edge-triggered as described in *ECLIC Interrupt Source Level or Edge-Triggered Attribute* (page 52).
- The IP bit of the interrupt source is both readable and writeable. The behavior of the software writing IP bits depends on whether the interrupt source is level or edge triggered. Please see *ECLIC Interrupt Source Level or Edge-Triggered Attribute* (page 52) for more details.
- For edge-triggered interrupt source, the IP bit may be cleared by the hardware itself. Please see *ECLIC Interrupt Source Level or Edge-Triggered Attribute* (page 52) for more details.

12.8 ECLIC Interrupt Source Level or Edge-Triggered Attribute

As shown in *The logic structure of the ECLIC unit* (page 46), each ECLIC interrupt source can be set as level triggered or edge triggered by setting the value of clicintattr[i].trig. The key points are the followings:

- When clicintattr[i].trig[0] == 0, this interrupt source is set as a level-triggered interrupt.
  - If the interrupt source is set as level-triggered, the IP bit reflects the level of the interrupt source in real time, so software writes to this IP bit is ignored, that is, the software cannot set or clear the IP bit by writing operation. If the software needs to clear the interrupt pending bit, it can only be done by clearing the original source of the interrupt.
- When clicintattr[i].trig[0] == 1 and clicintattr[i].trig[1] == 0, this interrupt source is set as a rising edge-triggered interrupt:
  - If the interrupt source is set as rising edge-triggered, when the ECLIC detects the rising edge of the interrupt source, the interrupt source is triggered in the ECLIC, and the IP bit of the interrupt source is asserted.
  - If the interrupt source is set as rising edge-triggered, the IP bit is writeable for the software, which means the software can set or clear the IP bit by writing operations.
- When clicintattr[i].trig[0] == 1 and clicintattr[i].trig[1] == 1, this interrupt source is set as a falling edge-triggered interrupt:
  - If the interrupt source is set as falling edge-triggered, when the ECLIC detects the falling edge of the interrupt source, the interrupt source is triggered in the ECLIC, and the IP bit of the interrupt source is asserted.
  - If the interrupt source is configured as falling edge-triggered, the IP bit is writeable for the software, which means the software can set or clear the IP bit by writing operations

---

Note: For rising or falling edge-triggered interrupt, in order to improve the efficiency of the interrupt processing, when the interrupt is taken (for vectored interrupt, the core jumps to the corresponding Interrupt Service Routine means taken; for non-vectored interrupt, the core jumps to the command entry and then operates the CSR *mnxti* or *jalmnxti* means taken), the hardware of the ECLIC will clear the IP bit automatically, the software doesn't need to clear the IP bit.

12.9 ECLIC Interrupt Level and Priority

As shown in *The logic structure of the ECLIC unit* (page 46), each interrupt sources of the ECLIC can be configured with specified level and priority, and the key points are the followings:

- The register clicintctl[i] of each interrupt source is 8-bit width theoretically, and effective bits actually implemented by the hardware are specified by the CLICINTCTLBITS in the register clicinfo. For example, if the value of the clicinfo.CLICINTCTLBITS field is 6, it means that only the upper 6-bit of the clicintctl[i] register are true valid bits, and the lowest 2 bits are tied to 1, as shown in *clicintctl[i] format example* (page 53).
  - Note: the field CLICINTCTLBITS is a readable constant value, and the software cannot overwrite it. The theoretically reasonable value range of it is 2 <= CLICINTCTLBITS <= 8. The actual value is determined by the specified hardware implementation. Please refer to the specific databook of the Nuclei processor core.
- The effective bits of clicintctl[i] register have two dynamic fields, which are used to specify the level and the priority of the interrupt source. The width of the level filed is defined by field nlbits in cliccfg. For example, if the value of cliccfg.nlbits is 4, it means that the upper 4-bit of the effective bits in clicintctl[i] is the level field while the other lower effective bits is the priority field, as shown in the example in *clicintctl[i] format example* (page 53).
  - Note: the field cliccfg.nlbits is both readable and writeable, which means the software can change its value.

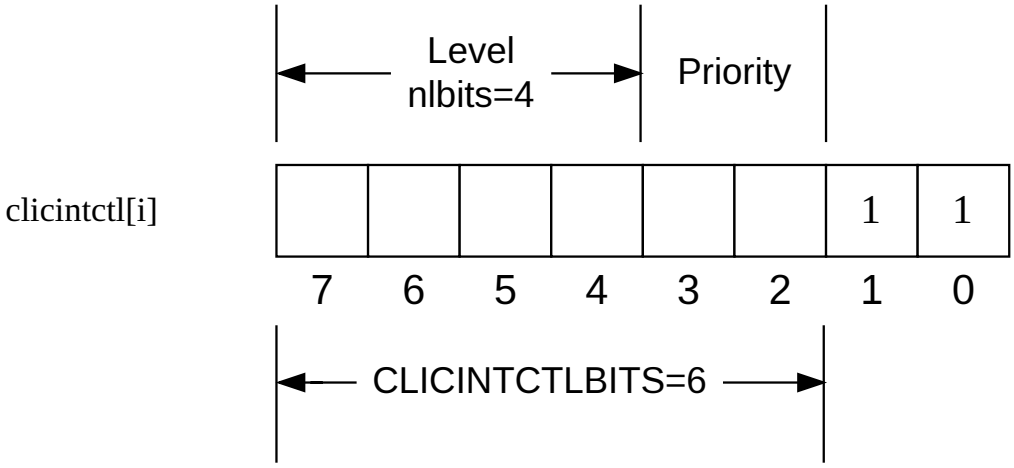


Fig. 12.3: clicintctl[i] format example

- The key points of interrupt level are the followings:
  - The value of level is read in a left-aligned manner. Except the effective bits (defined by the value of cliccfg.nlbits), the low ineffective bits are all filled with the constant 1, as shown in the example in *Example for the decoding of level* (page 53).
    - Note: if cliccfg.nlbits > clicinfo.CLICINTCTLBITS, it means that the number of bits indicated by nlbits exceeds the effective bits of the clicintctl[i] register, and the excess bits are all filled with the constant 1.
    - Note: if cliccfg.nlbits = 0, the value of level will be regarded as a fixed value 255. As shown in *Examples of cliccfg settings* (page 54).
  - The greater value of level, the higher priority, note:
    - Higher-level interrupts can preempt lower-level interrupts, which is called as interrupt preemption, as detailed in *(CLIC mode) Interrupt Preemption* (page 32).
    - If there are multiple pending interrupts (IP is 1), then the ECLIC needs to make an arbitration to determine which interrupt needs to be sent to the core to take. The arbitration needs to take the level of each interrupt source into the consideration. Please see *Interrupt Levels, Priorities and Arbitration* (page 26) for details.

#nlbits	Encoding	The value of level
1	L..... (= L1111111)	127, 255
2	LL..... (= LL111111)	63, 127, 191, 255
3	LLL..... (= LLL11111)	31, 63, 95, 127, 159, 191, 223, 255
4	LLLL.... (= LLLL1111)	15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255

"L" indicates the field of level
"." indicates the rest bits and are filled with the constant 1

Fig. 12.4: Example for the decoding of level

Examples of cliccfg settings:

CLICINTCTLBITS	nlbits	clicintctl[i]	the value of level
0	2	.....	255
1	2	L.....	127, 255
2	2	LL.....	63, 127, 191, 255
3	3	LLL.....	31, 63, 95, 127, 159, 191, 223, 255
4	1	LPPP....	127, 255

Fig. 12.5: Examples of cliccfg settings

- The key points of the interrupt priority are the follows:
  - The value of priority is also read in a left-aligned manner. Except the effective bits (clicinfo.CLICINTCTLBITS - cliccfg.nlbits), the low ineffective bits are all filled with the constant 1.
  - The greater the value of the priority, the higher priority, note:
    * The priority of the interrupt does not participate in the judgment of the interrupt preemption, which means whether the interrupt can be preempted or not has nothing to do with the value of the priority of the interrupt.
    * When multiple interrupts are simultaneously pending, the ECLIC needs to make an arbitration to determine which interrupt is sent to the core to handle. The arbitration needs to refer to the value of level/priority of each interrupt source. Please see *ECLIC Interrupt Arbitration Mechanism* (page 55) for details.

12.10 ECLIC Interrupt Vectored and Non-Vectored Processing Mode

Each interrupt source of the ECLIC can be set to vectored or non-vectored (via the shv field of the register clicintattr[i]). The key points are the followings:

- If the interrupt is set as vectored (clicintattr[i].shv==1), the core will directly jump to the target address stored in the vector table entry when the interrupt is taken. For a detailed description of the interrupt vectored processing mode, please see *Vectored Processing Mode* (page 37).
- If the interrupt is set as non-vectored (clicintattr[i].shv==0), the core will jump to the common base entry shared by all interrupts when the interrupt is taken. For a detailed description of the interrupt non-vectored processing mode, please see *Non-Vectored Processing Mode* (page 34).

12.11 ECLIC Interrupt Threshold Level

As shown in *The logic structure of the ECLIC unit* (page 46), the ECLIC can set the threshold level (mth) of a specific interrupt threshold level. The key points are as follows:

- The mth register is an 8-bit register, all bits are readable and writable, and the software can write this register to set the threshold. Note: this threshold indicates a level value.
- Only when the level of the interrupt finally arbitrated by the ECLIC is higher than the value in the mth register, the interrupt can be sent to the processor core.

12.12 ECLIC Interrupt Arbitration Mechanism

The principles for the ECLIC to arbitrate all of its interrupt sources are as follows:

- Only interrupt sources that meet all of the following conditions can participate in the arbitration:
  - The enable bit (clicintie[i]) of the interrupt source must be 1.
  - The pending bit (clicintip[i]) of the interrupt source must be 1.
- The rules for the arbitration among all participated interrupt sources are:
  - First, check the level, the larger the level value of the interrupt source, the higher the arbitration priority.
  - If the level is equal, then check the priority, the interrupt source that has greater value of priority will have higher arbitration priority.
  - If both level and priority are equal, then the ID is taken into the consideration. The interrupt source with the larger interrupt ID has higher arbitration priority.
- If the level value of the interrupt source that wins the arbitration has a greater value than the level value in mth, then the interrupt request signal to the core will be asserted.

12.13 ECLIC Interrupt Taken, Preemption and Tail-Chaining

After the ECLIC interrupt request is sent to the processor core, the core will respond to it. Through the coordination by the ECLIC and the core, the operation of interrupt preemption and tail-chaining are supported. Please see *(CLIC mode) Interrupt Preemption* (page 32), *(CLIC mode) Interrupt Tail-Chaining* (page 33), and *Non-Vectored Processing Mode* (page 34) for more details.

13 CIDU Introduction

13.1 CIDU Overview

As previous chapters introduce that ECLIC is very suitable for real time application and it only applies to a single core, and PLIC is very suitable for Linux application (single core or SMP). If there is a SMP system which is designed for real time application or both Linux and real time application, a Cluster Interrupt Distribution Unit (CIDU) is needed in these scenarios and Nuclei processor core can optionally support CIDU.

For the location and connection of CIDU within the cluster, please refer to the *Interrupt Connection (for multi-core with ECLIC and CIDU)* (page 63) and *Interrupt Connection (for multi-core with ECLIC/PLIC and CIDU)* (page 64). The CIDU is used to distribute external interrupts to the core's ECLIC, also it provides Inter Core Interrupt (ICI) and Semaphores Mechanism. Its features are as follows:

- Support up to 16 Cores in one cluster
- Support up to 4096 external interrupts sources
- Support up to 16 Inter Core Interrupts
- Support 32 Semaphores

13.2 CIDU Registers and Description

Table 13.1: CIDU Registers list

Addr	RW	Name	Descriptions
0x0	W1C/R	CORE0_INT_STATUS	Core0's Inter Core Interrupt status register
…	…	…	…
0x7c	W1C/R	CORE31_INT_STATUS	Core31's Interrupt Status Register
0x80	RW	SEMAPHORE0	First Semaphore register to use.
…	…	…	…
0xFC	RW	SEMAPHORE31	Last Semaphore register to use.
0x3FFC	WO	ICI_SHADOW_REG	ICI Interrupt source core ID and target core ID
0x4000	RW	INT0_INDICATOR	Indicate interrupt0 can be received by which cores
…	…	…	…
0x7FFC	RW	INT4095_INDICATOR	Indicate interrupt4095 can be received by which cores
0x8000	RW	INT0_MASK	Mask the INT0 to the cores or not when the INT0 Indicator is on
…	…	…	…
0xBFFC	RW	INT4095_MASK	Mask the INT4095 to the cores or not when the INT4095 Indicator is on.

continues on next page

Table 13.1 – continued from previous page

Addr	RW	Name	Descriptions
0xC084	RO	CORE_NUM	Indicate the static configuration core num in the cluster.
0xC090	RO	INT_NUM	Indicate the static configuration interrupt number

Note:

- CIDU is a module within a cluster while IDU can be SOC level module to manage the SOC level interrupt/NMI/event distribution and provide the interrupt cluster interrupt and semaphores. This document will not describe IDU part, if users want to know detail of IDU, please contact Nuclei Support.
- In a Nulcei multi-core system, each core has an unique serial number, the serial number starts from 0 and is continuous, also the number is static and has a direct mapping with the CSR mhartID. In this chapter, the number is called Core Num or Core ID.
- In a Nuclei Subsystem Design with multi cluster of Nuclei Core, then the Core Num or Core ID of this chapter is only the one field (bits 0 ~ 7) of CSR mhartID.

13.3 CIDU External Interrupt Distribution

The CIDU acts as an interrupt distributor in the cluster, it can broadcast all external interrupts to all target cores or let the first coming core to claim the specific interrupt.

13.3.1 Interrupt Distribution Broadcast Mode

If user wants to use CIDU to broadcast the external interrupt to some/all target cores, it just needs to turn on the corresponding bits of the related interrupt's INTn_INDICATOR register. The register detail description is as follows:

Table 13.2: INTn_INDICATOR Register Description

Field Name	Bits	Reset Value	Description
INTn_INDICATOR	core_num_1:0	0x1	Each bit indicates the core can receive this interrupt or not. 1 means yes, 0 means no.

13.3.2 Interrupt Distribution First Come First Claim Mode

If user wants to use CIDU to broadcast the external interrupt to some/all target cores, then let the first coming core to claim the interrupt, beside the INTn_INDICATOR register, it also needs INTn_MASK register to cooperate. The INTn_MASK register's description is as follows:

Table 13.3: INTn_MASK Register Description

Field Name	Bits	Reset Value	Description
INTn_MASK	core_num_1:0	{core_num{1}}	Each bit indicates it should mask the indicator bit or not. 1 means no, 0 means yes. Can only be written when its value is all 1 (reset value) or be written as all 1.

The key points are the followings:

- Write the right value to INTn_Indicator register.
- When the external interrupt *n* comes, it will broadcast to all targeted cores. In the interrupt *n*'s handler of all cores, firstly set the bit[Core ID] as 1'b1 and clear other bits as 1'b0 in INTn_MASK, then read back the INTn_MASK value, if bit[Core ID] is 1'b1, it means it claim the interrupt successfully, then continues to handle the interrupt normally; if not, it means other core has claimed the interrupt, so quit the interrupt handler.

- When a core claims the interrupt successfully and handle the interrupt, before executing mret/sret, it should write the reset value (all 1) to INTn_MASK.

13.4 CIDU Inter Core Interrupt

Inter Core Interrupt (ICI) means that one core can send interrupt to another core in a multi-core cluster. CIDU ICI belongs to Internal Interrupt (please refer to *Interrupt Type* (page 24) and *Internal Interrupt* (page 25) to review the concept) , and the **CIDU ICI Interrupt ID is fixed to 16**. CIDU implements a good Inter Core Interrupt mechanism with two kind registers as follows:

Table 13.4: ICI_SHADOW_REG Register Description

Field Name	Bits	Reset Value	Description
REV_CORE_ID	core_num-1:0	0x0	The Core ID which receives this Inter Core Interrupt
SEND_CORE_ID	core_num+15:16	0x0	The Core ID which sends this Inter Core Interrupt

Table 13.5: COREn_INT_STATUS Register Description

Field Name	Bits	Reset Value	Description
COREn_INT_STATUS	core_num-1:0	0x0	inter core interrupt status register of core n, bit 0 means core0 and bit 31 means core 31. Each bit can be cleared by writing 1.

If Core n (Core ID=n) wants to send the ICI to Core m (Core ID =m), then the flow is simple:

- Core n writes the value n to bit[31:16] and writes the value m to bit[15:0] in register ICI_SHADOW_REG.
- Then CIDU will write the n to COREm_INT_STATUS register and trigger ICI to Core m automatically.
- When Core m receives the ICI and enter the ICI Interrupt Service Routine (ISR), then it can query the bit[n] in Corem_INI_STATUS register and will get that the current ICI is triggered by Core n. And the Core m also need to write 1 to clear the bit n of the Corem_INT_STATUS before the ISR returns.

Note:

- The ICI_SHADOW_REG is Write Only register, if multiple cores write this register simultaneously, it will record all the write behavior (won't miss any) and handle them properly.
- In the ISR of ICI, it should clear the corresponding bit/bits of its own COREn_INT_STATUS. If the ISR has finished the job of Core_n's ICI, then clear Core n; if it has finished Core_n and Core_m's, then clear both the bits and etc.
- RISC-V Spec CLINT defines a Software Interrupt per core and Nuclei Processor Core's Timer Module implements it, upstream RISC-V Linux already adopt CLINT's Software Interrupt to implement Inter-Processor Interrupt (IPI). But currently Software Interrupt has only 1-bit pending bit, it is hard to tell which cores send the ICI while CIDU well enhances it. User can choose CLINT Software Interrupt or CIDU to implement ICI by request.

13.5 CIDU Semaphore

CIDU also provides up to 32 semaphores to let users to configure and use. And Semaphore is very useful for multi-core cluster or multi-cluster without SMP enable. The related register is as follows:

Table 13.6: SEMAPHOREn Description

Field Name	Bits	Reset Value	Description
CORE_STATUS	3:0	0xf	Reset value is 0xf, other value means the specific core owns the semaphore.

continues on next page

Table 13.6 – continued from previous page

Field Name	Bits	Reset Value	Description
CLUSTER_STATUS	9:4	0x3f	Reset value is 0x3f, other value means the specific cluster owns the semaphore.
Reserved	31:10	0	Reserved.

The key points of software to use semaphores are the followings:

- All Cores in the clusters agree on using SEMAPHORE_n register to protect a critical resource (an UART device for example).
- If Core m of Cluster k wants to access the critical resource, it should try to own the SEMPAPHORE_n register by writing [k, m], then read the SEMAPHORE_n value and compare with the value with original [Cluster_ID, Core ID], if matches, then it owns the semaphore successfully and can access the critical resource; if it not matches, obviously it does not own the semaphore and can't access the critical resource.
- When the Core m of Cluster k owns the register SEMPAPHORE_n and finishes the job related to the critical resource, then it should release the register by write 0x3ff.

14 ECLIC, PLIC and CIDU Connection Diagram

The ECLIC and PLIC are independently configurable, so they can be co-existed or not, depends on configurations. The key points are as followings:

- For the single-core real-time or micro-controller applications, it is recommended to just have ECLIC configured, as depicted in *Interrupt Connection (for single-core with ECLIC configured only)* (page 61)
- For the single-core Linux capable applications, it is recommended to configure PLIC. But in this case, the ECLIC can also be configured, hence, PLIC and ECLIC become coexisting.
  - If the ECLIC is enabled by software, then the interrupts will be handled by ECLIC, and the PLIC will be bypassed, as depicted in *Interrupt Connection (for single-core with PLIC/ECLIC configured and PLIC enabled)* (page 61).
    - In this mode, the hardware of UX class core can also worked as microcontroller, i.e., Nuclei UX class core is downward-compatible to Nuclei NX class core.
  - If the ECLIC is disabled by software, then the interrupts will be handled by PLIC, and the ECLIC will be bypassed, as depicted in *Interrupt Connection (for single-core with PLIC/ECLIC configured and ECLIC enabled)* (page 62).
- For the symmetric multi-processor (SMP) Linux capable applications, it is recommended to just have PLIC configured only, as depicted in *Interrupt Connection (for multi-core with PLIC configured only)* (page 62).
- For the symmetric multi-processor (SMP) real-time or micro-controller applications, it is recommended to have CIDU for cluster level and ELIC for core level configured, as depicted in *Interrupt Connection (for multi-core with ECLIC and CIDU)* (page 63).
- For the symmetric multi-processor (SMP) to run both Linux and real-time or micro-controller applications, it is recommended to have CIDU, ECLIC and PLIC . As depicted in *Interrupt Connection (for multi-core with ECLIC/PLIC and CIDU)* (page 64).

Note:

In the flowing diagrams, each core may have other implementation related Internal Interrupts, but we only show the Software Interrupt and Timer Interrupt for short.

14.1 Single-core with ECLIC configured only

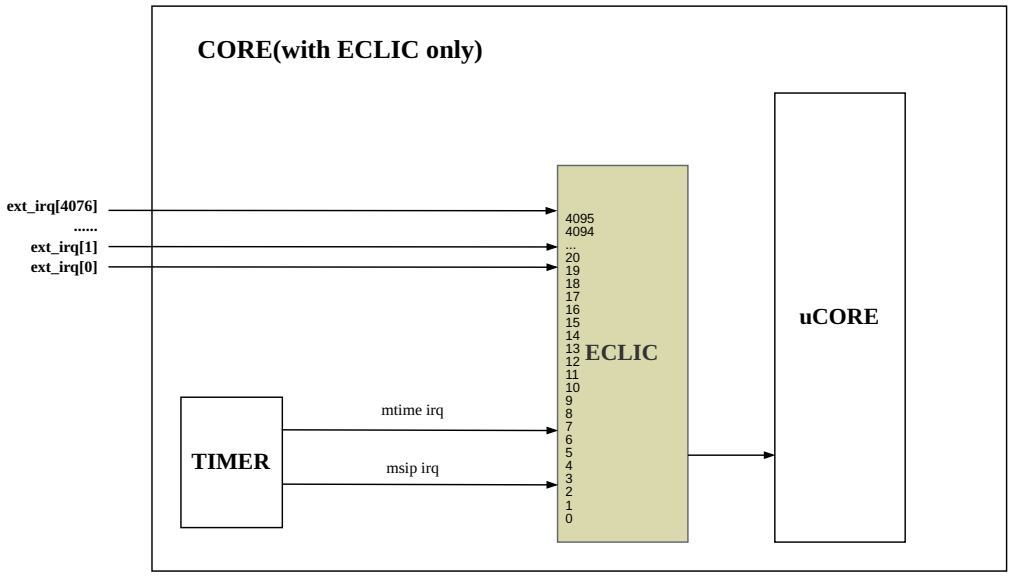


Fig. 14.1: Interrupt Connection (for single-core with ECLIC configured only)

14.2 Single-core with PLIC/ECLIC configured and PLIC enabled

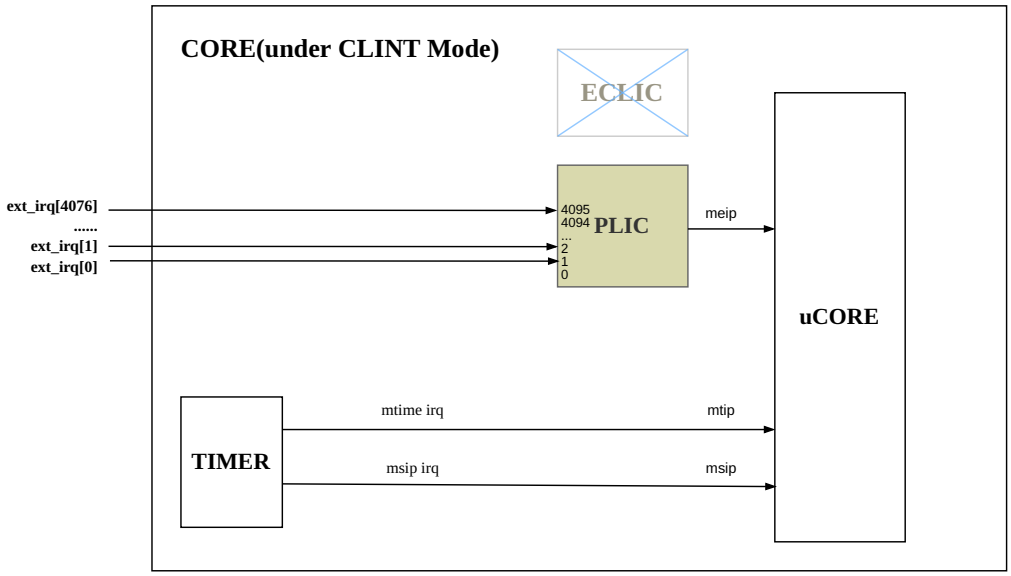


Fig. 14.2: Interrupt Connection (for single-core with PLIC/ECLIC configured and PLIC enabled)

14.3 Single-core with PLIC/ECLIC configured and ECLIC enabled

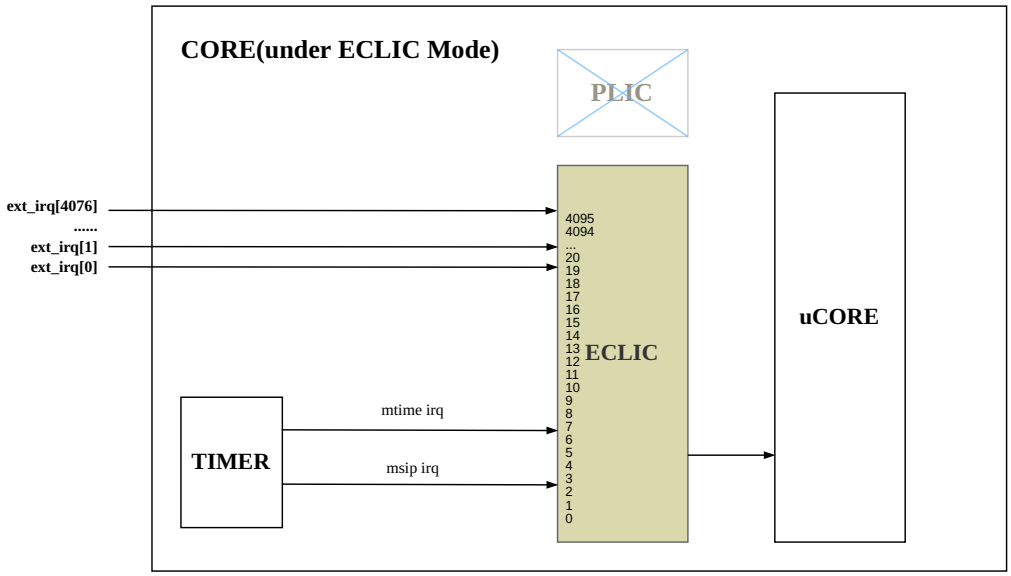


Fig. 14.3: Interrupt Connection (for single-core with PLIC/ECLIC configured and ECLIC enabled)

14.4 Multi-core with PLIC configured only

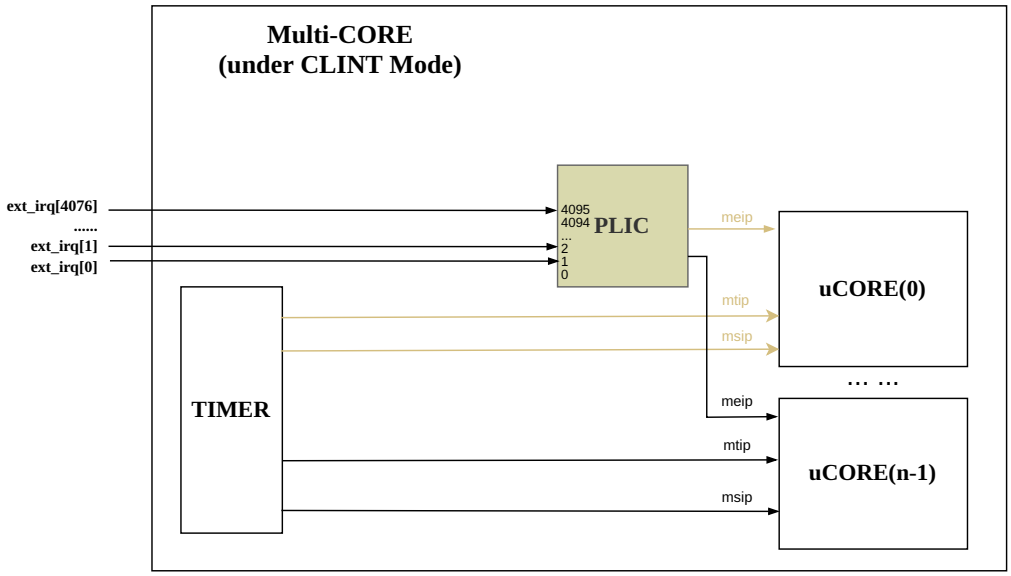


Fig. 14.4: Interrupt Connection (for multi-core with PLIC configured only)

14.5 Multi-core with ECLIC and CIDU

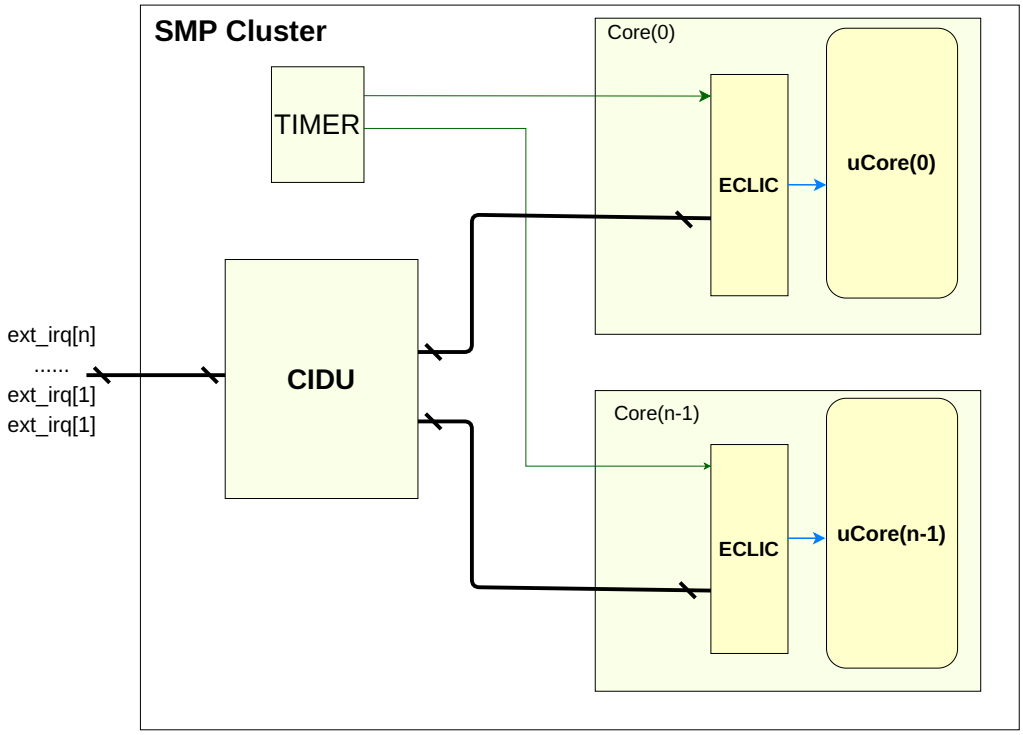


Fig. 14.5: Interrupt Connection (for multi-core with ECLIC and CIDU)

14.6 Multi-core with ECLIC/PLIC and CIDU

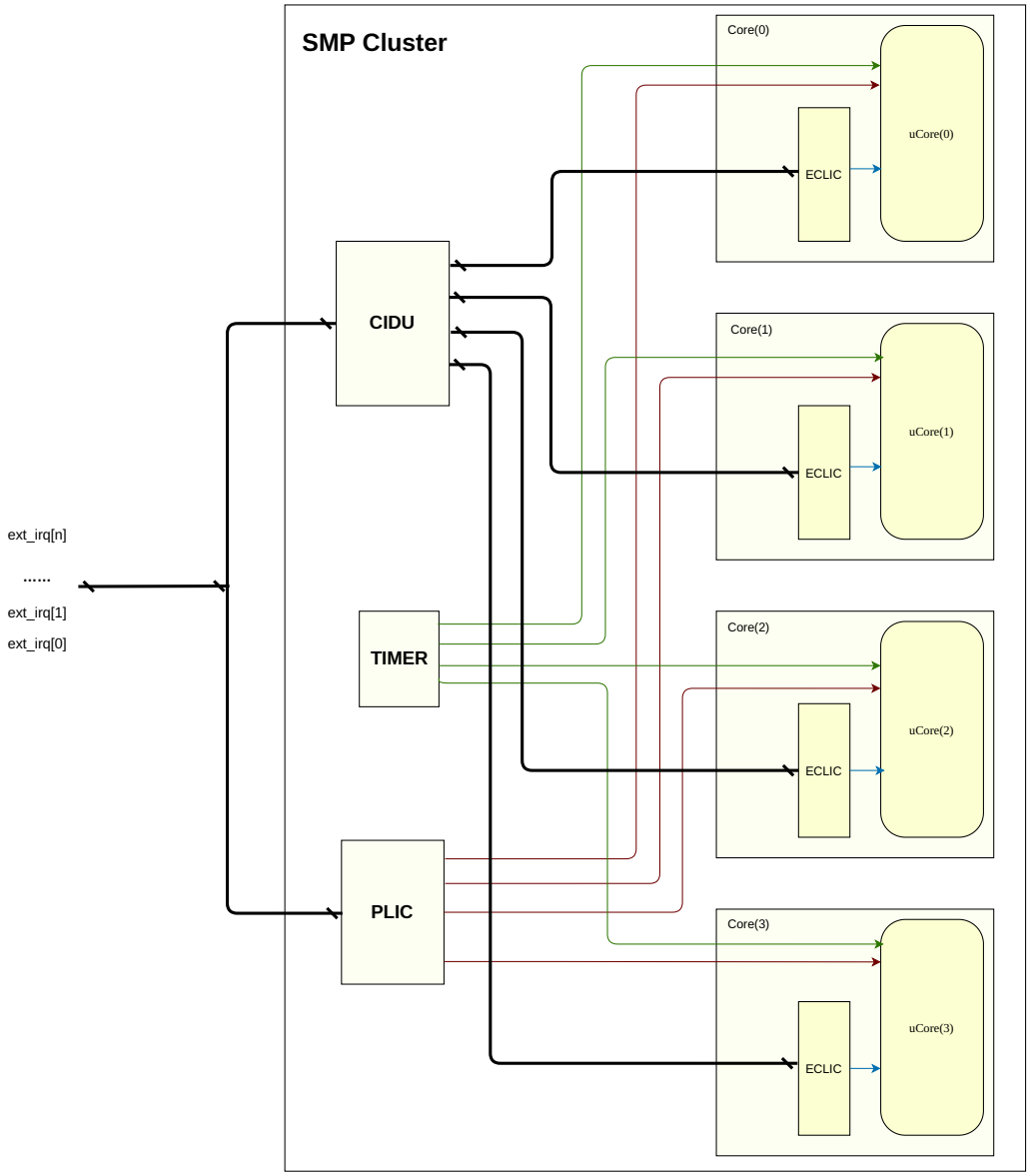


Fig. 14.6: Interrupt Connection (for multi-core with ECLIC/PLIC and CIDU)

15 PMP Introduction

15.1 PMP Overview

Nuclei processor core can optionally support the PMP (Physical Memory Protection) and Smepmp features, which is part of RISC-V standard privileged architecture specification.

15.2 PMP Specific Features to Nuclei Core

In order to simplify the hardware implementation, or there are some features are intrinsically hardware implementation relevant, Nuclei processor core have some PMP specific features, which is detailed at next sections.

15.2.1 Configurable PMP Entries Number

RISC-V standard privileged architecture specified the number of PMP entries up to 16, but in the real hardware the PMP entries is limited. Nuclei processor core have the PMP entries number configurable at the build time, please refer to the specific databook of the Nuclei processor core.

For those PMP CSR registers relevant to non-existed PMP entries, they are tied to zeros. For example, if the PMP Entries Number is configurable to 8, then the PMP entry 9 ~ 16 relevant CSR registers are tied to zeros.

15.2.2 Configurable PMP Grain

RISC-V standard privileged architecture allows platform to define its own PMP grain, please refer to the RISC-V privileged specification for more details of this PMP grain definitions. Nuclei processor core have the PMP grain configurable at the build time, please refer to the specific databook of the Nuclei processor core.

15.2.3 Support TOR mode in A field of pmpcfg<x> registers

RISC-V standard privileged architecture specified 4 types of modes in A field of pmpcfg<x> registers. Nuclei processor core already supports the TOR mode in A field of pmpcfg<x> registers.

15.2.4 Corner cases for Boundary Crossing

Since the RISC-V standard privileged architecture specified the PMP regions as naturally aligned power-of-2 regions (NAPOT), but in the Nuclei processor core hardware implementations, there might be unaligned access crossing the boundary, which is handled in this way:

- If it is normal load/store instructions:
  - If the processor core is configured to not support the unaligned memory access, then it will trigger address misaligned exception.
  - If the processor core is configured to support the unaligned memory access, then hardware will break the unaligned access into multiple byte or half-word aligned memory accesses (called micro-operations), each micro-opertation will be checked with PMP, if violated PMP permission, it will trigger Load or Store access fault exception, and the exception is imprecise.
- If it is AMO/LR/SC instructions:
  - Since the RISC-V standard privileged architecture specified AMO/LR/SC instructions definitely not support the unaligned memory access, hence it will trigger address misaligned exception.
- If it is the instruction fetching, since the processor core will always break the instruction fetching as the aligned memory access (e.g., 32bits or 64bits aligned), each aligned memory access will be checked with PMP, if violated PMP permission, it will trigger instruction access fault exception, and the exception is precise.

16 TEE Introduction

16.1 Revision History

Rev.	Revision Date	Revised Content
1.0.0	2019/11/12	1. Initial Release.
1.0.1	2021/02/01	1. Add sdcause in 2.2. 2. Modify some typos related with user mode and supervisor mode.
1.1.0	2023/02/08	1. Nuclei TEE follows RISC-V Ssmpu Extension Spec.

16.2 TEE Introduction

Nuclei provides the TEE (Trusted Execution Environment) to achieve better isolation for implementing supervisor-level interrupt/exception handling and sPMP, the sPMP basically follows RISC-V Ssmpu Extension v0.9.0.

With TEE, it is flexible to isolate machine-level interrupt/exception from lower privileged-level interrupt/exception.

When the TEE is configured, and the outer execution environment has delegated specified interrupts and exceptions to supervisor-level, then hardware can transfer control directly to a supervisor-level trap handler without invoking the outer execution environment. Since then, users have more flexibility to define the behavior of hardware when an interrupt or exception is taken. Users can delegate the specified exceptions through the TEE. As for interrupts, with the TEE, users can delegate the designated interrupts only when ECLIC is configured, please refer ECLIC register *clicintattr* in later chapter. The NMI (Non-maskable-interrupt) cannot be trapped to the supervisor-mode or user-mode for any configuration.

Moreover, with TEE, it enables S-mode OS to limit the physical addresses accessible by U-mode software.

When the TEE is configured, sPMP entries can be set to achieve better isolation between OS (running on S-mode) from user software (running on U-mode), and to achieve better scalability of PMP-based TEE/enclave with smaller TCB (Trusted Computing Base).

16.3 TEE Related CSRs

16.3.1 Added CSRs List For The TEE

Based on the basic CSRs, the TEE has added some CSRs as shown in following table. They can be classified to following as the following five categories:

- RISC-V Standard S Mode CSRs:

  status, *sie*, *stvec*, *scouteren*, *sscratch*, *spec*, *scause*, *stval*, *sip*, *satp*

- RISC-V Standard M Mode CSRs for S-mode and trap:

  medeleg, *mideleg*

- Nuclei customized CSRS for S-mode PMP:

  spmpcfg<x>, *spmpaddr<x>*

- RISC-V CLIC Draft Extended for S-mode:

  stvt, *snxti*, *sintstauts*, *stvt2*

- Nuclei customized S-mode CSRs for interrupt and exception:

  jalsnxti, *sscratchcsw*, *sscratchcswl*, *pushscause*, *pushsepc*, *sdcause*

Table 16.2: Added CSRs for the TEE

CSR Type	Number	Privilege	Name	Description
RISC-V Standard CSR	0x100	SRW	sstatus	Supervisor status register
	0x104	SRW	sie	Supervisor interrupt-enable register
	0x105	SRW	stvec	Supervisor trap handler base address
	0x106	SRW	scounteren	Supervisor counter enable
	0x140	SRW	sscratch	Scratch register for supervisor trap handlers
	0x141	SRW	sepc	Supervisor exception program counter
	0x142	SRW	scause	Supervisor trap cause
	0x143	SRW	stval	Supervisor trap value register
	0x144	SRW	sip	Supervisor interrupt pending
	0x180	SRW	satp	Supervisor address translation and protection
	0x302	MRW	medeleg	Machine exception delegation register
	0x303	MRW	mideleg	Machine interrupt delegation register
Nuclei Customized CSRs	0x1A0	SRW	spmpcfg0	Supervisor physical memory protection configuration
	0x1A1	SRW	spmpcfg1	Supervisor physical memory protection configuration
	0x1A2	SRW	spmpcfg2	Supervisor physical memory protection configuration
	0x1A3	SRW	spmpcfg3	Supervisor physical memory protection configuration
	0x1B0	SRW	spmpaddr0	Supervisor physical memory protection address register
	0x1B1	SRW	spmpaddr1	Supervisor physical memory protection address register
	...	SRW	...	...
	(0x1B0+n)	SRW	spmpaddrn	Supervisor physical memory protection address register
	0x107	SRW	stvt	Supervisor Trap-handler vector table base address
	0x145	SRW	snxti	Supervisor interrupt handler address and enable modifier

continues on next page

Table 16.2 – continued from previous page

CSR Type	Number	Privilege	Name	Description
	0x146	SRW	sintstatus	Supervisor current interrupt level
	0x148	SRW	sscratchcsw	Scratch swap register for multiple privilege modes
	0x149	SRW	sscratchcswl	Scratch swap register for supervisor interrupt levels
	0x947	SRW	jalsnxti	Jumping to next supervisor interrupt handler address and interrupt-enable register
	0x948	SRW	stvt2	ECLIC non-vectored supervisor interrupt handler address register
	0x949	SRW	pushscause	Push scause to stack
	0x94a	SRW	pushsepc	Push sepc to stack
	0x9c0	SRW	sdcause	Detail infromation for scause

16.3.2 RISC-V Standard CSR For TEE

16.3.2.1 sstatus

The *sstatus* register is a SXLEN read/write register the lower 32-bits formatted as shown in *sstatus register* (page 69). The *sstatus* register keeps track of and controls the hart's current operating state.

Fig. 16.1: sstatus register

The supervisor interrupt-enable bit SIE indicates supervisor-level interrupts are disabled when it is clear. The value of SIE is preserved in SPIE when a supervisor-level trap is taken, and the value of SIE is set to zero to provide atomicity for the supervisor-level trap handler.

The SUM (permit Supervisor User Memory access) bit modifes the privilege with which S-mode loads and stores access user mode memory, see *SMAP and SMEP with sPMP* (page 90) for more details about SUM bit .

The SRET instruction is used to return from traps in S-mode, an instruction unique to S-mode. SRET sets PC to sepc, restores interrupt-enable setting by copying SPIE to SIE, and sets SPIE bit.

When the TEE is configured, the *mstatus* register mirrors the SIE and SPIE fields in bit 1 and bit 5, the same location with *sstatus* as shown in *mstatus register* (page 69).

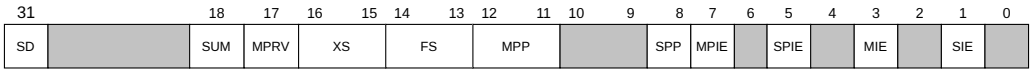


Fig. 16.2: mstatus register

16.3.2.2 sie

The *sie* register has **NO** effect when interrupt handling mode is ECLIC, and return data are all zeros while reading the register.

16.3.2.3 stvec

The *stvec* register is a SXLEN read/write register the lower 32-bits formatted as shown in *stvec register* (page 70). It functions in an analogous way to the *mtvec* register defined in M-mode.

The *stvec* register holds S-mode exception and interrupt configuration, consisting of exception and CLIC non-vector base address (BASE).

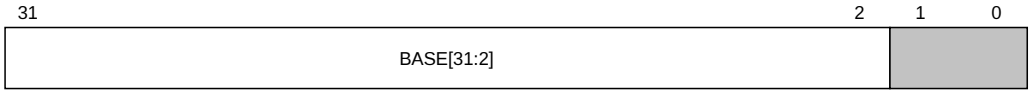


Fig. 16.3: stvec register

16.3.2.4 scounteren

The counter-enable register *scounteren* controls the availability of the hardware performance monitoring counters to U-mode.

When the CY, TM or IR bit in the *scounteren* register is clear, attempts to read the cycle, time or instret register while executing in U-mode will cause an illegal instruction exception. When one of these bits is set, access to the corresponding register is permitted.

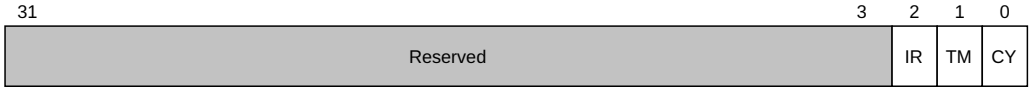


Fig. 16.4: scounter register

16.3.2.5 stvt

The *stvt* register is a SXLEN read/write register the lower 32-bits formatted as shown in *stvt register* (page 70). It functions in an analogous way to the *mtvt* register defined in M-mode.

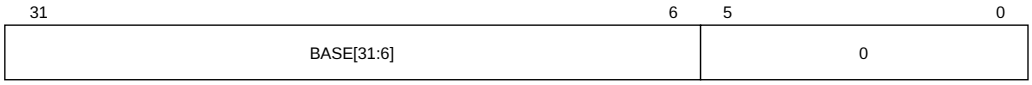


Fig. 16.5: stvt register

The *stvt* register holds the base address of ECLIC S-mode vector interrupts, and the base address is aligned at least 64-byte boundary. In order to improve the performance and reduce the gate count, the alignment of the base address in *stvt* is determined by the actual number of interrupts, which is shown in *Alignment of stvt base address* (page 70).

Table 16.3: Alignment of stvt base address

interrupt number	alignment
0 to 16	64-byte
17 to 32	128-byte
33 to 64	256-byte
65 to 128	512-byte
129 to 256	1KB
257 to 512	2KB
513 to 1024	4KB

16.3.2.6 sscratch

The *sscratch* register is a SXLEN read/write register dedicated for use by supervisor mode. Typically, it is used to hold a pointer to a user mode hart-local context space and swapped with a supervisor register upon entry to a S-mode trap handler.

Fig. 16.6: sscratch register

16.3.2.7 sepc

The *sepc* register is a SXLEN read/write register the lower 32-bits formatted as shown in *sepc register* (page 71). It functions in an analogous way to the *mepc* register defined in M-mode.

When an exception or interrupt is taken into S-mode, *sepc* is written with the virtual address of the instruction that encountered the exception or interrupt. Otherwise, *sepc* is never written by the implementation, though it may be explicitly written by software.

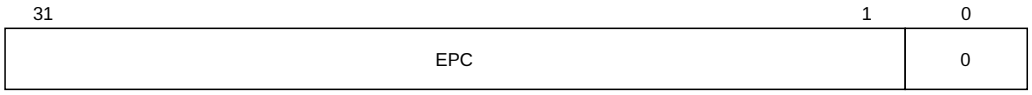


Fig. 16.7: sepc register

16.3.2.8 scause

The *scause* register is a SXLEN read/write register the lower 32-bits formatted as shown in *scause register* (page 71). It functions in an analogous way to the *mcause* register defined in M-mode.

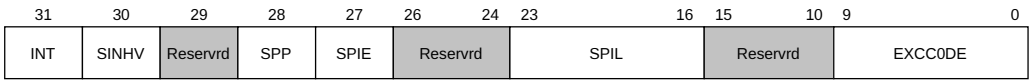


Fig. 16.8: scause register

Table 16.4: scause register description

Field Name	Bit	Description
INT	31	The bit is set if the trap was caused by an interrupt.
SINHV	30	The bit indicates that the interrupt is reading the interrupt vector table.
Reserved	29	0
SPP	28	The bit indicates privilege mode before S-mode exception or interrupt is taken. It mirrors sstatus.spp
SPIE	27	The bit holds the SIE value before S-mode exception or interrupt is taken. It mirrors sstatus.spie
Reserved	26:24	0
SPIL	23:16	The bits hold the interrupt level before S-mode interrupt is taken.
Reserved	15:10	0
EXCCODE	9:0	The bits hold the code of last exception or interrupt

16.3.2.9 stval

The *stval* register is a SXLEN read/write register the lower 32-bits formatted as shown in *stval register* (page 72). When a trap is taken into S-mode, *stval* is written with exception-specific information to assist software in handling the trap. Otherwise, *stval* is never written by the implementation, though it may be explicitly written by software.

When a hardware breakpoint is triggered, or an instruction-fetch, load, or store address-misaligned, access exception occurs, *stval* is written with the faulting effective address.

On an illegal instruction trap, *stval* is written with the faulting instruction code.

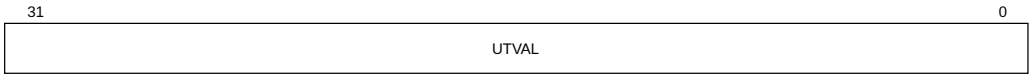


Fig. 16.9: stval register

16.3.2.10 sip

The *sip* register has **NO** effect when interrupt handling mode is ECLIC, and return data are all zeros while reading the register.

16.3.2.11 snxti

The *snxti* register is a SXLEN read/write register the lower 32-bits formatted as shown in *snxti register* (page 72). It functions in an analogous way to the *mnxti* register defined in M-mode.

The *snxti* CSR can be used by software to service the next S-mode horizontal or lower level interrupt when it has a higher level than the saved interrupt context (held in *scause.spil*), without incurring the full cost of an interrupt pipeline flush and context save/restore.

Note: If the next interrupt is an M-mode one while the execution privilege mode is S-mode, the processor will take the next interrupt directly in a nested way. In other hand, if the processor is running in M-mode, then a S-mode interrupt will not be taken.

The *snxti* CSR is designed to be accessed using CSRRSI/CSRRCI instructions, where the value read is a pointer to an entry in the S-mode interrupt handler table and the write back updates the S-mode interrupt-enable status.

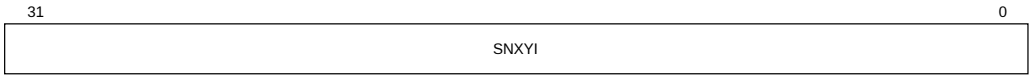


Fig. 16.10: snxti register

16.3.2.12 sintstatus

The *sintstatus* register is a SXLEN read/write register the lower 32-bits formatted as shown in *sintstatus register* (page 72). It functions in an analogous way to the *mintstatus* register defined in M-mode.

The *sintstatus* register holds the active interrupt level for S-mode. The SIL field is read-only.

Fig. 16.11: sintstatus register

When the TEE is configured, the *mintstatus* register mirrors the SIL field in bit 8-15, the same location with *sintstatus* as shown in *mintstatus register* (page 73).

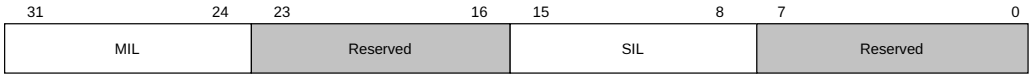


Fig. 16.12: mintstatus register

16.3.2.13 sscratchcsw

The *sscratchcsw* register is a SXLEN read/write register the lower 32-bits formatted as shown in *sscratchcsw register* (page 73). It functions in an analogous way to the *mscratchcsw* register defined in M-mode.

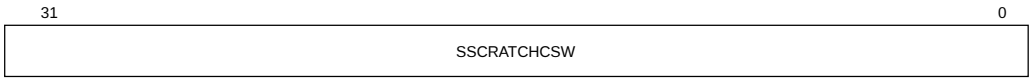


Fig. 16.13: sscratchcsw register

To accelerate interrupt handling with multiple privilege modes, sscratchcsw is defined for supervisor mode to support conditional swapping of the sscratch register when transitioning between supervisor mode and user mode.

```
csrrw rd, sscratchcsw, rs1


  // Pseudocode operation.

  if (scause.spp != S-mode) then {

    t = rs1; rd = sscratch; sscratch = t;

  } else {

    rd = rs1; // sscratch unchanged.

  }

  // Usual use: csrrw sp, sscratchcswl, sp
```

16.3.2.14 sscratchcswl

The *sscratchcswl* register is a SXLEN read/write register the lower 32-bits formatted as shown in *sscratchcswl register* (page 73). It functions in an analogous way to the *mscratchcswl* register defined in M-mode.

Fig. 16.14: sscratchcswl register

Within U-mode, *sscratchcswl* is useful to separate interrupt handler tasks from application tasks to enhance robustness, reduce space usage, and aid in system debugging. Interrupt handler tasks only have non-zero interrupt levels, while application tasks have an interrupt level of zero.

The *sscratchcswl* CSR is added to support faster swapping of the stack pointer between S-mode interrupt and non-interrupt code running in the same privilege mode.

```
csrrw rd, sscratchcswl, rs1

  // Pseudocode operation.
```

(continues on next page)

(continued from previous page)

```
if ( (scause.spil==0) != (sintstatus.sil==0) ) then {

  t = rs1; rd = sscratch; sscratch = t;

} else {

  rd = rs1; // sscratch unchanged.

}

// Usual use: csrrw sp, sscratchcswl, sp
```

16.3.2.15 satp

The *satp* CSR provides the Supervisor Address Translation and Protection.

16.3.2.16 medeleg

The *medeleg* register is a SXLEN read/write register the lower 32-bits formatted as shown in *Medeleg register* (page 74). To increase performance, the *medeleg* register contains individual read/write bits to indicate that certain exceptions should be processed directly by a lower privilege level.

By default, all exceptions at any privilege level are handled in machine mode. When N-Extension is configured, setting a bit in *medeleg* will delegate the corresponding exception in S/U-mode to the S-mode exception handler.

Fig. 16.15: Medeleg register

Table 16.5: Medeleg register description

Field Name	Bits	Description
Reserved	SXLEN-1:16	0
	15	Delegate Store/AMO Page Fault Exception
Reserved	14	0
	13	Delegate Load Page Fault Exception
	12	Delegate Instruction Page Fault Exception
	11	Not used, tie to 0
Reserved	10	0
	9	Not used, tie to 0
	8	Delegate Environment Call from U-Mode
	7	Delegate Store/AMO Access Fault Exception
	6	Delegate Store/AMO Address Misaligned Exception
	5	Delegate Load Access Fault Exception
	4	Delegate Load Access Misaligned Exception
	3	Delegate Breakpoint Exception
	2	Delegate Illegal Instruction Exception
	1	Delegate Instruction Access Fault Exception
	0	Delegate Instruction Access Misaligned Exception

16.3.2.17 mideleg

The *mideleg* register has **NO** effect when interrupt handing mode is ECLIC, and return data are all zeros while reading the register.

16.3.3 Nuclei Customized CSR For TEE

16.3.3.1 S-mode PMP CSRs

The *spmpcfg<x>* and *spmpaddr<x>* are detailed described in Section *Smpu CSRs Introdution* (page 87).

16.3.3.2 S-mode Interrupt/Exception and ECLIC CSRs

16.3.3.2.1 jalsnxti

The *jalsnxti* register is a SXLEN read/write register the lower 32-bits formatted as shown in *jalsnxti register* (page 75). It functions in an analogous way to the *jalmnxti* register defined in M-mode.

The *jalsnxti* register is designed to speed up the S-mode interrupt handling comparing to using *snxti*. In addition to enabling the interrupt, accessing this CSR by 'csrrw ra, jalsnxti, ra' will directly jump to the next S-mode horizontal or lower level interrupt entry address for the same privilege mode, at the same time, save the return address which is the pc of the executing instruction, to the ra register. For more details please refer to *Supervisor-Level Non-vector Mode* (page 84).

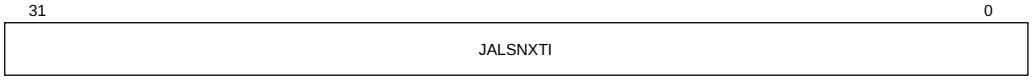


Fig. 16.16: jalsnxti register

16.3.3.2.2 stvt2

The *stvt2* register is a SXLEN read/write register the lower 32-bits formatted as shown in *stvt2 register* (page 75). It functions in an analogous way to the *mtvt2* register defined in M-mode.

The *stvt2* register holds S-mode ECLIC configuration, consisting of ECLIC non-vector base address(BASE) and ECLIC enable(STVT2EN). The ECLIC non-vector base address is aligned on a 4-byte boundary

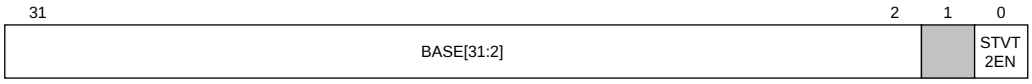


Fig. 16.17: stvt2 register

Table 16.6: stvt2 register description

Field Name	Bits	Description
BASE	SXLEN-1:2	The base address of ECLIC non-vector interrupt in S-mode
Reserved	1	0
STVT2EN	0	Setting this bit will enable ECLIC mode,that means non-vector interrupt base address in S-mode is specified by stvt2 instead of stvec

16.3.3.2.3 sdcause

Since there might be some exceptions share the same scause.EXCCODE value. To further record the differences, Nuclei processor core customized CSR *sdcause* register to record the detailed information about the exception.

Table 16.7: sdcause register description

Field	Bit	Description
Reserved	SXLEN-1:2	Reserved 0
sdcause	2:0	Further record the detailed information about the exception. When scause.EXCCODE = 1 (Instruction access fault) • 0: Reserved • 1: PMP permission violation • 2: Bus error • 3-7: Reserved When scause.EXCCODE = 5 (Load access fault) • 0: Reserved • 1: PMP permission violation • 2: Bus error • 3: NICE extended long pipeline instruction return error. Note: although this error ideally is nothing to do with the Load access fault, but they just shared the same scause.EXCCODE to simplify the hardware implementation • 4-7: Reserved When scause.EXCCODE = 7 (Store/AMO access fault) • 0: Reserved • 1: PMP permission violation • 2: Bus error • 3-7: Reserved When scause.EXCCODE = 12 (Instruction page fault) • 0-4: Reserved • 5: Page fault • 6: SPMP permission violation • 7: Reserved When scause.EXCCODE = 13 (Load page fault) • 0-4: Reserved • 5: Page fault • 6: SPMP permission violation • 7: Reserved When scause.EXCCODE = 15 (Store/AMO page fault) • 0-4: Reserved • 5: Page fault • 6: SPMP permission violation • 7: Reserved

16.3.3.2.4 pushscause

The *pushscause* register is a SXLEN read/write register the lower 32-bits formatted as shown in *pushscause register* (page 76). It functions in an analogous way to the *pushmcause* register defined in M-mode.

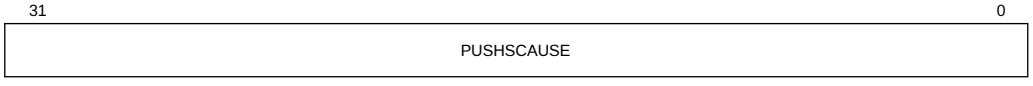


Fig. 16.18: pushscause register

The *pushscause* register is a function CSR and it can't be read or written as a normal CSR. Any CSR instruction except

for csrrwi accessing this register will raise an illegal instruction exception.

The *pushscause* register is designed to speed up the context saving of S-mode interrupt. Using csrrwi instruction, the data of *scause* will be pushed to memory address = sp + imm*4.

```
csrrwi x0, pushscause ,2 // push scause to memory address = sp + 8
```

16.3.3.2.5 pushsepc

The *pushsepc* register is a SXLEN read/write register the lower 32-bits formatted as shown in *pushsepc register* (page 77). It functions in an analogous way to the *pushmepc* register defined in M-mode.

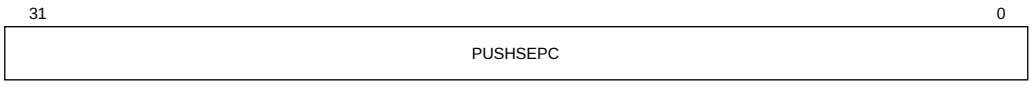


Fig. 16.19: pushsepc register

The *pushsepc* register is a function CSR and it can't be read or written as a normal CSR. Any CSR instruction except for csrrwi accessing this register will raise an illegal instruction exception.

The *pushsepc* register is designed to speed up the context saving of S-mode interrupt. Using csrrwi instruction, the data of *sepc will be pushed* to memory address = sp + imm*4.

```
csrrwi x0, pushsepc ,2 // push sepc to memory address = sp + 8
```

16.4 TEE Interrupt Operation

This chapter describes the behavior of TEE interrupts, especially for supervisor-level interrupts. A supervisor-level interrupt is taken only when the current mode is supervisor mode and the s*status*.SIE field is set or when the current mode is user mode. Several main changes in the TEE will be mentioned in the following sections.

16.4.1 ECLIC Memory Map

The ECLIC memory map has been changed to support the TEE. The ECLIC memory map can support up to 1,024 total interrupt inputs, and an M-mode ECLIC Region and a S-mode ECLIC Region. *ECLIC Modified Memory Map* (page 77) describes the detailed memory map scheme.

Table 16.8: ECLIC Modified Memory Map

Offset	R/W	Name	Width(bits)	Description
0x0000	RW	cliccfg	8	-
0x0004	R	clicinfo	32	-
0x0008	RW	mintthresh	32	-
0x1000+4*i	RW	clicintip[i]	8	M-mode ECLIC Region
0x1001+4*i	RW	clicintie[i]	8	
0x1002+4*i	RW	clicintattr[i]	8	
0x1003+4*i	RW	clicintctl[i]	8	
0x2008	RW	sintthresh	32	-
0x3000+4*i	RW	clicintip[i]	8	S-mode ECLIC Region
0x3001+4*i	RW	clicintie[i]	8	
0x3002+4*i	RW	clicintattr[i]	8	
0x3003+4*i	RW	clicintctl[i]	8	

Note:

- The above "i" indicates the interrupt ID, an interrupt *i* has its own corresponding clicintip[i], clicintie[i], clicintattr[i], and clicintctl[i] registers.
- Aligned byte, half-word, and word accesses to ECLIC registers are supported.
- The above "R" means read-only, and any write to this read-only register will be ignored without bus error.
- If an input *i* is not present in the hardware, the corresponding clicintip[i], clicintie[i], clicintctl[i] memory locations appear hardwired to zero.

16.4.1.1 M-mode ECLIC Region

This region is designed to support M-mode access. If an input *i* is not present in the hardware, the corresponding *clicintip[i]*, *clicintie[i]*, *clicintctl[i]* memory locations appear hardwired to zero.

16.4.1.2 S-mode ECLIC Region

Supervisor-mode ECLIC regions only expose interrupts that have been configured to be supervisor-accessible via the M-mode CLIC region. System software must configure PMP and sPMP permissions to make sure this region can only be accessed from appropriate Supervisor-mode codes.

Any interrupt *i* that is not accessible to S-mode appears as hard-wired zeros in *clicintip[i]*, *clicintie[i]*, and *clicintctl[i]*.

If *clicintattr [i]* is set to S-mode (bit 7 is clear, and bit 6 is set), then interrupt *i* is visible in the S-mode region except that only the low 6 bits of *clicintattr [i]* can be written via the S-mode memory region.

16.4.2 ECLIC Modified Memory Mapped Registers

Following only describes the ECLIC modified memory mapped registers.

16.4.2.1 cliccfg

This *cliccfg* register is a global configuration register. *cliccfg bit assignments* (page 78) describes the bit assignments of this register.

Table 16.9: cliccfg bit assignments

Field	Bits	R/W	Reset Value	Description
Reserved	7	R	N/A	Reserved, ties to 0.
nmbits	6:5	R	N/A	nmbits ties to 1, indicates supervisor-level interrupt supporting.
nlbits	4:1	RW	0	nlbits specifies interrupt level and priority.
Reserved	0	R	N/A	Reserved, ties to 1.

16.4.2.2 mintthresh

The *mintthresh* register holds the current threshold level for each privilege mode (i.e., mth, sth). *mintthresh bit assignments* (page 78) describes the bit assignments of this register.

Table 16.10: mintthresh bit assignments

Field	Bits	R/W	Reset Value	Description
mth	31:24	RW	0	Interrupt-level threshold for M-mode.
Reserved	23:16	R	N/A	Reserved, ties to 0.
sth	15:8	RW	0	Interrupt-level threshold for S-mode.

continues on next page

Table 16.10 – continued from previous page

Field	Bits	R/W	Reset Value	Description
Reserved	7:0	R	N/A	Reserved, ties to 0.

16.4.2.3 sintthresh

The *sintthresh* register holds the current threshold level for Supervisor mode. *sintthresh bit assignments* (page 79) describes the bit assignments of this register.

Table 16.11: sintthresh bit assignments

Field	Bits	R/W	Reset Value	Description
Reserved	31:24	R	N/A	Reserved, ties to 0.
Reserved	23:16	R	N/A	Reserved, ties to 0.
sth	15:8	RW	0	Interrupt-level threshold for S-mode.
Reserved	7:0	R	N/A	Reserved, ties to 0.

Note: here sth is a mirror to mintthresh.sth and will be updated synchronously.

16.4.2.4 clicintattr[i]

This register specifies various attributes for each interrupt. *clicintattr[i] bit assignments* (page 79) describes the bit assignments of this register.

Table 16.12: clicintattr[i] bit assignments

Field	Bits	R/W	Reset Value	Description
mode	7:6	M-mode: RW S-mode: R	3	Specifies in which privilege mode the interrupt should be taken: 3: Machine Mode 1: Supervisor Mode Note: M-mode can Read/Write this field, but S-mode can only Read this field. So ECLIC with TEE does not reply on CSR *mideleg* to delegate interrupts. **It can only write 1 or 3 to this field, other values are ignored.**
Reserved	5:3	R	N/A	Reserved, ties to 0.
trig	2:1	RW	0	Specifies the trigger type and edge polarity for each interrupt input.
shv	0	RW	0	Specifies hardware vectoring or non-vectoring mode.

16.4.3 ECLIC Interrupt Arbitration

As mentioned above, the *clicintattr[i]*.mode field is implemented to specify in which privilege mode the interrupt should be taken. In order to support the TEE, the arbitration logic needs to take the *clicintattr[i]*.mode into the consideration. Hence, the winner of the arbitration is determined by interrupts' mode, level and priority. In the modified arbitration logic, privilege mode priors to level and priority, and level priors to priority. *Modified arbitration scheme* (page 80) shows the modified arbitration scheme. For each interrupt *i*, its privilege mode, level and priority are combined to represent an unsigned integer number. The interrupt that has the greatest number wins the arbitration.

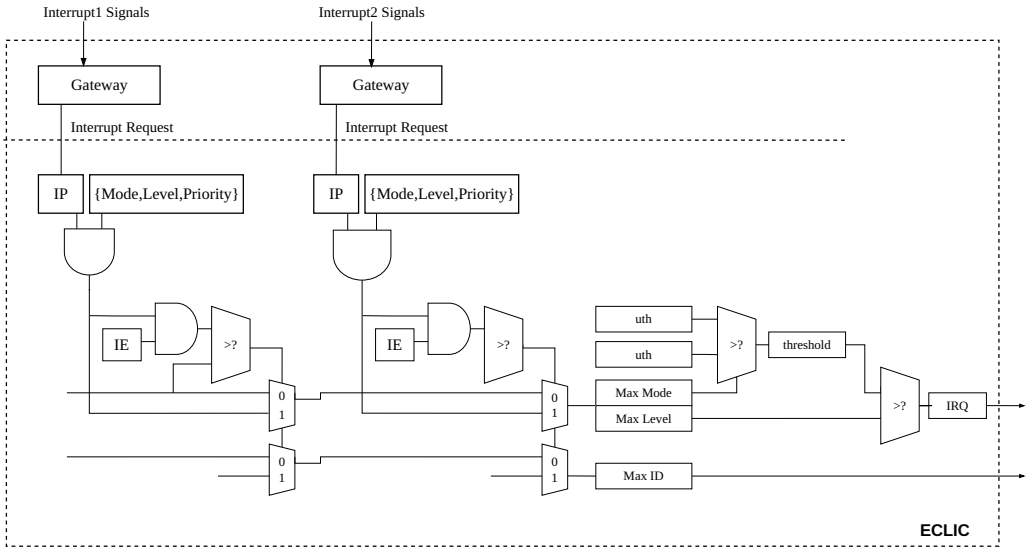


Fig. 16.20: Modified arbitration scheme

16.4.4 Supervisor-Level Interrupt Flow

This section describes the overall behavior when a supervisor-level interrupt is taken.

16.4.4.1 Enter a Supervisor-Level Interrupt Handler

If a supervisor-level interrupt wins the arbitration and is taken (it can only happen in supervisor/user mode), the following listed hardware behaviors will happen at the same time:

- Stop executing the current program, and jump to a new PC, then execute.
- Update the following listed CSRs at the same time:

  sepc

  sstatus

  scause

  sintstatus

- The privilege mode changes to supervisor mode.
- *General flow of entering a supervisor-level interrupt handler* (page 81) shows the general flow of entering a supervisor-level interrupt handler.

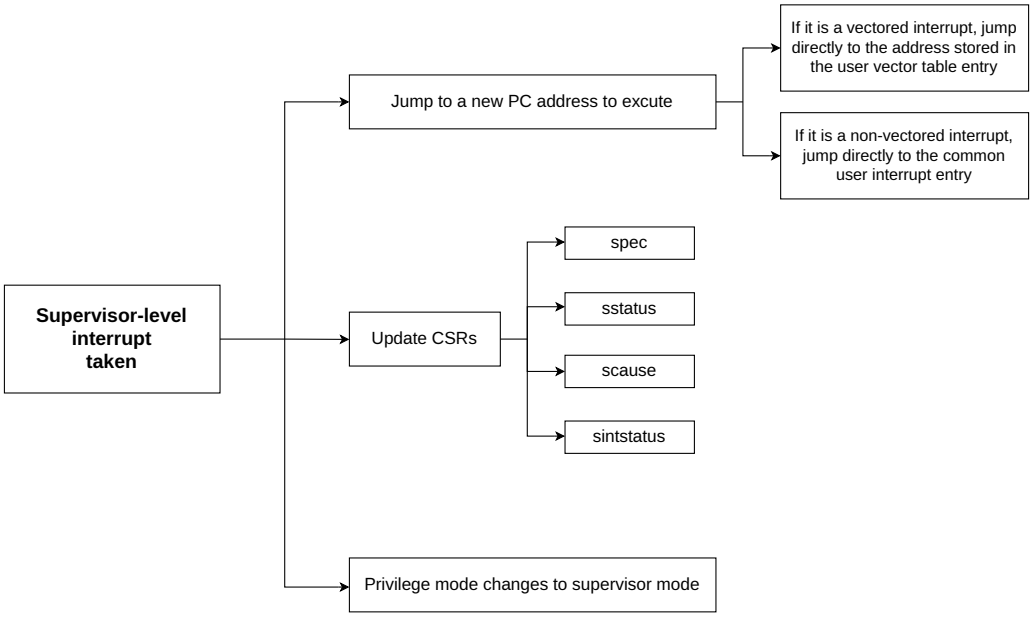


Fig. 16.21: General flow of entering a supervisor-level interrupt handler

16.4.4.1.1 Jump to a New PC To Execute

Each interrupt can be configured to vectored or non-vectored (through *clicintattr[i]*.shv).

- If the interrput is vectored, the core will jump directly to the address stored in the supervisor vector table entry, which is specified by *stvt*.
- If the interrput is non-vectored, the core will jump directly to the common supervisor interrupt entry, which is specified by *stvt2*.

16.4.4.1.2 Update CSR *sepc*

When a supervisor-level interrupt is taken, the sepc will be updated to the interrupted PC. In this way, once returning from this supervisor-level interrupt handler, the core can restore the PC from *spec*.

16.4.4.1.3 Update CSR *scause*

The *scause* will be updated as follows:

- The *scause*.EXCCODE field will be updated to indicate the current interrupt ID.
- The *scause*.SPIL field will be updated to indicate the interrupted interrupt level(ie, *sintstatus*.SIL). In this way, once returning from supervisor-level interrupt handler, *sintstatus*.SIL can restore is interrupted value from *scause*.SPIL.
- If the interrupt is vectored, the *scause*.SINHV field will be updated to indicate its hardware vectoring state.

16.4.4.1.4 Update CSR *sstatus*

The *sstatus* will be updated as follows:

- The *sstatus*.SPIE field will be updated to the value of *sstatus*.SIE field, while *sstatus*.SIE field will be updated to zero.
- Note: The *scause*.SPIE field mirrors the *sstatus*.SPIE field, and is aliased into *scause* to reduce context save/restore code.

16.4.4.1.5 Update CSR *sintstatus*

The *sintstatus* will be updated as follows:

- The *sintstatus*.SIL field will be updated to hold the active interrupt level.

16.4.4.1.6 Privilege Mode Changed to Supervisor Mode

A supervisor-level interrupt can be taken from supervisor/user mode, after being taken the privilege mode will change to supervisor mode.

16.4.4.2 Return from a Supervisor-Level Interrupt Handler

The regular sret instruction is used to return from a supervisor-level interrupt handler. After the execution of sret, the following listed hardware behaviors will happen at the same time.

- Stop executing the current program, and jump to a new PC stored in sepc to execute.
- Update the following listed CSRs at the same time:

  sstatus

  scause

  sintstatus
- The privilege mode changes to the previous interrupted mode.
- *General flow of returning from an supervisor-level interrupt handler* (page 83) shows the general procedure.

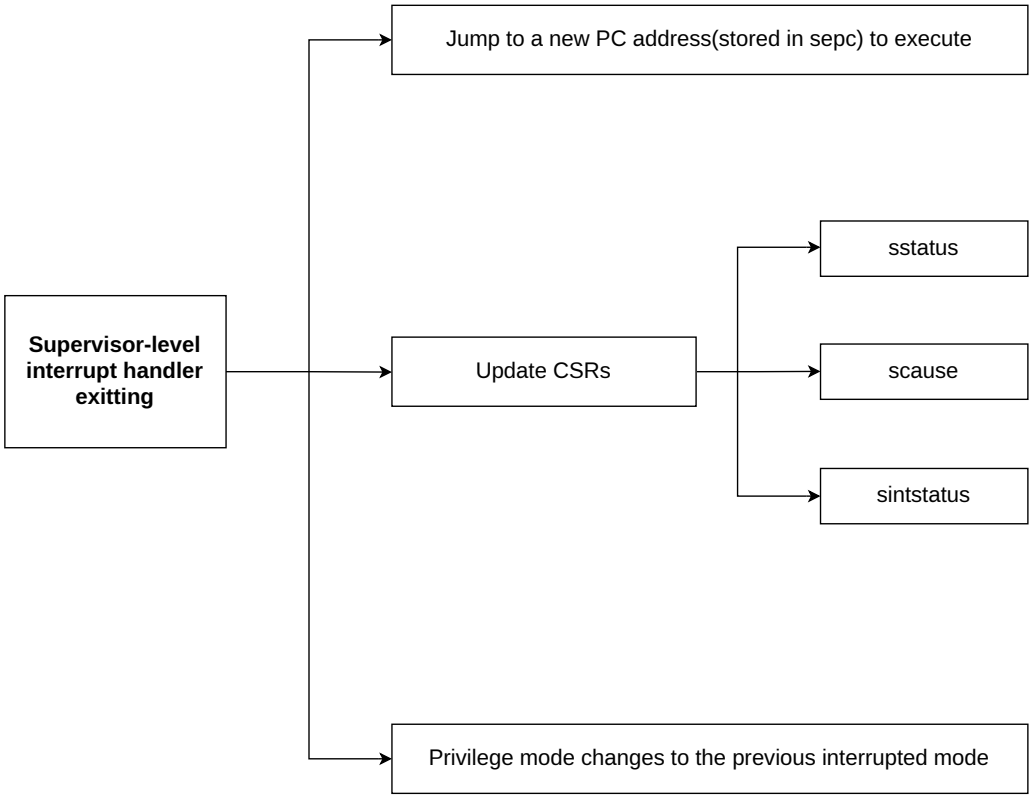


Fig. 16.22: General flow of returning from an supervisor-level interrupt handler

16.4.4.2.1 Jump to a New PC To Execute

The core will jump to a new address stored in spec to execute, after executing the sret instruction to return from a supervisor-level interrupt handler.

16.4.4.2.2 Update CSR *sstatus*

After the execution of sret, the CSR *sstatus* will be updated as follows.

- The *sstatus*.SIE field will be updated to the value of *sstatus*.SPIE field.
- The *sstatus*.SPIE field will be set to 1.

16.4.4.2.3 Update CSR *scause*

After the execution of sret, the CSR *scause* will be updated as follows.

- Note: The *scause*.SPIE field mirrors the *sstatus*.SPIE field, and is aliased into scause to reduce context save/restore code.

16.4.4.2.4 Update CSR *sintstatus*

After the execution of sret, the CSR *sintstatus* will be updated as follows.

- The *sintstatus*.SIL field will be updated to the value of the *scause*.SPIL filed.

16.4.4.2.5 Privilege Mode changed to the Previous Privilege Mode

The privilege mode will be changed to the previous privilege mode (which is stored in sstatus.SPP) when returning from this interrupt handler.

16.4.4.3 Supervisor-Level Non-vector Mode

Supervisor-Level non-vector mode interrupt flow is the same as machine-level non-vector mode flow, except:

- The CSRs that should be saved and restored during interrupt processing are different.

  During supervisor-level interrupt processing, *sepc* and *scause* should be saved and restored.

- The instruction used for tail-chaining is different.

  For supervisor-level non-vector mode, the instruction "csrrw ra, jalsnxti, ra" is used to realize supervisor-level interrupt tail-chaining.

16.4.4.4 Supervisor-Level Vector Mode

Supervisor-Level vector mode interrupt flow is the same as machine-level vector mode flow, except:

- The CSRs that should be saved and restored during interrupt processing are different.

  During supervisor-level interrupt processing, *sepc* and *scause* should be saved and restored.

- The "interrupt attribute" for C is different.

  For a supervisor-level vector mode handler, the "interrupt attribute" for C has the following syntax:

```
void __attribute__ (interrupt("supervisor")) foo (void)

{

  extern volatile int INTERRUPT_FLAG;

  INTERRUPT_FLAG = 0;

  extern volatile int COUNTER;

  #ifdef __riscv_atomic

    __atomic_fetch_add (&COUNTER, 1, __ATOMIC_RELAXED);

  #else

    COUNTER++;

```

(continues on next page)

(continued from previous page)

```
19  #endif
20
21 }
```

16.4.5 Nesting between Privilege Modes

A machine-level interrupt can pre-empt a supervisor-level interrupt which has already been taken in supervisor mode, while a supervisor-level interrupt can never pre-empt a machine-level interrupt. Besides, a supervisor-level interrupt occurring in machine mode can not be taken.

16.5 TEE Exception Operation

This chapter describes the behavior of TEE exception, especially for S-mode exception. The S-mode exception is taken only when the current mode is supervisor mode and the corresponding bit in *medeleg* is set or the current mode is user mode. On the other hand, the *medeleg* register will be ignored when the processor is in machine mode, which means any exception happens in machine mode will never be delegated.

16.5.1 TEE Exception Mask

For a processor that supports the TEE, unlike interrupts, exceptions can't be masked in any privilege mode at any time. Once there is an exception, the processor will take it in either machine or supervisor mode.

16.5.2 TEE Exception Priority

The priority of exceptions in one privilege mode is defined by exception code, that is, the exception with smaller code has a higher priority. If a S-mode exception and an M-mode exception occur at the same time in supervisor mode, the M-mode exception has a higher priority.

16.5.3 S-mode Exception Taken

When entering a S-mode exception, the hardware behavior can be briefly described as follows.

- The PC is changed to the address defined in the *stvec*.
- *scause* is updated to reflect the type of exception.
- *sepc* is updated to save the PC address where the exception happens.
- *stval* is updated to record the memory access address or instruction code.
- *sstatus.sie* is copied to *sstatus.spie* and *sstatus.sie* is cleared.

16.5.4 S-mode Exception Return

When returning from a S-mode exception trap handler, the hardware behavior can be briefly described as follows.

- The PC is restored from the address in *sepc*.
- *sstatus*.sie is copied to *sstatus*.sipe and *spie* is set.

16.5.5 S-mode Exception Nesting

Nuclei TEE does not support S-mode exception nesting with another S-mode exception. The behavior is unpredictable.

However, M-mode exception or NMI taken in S-mode exception is controllable, as the S-mode trap CSRs won't be updated.

16.6 TEE Low-Power Mechanism

16.6.1 TEE WFI Mechanism

There are some changes for WFI mechanism in the TEE. Due to the addition of S-mode interrupt, WFI can be divided into U-mode WFI, S-mode WFI and M-mode WFI.

U-mode WFI means that the processor executes wfi instruction in User Mode, then the processor enters the User-Level Low-Power Mode.

- The processor will be waked up if a S-mode interrupt wins the interrupt arbitration, and the processor will be trapped to the corresponding handler no matter *sstatus*.sie is set or not.
- The processor will be waked up if an M-mode interrupt wins the interrupt arbitration, and the processor will be trapped to the corresponding handler no matter *mstatus*.mie is set or not.

S-mode WFI means that the processor executes wfi instruction in Supervisor Mode, then the processor enters the Supervisor-Level Low-Power Mode.

- The processor will be waked up if a S-mode interrupt wins the interrupt arbitration, and whether the processor will resume execution or be trapped to the interrupt handler depends on *sstatus*.sie.
- The processor will be waked up if an M-mode interrupt wins the interrupt arbitration, and the processor will be trapped to the corresponding handler no matter *mstatus*.mie is set or not.

M-mode WFI means that the processor executes wfi instruction in Machine Mode, then the processor enters the Machine-Level Low-Power Mode.

- Any S-mode interrupts cannot win the interrupt arbitration when the processor stays in the Machine-Level Low-Power mode, so the processor will still stay in Low-Power mode when it encounters a S-mode interrupt.
- The processor will be waked up if an M-mode interrupt wins the interrupt arbitration, and whether the processor will resume execution or be trapped to the interrupt handler depends on *mstatus*.mie.

In addition, NMI and debug request can also wake up U-mode WFI, S-mode WFI and M-mode WFI.

16.6.2 TEE WFE Mechanism

The WFE mechanism for a core with the TEE remains unchanged. No matter what the privilege mode is, setting *wfe* and executing the wfi instruction will take the processor to Low-Power mode. The processor could be waked up by event, NMI, and debug request.

16.7 TEE Physical Memory Protection Mechanism

In addition to PMP, Smpu (sPMP is the earlier spec) mechanism is implemented to achieve better isolation in TEE.

16.7.1 TEE PMP Mechanism

The PMP mechanism in TEE remains unchanged.

16.7.2 TEE Smpu Mechanism

The Smpu mechanism provides supervisor-mode control CSRs to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region. The Smpu values are checked after the physical address to be accessed pass PMP checks described in the RISC-V privileged spec.

Smpu checks are applied to all accesses when the hart is running in U-mode, and for load/store when the MPRV bit is set in the *mstatus* CSR and the MPP field in the *mstatus* CSR contains U-mode. Optionally, Smpu checks can also apply to S-mode accesses, in which case the Smpu values are locked to S-mode software, so that S-mode cannot change their values. Unlike PMP registers, Smpu registers can always be modified by M-mode software even when they are locked. Smpu registers can grant permissions to U-mode, which has none by default, and revoke permissions from S-mode, which has full permissions by default.

16.7.2.1 Smpu CSRs Introdution

16.7.2.1.1 Smpu CSRs List

Following *Smpu CSRs* (page 87) lists the Smpu CSRs. Like PMP, Smpu entries are described by an 8-bit configuration region and one SXLEN-bit address CSR. Some Smpu settings additionally use the address register associated with the preceding Smpu entry. The number of Smpu entries can vary by implementation, and up to 16 Smpu entries is supported.

Table 16.13: Smpu CSRs

Number	Privilege	Name	Description
0x170	SRW	smpuswitch0	Supervisor context switch register
0x171	SRW	smpuswitch1	Supervisor context switch register (**RV32 ONLY**)
0x1A0	SRW	spmpcfg0	Supervisor physical memory protection configuration
0x1A1	SRW	spmpcfg1	Supervisor physical memory protection configuration (**RV32 ONLY**)
0x1A2	SRW	spmpcfg2	Supervisor physical memory protection configuration
0x1A3	SRW	spmpcfg3	Supervisor physical memory protection configuration (**RV32 ONLY**)
0x1B0	SRW	spmpaddr0	Supervisor physical memory protection address register
0x1B1	SRW	spmpaddr1	Supervisor physical memory protection address register
...	SRW	...	...
(0x1B0+n)	SRW	spmpaddrn	Supervisor physical memory protection address register

16.7.2.1.2 spmpcfg<x>

The sPMP configuration registers are packed into CSRs in the same way as PMP does. For RV32, four CSRs, spmpcfg0~spmpcfg3, hold the configurations spmp0cfg~spmp15cfg for the 16 PMP entries, as shown in *RV32 spmpcfg<x> layout* (page 88). For RV64, spmpcfg0 and spmpcfg2 hold the configurations for the 16 PMP entries, as shown in *RV64 spmpcfg<x> layout* (page 88); spmpcfg1 and spmpcfg3 are illegal.

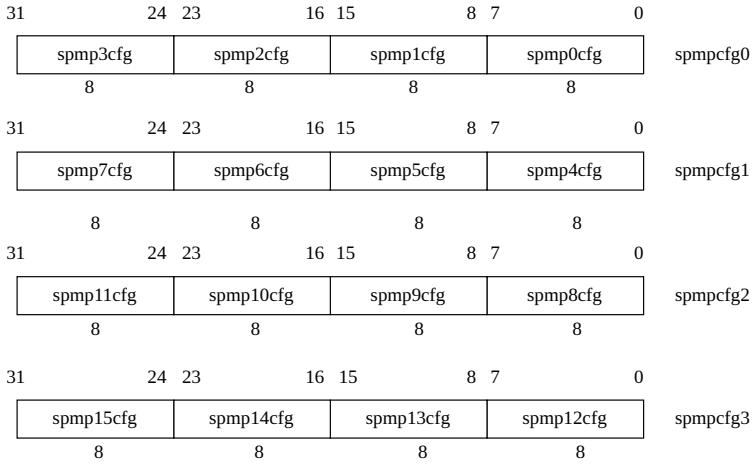


Fig. 16.23: RV32 spmpcfg<x> layout

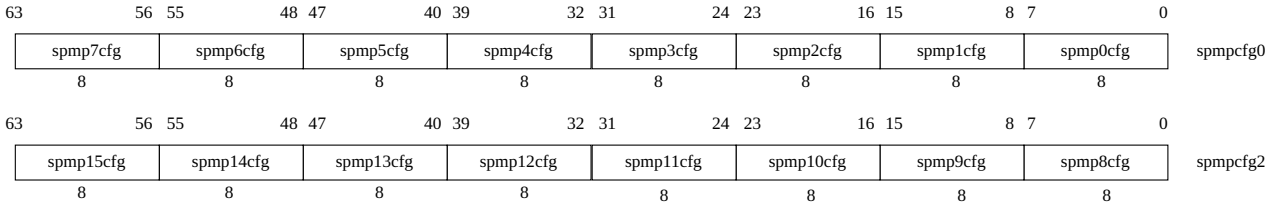


Fig. 16.24: RV64 spmpcfg<x> layout

The format of sPMP configuration registers is the same as PMP configuration registers, as is shown in *spmp<x>cfg format* (page 88).

The R, W, and X bits, when set, indicate that the sPMP entry permits read, write and instruction execution, respectively. When one of these bits is clear, the corresponding access type is denied.

The U bit represents that the sPMP entry is for user mode, and will be used to enforce SMAP and SMEP (described in *SMAP and SMEP with sPMP* (page 90)).

The A bits encodes the address-matching mode of the associated sPMP entry. The encoding of A field is the same as PMP's.

The remaining L field will be described in the following *sPMP Locking and Privilege Mode* (page 89).

Table 16.14: spmp<x>cfg format

Field	Bits	Reset Value	Description
L	7	0	Indicates the sPMP entry is locked.
U	6	0	Indicates the sPMP entry is for user mode.
Reserved	5	N/A	Reserved, ties to 0.
A	4:3	0	Encodes the sPMP entry's address-matching mode.
X	2	0	Indicates the sPMP entry permits execution.
W	1	0	Indicates the sPMP entry permits write.
R	0	0	Indicates the sPMP entry permits read.

16.7.2.1.3 spmpaddr<x>

The sPMP address registers are CSRs named spmpaddr0-spmpaddr15. Each SPMP address register encodes bits 33-2 of 34-bit physical address for RV32, as shown in *RV32 spmpaddr<x> format* (page 89). For RV64, each SPMP address bits may be implemented, and so the spmpaddr registers are WARL.

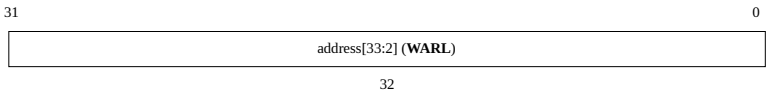


Fig. 16.25: RV32 spmpaddr<x> format

Fig. 16.26: RV64 spmpaddr<x> format

16.7.2.2 sPMP Locking and Privilege Mode

The L bit indicates that the sPMP is locked to S-mode, i.e., S-mode writes to the configuration register and associated address registers are ignored. Locked sPMP entries can only be unlocked by M-mode or by a system reset. If sPMP entry i is locked, writes to the *spmp<x>cfg* and *pmpaddr<x>* are ignored. In addition to locking the sPMP entry, the L bit indicates whether the R/W/X permissions are enforced on S-mode accesses. When the L bit is set, these permissions are enforced for both user and supervisor modes (M-mode accesses are not affected). When the L bit is clear, any S-mode access matching the sPMP entry will succeed; the R/W/X permissions apply only to U modes.

16.7.2.3 sPMP Exception

Failed accesses generate a load, store, or instruction page fault (these exceptions should be delegated to S-mode software when happen).

16.7.2.4 sPMP Priority and Matching Logic

The sPMP checks only take effect after the memory access passes the PMP permission checks. An M-mode access will not be checked by sPMP property.

Like PMP entries, sPMP entries are also statically prioritized. The lowest-numbered sPMP entry that matches any byte of an access determines whether that access succeeds or fails. The matching sPMP entry must match all bytes of an access, or the access fails, irrespective of the L, R, W, and X bits.

If a sPMP entry matches all bytes of an access, then the L, R, W and X bits determine whether the access succeeds or fails. If the privilege mode of the access is M, the access succeeds. Otherwise, if the L bit is set or the privilege mode of the access is U, then the access succeeds only if the R, W, or X bit corresponding to the access type is set.

If no sPMP entry matches an S-mode access (i.e., there is no such a sPMP entry whose L bit is set and region contains the memory access), the access succeeds, otherwise the access is checked according to the permission bits in sPMP entry. If no sPMP entry matches an U-mode access, but at least one sPMP entry is implemented, the access fails.

16.7.2.5 SMAP and SMEP with sPMP

For SMAP, the SUM (permit Supervisor User Memory access) bit in the status register is leveraged to indicate the privilege with which S-mode loads, stores, and instruction fetches access physical memory.

- When SUM=0, S-Mode physical memory accesses to memory that are accessible by

  U-Mode (U=1 in *RV32 spmpcfg<x> layout* (page 88)) will fault.

- When SUM=1, these accesses are permitted.

The SUM can take effect even when page-based virtual memory is not in effect.

For SMEP, it is not allowed for S-mode to execute codes in physical memory that are for

U-mode (U=1 in *RV32 spmpcfg<x> layout* (page 88)).

16.8 TEE Configuration

<xxx>_CFG_HAS_TEE is a global option to configure the implementation of TEE.

17 MMU Introduction

17.1 MMU Overview

Nuclei processor U/UX class core can have MMU (Memory Management Unit) configured for Linux capable applications, which implements the address translation defined in RISC-V privileged specification, to support Page-Based Virtual-Memory System, which can be used for converting Virtual-Address to Physical-Address and corresponding Permission Checking. MMU is part of RISC-V standard privileged architecture specification.

17.2 Support features

MMU support below official features.

- Sv32
- Sv39
- Sv48
- Svvptc Extension v1.0
- Svpbmt Extension v1.0
- Svnapot Extension v1.0

17.3 MMU Specific Features to Nuclei Core

In order to simplify the hardware implementation, Nuclei processor core have some MMU specific features, which are detailed described at next sections.

17.3.1 TLB

MMU has two level TLB (Translation Lookaside Buffer) implemented to cache the page tables for fast subsequent accessing:

- MTLB: main/joint TLB for both instruction and data page table
- I/D-TLB: I/D-TLB is dedicate for instruction/data page table, each has 8/16 entries

I/D-TLB will be accessed first, if miss then MTLB will be accessed.

MMU supports 4KB, 2MB, 1GB(Sv39) and 512GB(Sv48) page types, which uses Hardware Page Table Walk mechanism to fetch page tables from memory when TLB miss without software handling.

To increase the timing, PMA and PMP information are saved into I/D-TLB. So if software change:

- The PMA information: Modify PMA CSR
  - mattri(n)
  - mppicfg_info
  - ilm_ctl
  - dlm_ctl
  - mcache_ctl
  - mseccfg
  - mmacro_ca_en/mmacro_dev_en/mmacro_noc_en
- PMP information: Modify PMP CSR.

Hardware will auto clear the I/D-TLB information.

Warning: If software change the PMP for page-self memory region, Software should execute "sfence.vma x0, x0" to clear the all TLB information.

17.3.2 ASID

TLB Support ASID, and ASID width is 16.

17.3.3 ECC

MTLB Support ECC, and ECC feature use CSR(mtlbcfg_ctl) to enable or disable.

17.3.4 NAPOT

MMU Support NAPOT, but for performance, hardware add one software switch to enable it.

If NAPOT bit in mtlb_ctl is 0, hardware disable the napot translation. If 63 bit in page table entry is 1, page fault will be generated.

17.3.5 U32

Typically, RV64 CPU execute the instruction with the whole 64-bit register, and can't execute the rv32-only instructions. So RV32 application can't run in RV64 CPU.

When config U32 feature, in user-mode. RV64 CPU execute the instruction:

- Use the low 32-bit register.
- Allow execute the rv32-only instructions.
- Forbid execute the rv64-only instructions.
- Use the SV32 to address translation.

So RV32 application can run in RV64 CPU in user-mode.

Software set the UXL bit of mstatus/sstauts CSR to control this feature.

- Write 1: U-mode working in XLEN is 32.
- Write 2: U-mode working in XLEN is 64.

U32 support isa:

- I
- M

- A
- F
- D
- C
- B
- K
- V
- Zc
- Xlcz
- Zicond
- Zfh

17.3.6 Consistency

MMU page data is consistency with Dcache, that mean software write page into dcache, MMU can get the new page in dcache when page table walk.

So if software known updated page information not in MMU TLB, software no need to flush dcache data to next level, and can access the page address immediately. MMU can get the new mapping.

For example:

- Software access 0xa000_0000, and get page fault.
- Software update the page in exception handler.
- Software back to access 0xa000_0000 code.
- MMU translate the 0xa000_0000, and get page from Dcache.
- MMU get new mapping PA and permissions.

18 Secure States

Nuclei CPU implement the Smwg extention, please see *WorldGuard Specification* (page 360) to get detail information.

CPU will have two states.

- Secure state
- Non-Secure state

Software in Machine privilege can control the CSR to set the state for S/U privilege.

Note: Machine privilege is always secure state.

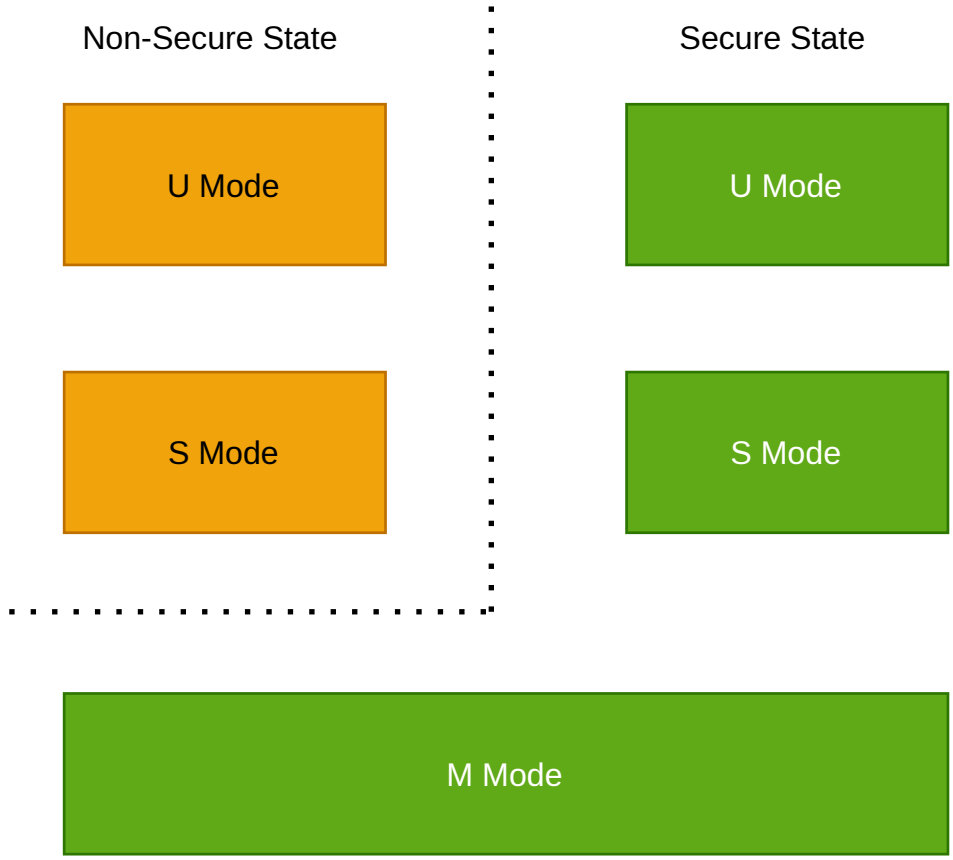


Fig. 18.1: CPU states

18.1 CSR Support

Software use mlwid(0x390) csr to set the state for S/U privilege.

Warning: This csr only can be accessed by machine privilege.

Below is the code to set secure state and non-secure state.

```
set_secure:
      li a0, 1
      csrs mlwid, a0
      ret
set_non_secure:
      li a0, 1
      csrc mlwid, a0
      ret
```

In M-mode, after setting this bit to configure the secure state, executing mret to return to S-mode or U-mode will result in the corresponding CPU transitioning to the specified secure state.

18.2 BUS Support

The arprot[1]/awprot[1] for axi bus, define the access is secure access or non-secure access.

- 0: secure access
- 1: non-secure access

The hprot[1] for ahbl bus, define the access is secure access or non-secure access.

- 0: secure access
- 1: non-secure access

18.3 Debug Support

Debug mode is split into secure debug mode and non-secure debug mode.

- Secure debug mode:
  - CPU from secure state enter debug mode
  - Debugger can access all sources.
- Non-secure debug mode:
  - CPU from non-secure state enter debug mode
  - Debugger can only access non-secure state sources.

There is a cpu input “dbg_sec_enable” to define the secure debug enable.

- 0: Disable secure debug
- 1: Enable secure debug

When secure debug disable, debugger send debug request to cpu, and cpu is running in secure state, cpu will ignore the debug request until cpu enter non-secure state.

When cpu enter non-secure state, cpu will acknoledge the debug request then enter non-secure debug mode.

Note: SBA(System Bus Access) is not support when Core support secure state.

18.4 MMU Support

Within the Instruction Translation Lookaside Buffer (ITLB), Data Translation Lookaside Buffer (DTLB), and Main Translation Lookaside Buffer (MTLB), a secure bit is maintained for every TLB entry. This bit indicates whether the entry pertains to a secure or non-secure context.

- When the system is in a secure state, TLB entries marked as secure can be matched and accessed.
- Conversely, when operating in a non-secure state, only TLB entries flagged as non-secure are eligible for matching.

In M-mode, upon changing the security state and updating the Supervisor Address Translation Physical register (satp CSR), there is no requirement for software to manually invalidate the TLB through an sfence instruction.

The invalidation of TLB entries occurs automatically during the context switch, based on their associated security states. This ensures that the TLB reflects the correct security level for the current execution context without explicit intervention from software.

Note: Secure/Non-secure bit is not exist in the page, so software should use different memory region to put secure page and non-secure page.

And software should ensure non-secure program can't access secure page.

When page table walk, mmu send below memory access to get page value:

- secure memory access when CPU is running in secure state.
- non-secure memory access when CPU is running in non-secure state.

18.5 Cache Support

Within the instruction cache (icache), data cache (dcache), and Level 2 (L2) cache, a secure bit accompanies each cacheline to denote its security states. This ensures that:

- When the processor is operating in a secure state, it can only retrieve data from cachelines that are also marked as secure.
- Conversely, while in a non-secure state, access is limited to cachelines that have been designated as non-secure.

When executing in M-mode, the process of altering the security state and updating the Supervisor Address Translation Physical (satp) Control and Status Register (CSR) does not necessitate a manual Cache Coherence Maintenance (CCM) operation to clear the cache. Instead, the invalidation of cache entries is handled automatically during a context switch, contingent upon their respective security states. This mechanism ensures that the cache contents align with the current security context without requiring explicit cache flushing instructions from software.

18.6 Sharing data Support

A set of Memory Attribute Tag Registers (MATTRIs), known as mattribute CSRs, has been introduced to enable data sharing mechanisms between secure and non-secure operational states.

These mattr CSRs serve the purpose of delineating specific segments within the non-secure address space. By doing so, they permit direct access to these segments from the secure state, effectively bridging the gap and allowing for seamless data exchange across different security contexts.

These accessible segments within the non-secure space are metaphorically termed as "skylights", signifying a window of opportunity for secure state entities to interact with non-secure resources without compromising the overall security posture.

Warning: Skylights only support data sharing, not code sharing.

In total, there are five distinct sets of mattribute CSRs designed to cater to various requirements and configurations in managing the inter-state data sharing efficiently and securely. Each set provides granular control over attributes such as memory type, cacheability, and access permissions, ensuring that data sharing is both flexible and secure across different execution domains.

CSR ID	Attri	CSR Name	Description
0x7f3	MRW	mattri0_base	entry0 base
0x7f4	MRW	mattri0_mask	entry0 mask
0x7f5	MRW	mattri1_base	entry1 base
0x7f6	MRW	mattri1_mask	entry1 mask
0x7f9	MRW	mattri2_base	entry2 base
0x7fa	MRW	mattri2_mask	entry2 mask
0x7fb	MRW	mattri3_base	entry3 base
0x7fc	MRW	mattri3_mask	entry3 mask
0x7fd	MRW	mattri4_base	entry4 base
0x7fe	MRW	mattri4_mask	entry4 mask

Warning: The base address should anligned to 4K.

The base register shows below:

bit	Attri	Description
0	MRW	**Entry Enable** 0: Disable 1: Enable If this bit is 0, other bit will be ignored by hardware
1	MRW	Device Attribute Enable
2	MRW	Non cacheable Attribute Enable
3	MRW	Share Attribute Enable
11:4		Reserved
PA_SIZE:12	MRW	Base address

For example, set the 0xa000_0000 to 0xafff_ffff to share region, the code is below:

```
li a0, 0xf0000000
csrw mattri4_mask, a0
li a0, 0xa0000000 + (1 << 3) + (1 << 0)
csrw mattri3_base, a0
```

In order to facilitate the setup of shared memory regions under the secure state of Supervisor Mode (S-mode), the following Control and Status Registers (CSRs) have been added.

CSR ID	Attri	CSR Name	Description
0x5f0	S-SRW	sattri0_base	entry0 base
0x5f1	S-SRW	sattri0_mask	entry0 mask
0x5f2	S-SRW	sattri1_base	entry1 base
0x5f3	S-SRW	sattri1_mask	entry1 mask
0x5f4	S-SRW	sattri2_base	entry2 base
0x5f5	S-SRW	sattri2_mask	entry2 mask
0x5f6	S-SRW	sattri3_base	entry3 base
0x5f7	S-SRW	sattri3_mask	entry3 mask
0x5f8	S-SRW	sattri4_base	entry4 base
0x5f9	S-SRW	sattri4_mask	entry4 mask
0x5fa	S-SRW	sattri5_base	entry5 base
0x5fb	S-SRW	sattri5_mask	entry5 mask
0x5fc	S-SRW	sattri6_base	entry6 base
0x5fd	S-SRW	sattri6_mask	entry6 mask
0x5fe	S-SRW	sattri7_base	entry7 base
0x5ff	S-SRW	sattri7_mask	entry7 mask

Warning: These CSR only can accessed by M-mode or Secure state S-mode.

The base register shows below:

bit	Attri	Description
0	MRW	**Entry Enable** 0: Disable 1: Enable If this bit is 0, other bit will be ignored by hardware
1		Reserved
2		Reserved
3	MRW	Share Attribute Enable
11:4		Reserved
PA_SIZE:12	MRW	Base address

The configured shared regions only apply to the S and U modes, they have no effect on the M mode.

18.7 Not Support Secure State

There are some memory region not support secure state.

- ILM
- DLM
- Iregion device
  - Timer
  - Eclic
  - Plic
  - …
- Cluster local memory

The aforementioned regions do not differentiate between secure and non-secure states.

18.8 Linux Boot

In the existing RISC-V Linux boot process, OpenSBI is tasked with transferring the kernel and root filesystem data into DDR. Both the kernel and root filesystem run in Non-Secure mode, while OpenSBI operates in Secure mode. This leads to a situation where, post data movement by OpenSBI, the relocated data becomes unreachable when the CPU transitions to Non-Secure mode (part of the transferred data may be located in the cache, which is differentiated between Secure and Non-Secure states).

To resolve this issue, it is required to establish a 'skylights' prior to the data transfer, designating the data as shared between Secure and Non-Secure states. Upon completion of the data movement, this skylights needs to be subsequently closed.

If the DDR address for the data relocation is 0xa000_0000, and the size of the data to be moved is 256MB, then the relocation code should be modified as follows:

```
// setup skylights
li a0, 0xf0000000
csrw mattri3_mask, a0
li a0, 0xa0000000 + (1 << 3) + (1 << 0)
csrw mattri3_base, a0

// move code/data

// close skylights
csrw mattri3_base, x0
```

Note: Regarding the Device Tree Blob (DTB) content, it also needs to be relocated to the non-secure region; otherwise, once the 'skylights' is closed, the kernel will be unable to access the DTB content.

19 PMA Introduction

PMA(Physical memory attribute) define the attribute of memory. PMA will affect CPU access memory behavior.

PMA are split into three attributes:

- Device(abbreviated as 'DEV') Attribute
- Non-Cacheable(abbreviated as 'NC') Attribute
- Cacheable(abbreviated as 'CA') Attribute

And the whole memory region are split into three regions:

- DEV Region: The PMA of this region is DEV attribute.
- NC Region: The PMA of this region is NC attribute.
- CA Region: The PMA of this region is CA attribute.

Note: Only CPU config I-Cache, the CA attribute and CA region are exist. Otherwise DEV and CA exist.

19.1 CPU Fetch

CPU can't fetch instruction from device region. If ifu find the fetch address is DEV region, it don't send fetch request to soc, and generate exception immediately.

If ifu find the fetch address is NC region, it send fetch request to SOC. When get instruction, it send instruction to exu immediately, don't save the instruction into I-Cache.

If ifu find the fetch address is CA region, it send fetch request to SOC. When get instruction, it send instruction to exu and save the instruction into I-Cache.

19.2 CPU Load/Store

The behavior of CPU load/store data in:

- Device Region:
  - The sequence of memory accesses is guaranteed to be in-order.
  - Not support write data merge.
  - Not support early response.
  - Load data will not refill into D-Cache.
  - Not Support data coherency in smp.

- NC Region:
  - The sequence of memory accesses is guaranteed to be in-order.
  - Support write data merge.
  - Support write early response.
  - Load data will not refill into D-Cache.
  - Not Support data coherency in smp.
- CA Region:
  - The sequence of memory accesses is not guaranteed to be in-order.
  - Support write data merge.
  - Support write early response.
  - Load data will refill into D-Cache.
  - Support data coherency in smp.
  - Support data prefetch.

19.3 Hardware PMA Setting

Hardware provide three group config to set the PMA.

- DEV Region config
- NC Region config
- CA Region config

Each group config have three config:

- REGION_NUM: specify the active region number, range is 0-8.
- REGION(n)_BASE: specify the group n base address.
- REGION(n)_MASK: specify the group n mask address.

REGION Size = ~mask + 1

For one address A, if A & REGION(n)_MASK == REGION(n)_BASE, that mean the attribute of address A belong the region attribute.

Note: Region granule is 4KB, so the low 11-bit of MASK must be 0.

19.4 Software PMA Setting

Hardware provide some software csr to modify the pma. Please refer to *mattrib(n)* (page 135) and *mattrim(n)* (page 136) to knwo how to set.

19.5 Priority PMA

PMA setting can overlap, if overlap, the Priority order is as belows:

- IREGION/PPI/CPPI/FIO
- Hardware/software NC
- Hardware/software CA
- Hardware/software DEV

For example:

one address is in IREGION, the pma of the address is always device. Hardware or software can't modify it.

one address is in hardware dev region, the PMA of the address is device attribute. Software can modify it to NC or CA attribute.

19.6 ILM/DLM/VLM PMA

The PMA of ILM, DLM and VLM is determined by hardware pma config. But software can modify it.

Warning: For correctness, the PMA of ILM can't set device attribute.

For efficiency, the PMA of DLM and VLM shouldn't set device attribute.

19.7 CLM PMA

The PMA of CLM is determined by hardware PMA config. But software can modify it.

Note: For efficiency, the PMA of CLM shouldn't set device attribute.

20 WFI/WFE Low-Power Mechanism

The Nuclei processor core can support sleep mode for lower power consumption.

20.1 Enter the Sleep Mode

The Nuclei processor core can enter sleep mode by executing the WFI instruction. When the core executes the WFI instruction, it will perform following operations:

- Stop executing the current instruction stream immediately.
- Waiting for the core to complete any outstanding transactions, such as fetching instructions, load or store operations, to ensure that all the transactions sent to the bus are completed.

Note: If a memory access error exception occurs while waiting for a bus operation to complete, the core will enter the exception handling mode rather than sleep mode.

- When all of the outstanding transactions are completed, the core safely enters an idle state, which is called the sleep mode.
- When enters the sleep mode:
  - The clocks of the main units inside the core will be gated off to save dynamic power consumption;
  - The output signal core_wfi_mode of the core will be asserted to indicate that this core is in the sleep mode;
  - The output signal core_sleep_value of the core will output the value of the CSR register sleepvalue

---

Note:

- This signal core_sleep_value is valid only when the core_wfi_mode is asserted; if the signal core_wfi_mode is 0, then the value of core_sleep_value must be 0. The software can indicate different sleep modes (0 as shallow sleep or 1 as deep sleep) by set the CSR register sleepvalue in advance.
- The Nuclei processor core behaves exactly the same for different sleep modes. These sleep modes only provide different output value (via output signal core_sleep_value) for different controlling scheme to the Power Management Unit (PMU) at the SoC system level. For example, if the PMU detect the core_wfi_mode is high and core_sleep_value is low, then it can switch the core_clk_aon to a lower frequency; if the PMU detect the core_wfi_mode is high and core_sleep_value is high, then it can turn off the core_clk_aon, and in this condition, it must restore core_clk_aon before interrupt/event wakes core up.
- When core enters to the deep sleep mode, normally the processor core will no longer be able to be debugged, and Nuclei Core has input signal *override_dm_sleep* to avoid this, please refer the signal description in related core's databook.

- If the core is one in a cluster (like UX900MP), and the core's SMP Enable Bit (*SMP_ENB* (page 337)) is on, then while the core is in Sleep Mode, it can be snooped by other clients in the cluster automatically, that means the core's clock will be on and handle the snoop request ,and then the clock will be off right after the request is done.

20.2 Wait for Interrupt

The Wait for Interrupt mechanism means that the core enters the sleep mode, and the core keeps on waiting for an interrupt to wake up.

As described in *Enter the Sleep Mode* (page 103), the Wait for Interrupt mechanism can be implemented with following programming flow:

1. Set the value of CSR sleepvalue to 0 or 1.
2. Set the value of CSR wfe.WFE to 0.
3. Execute the WFI instruction.

20.3 Wait for Event

The Wait for Event mechanism means that the core enters the sleep mode, and the core keeps waiting for an event to wake up. When the core wakes up by the event, it continues to execute the instruction right after the wfi.

As mentioned in *Enter the Sleep Mode* (page 103), the Wait for Event mechanism can be implemented by executing the WFI instruction combined with the following sequence of instructions:

1. Set the value of CSR sleepvalue to 0 or 1.
2. Set the value of CSR wfe.WFE to 1.
3. Execute the WFI instruction..

20.4 Exit the Sleep Mode

The core can exit the Sleep Mode (be woken up) by four ways:

- NMI
- Interrupt
- Event
- Debug request

Each way is described in details in next part. When core exits the Sleep Mode, the output signal core_wfi_mode of the core is cleared to 0.

20.4.1 Wake Up by NMI

NMI can always wake up the core. When the core detects a rising edge of the input signal nmi, the core is woken up and jumps to the NMI service routine.

20.4.2 Wake Up by Interrupt

Interrupts can wake up the core as well:

- If the value of wfe.WFE is set to 0, the core will be waited for the interrupt to wake up. In this case, the behavior of WFI wake up is just following the RISC-V standard architecture. This document will not repeat its content here, please refer to RISC-V standard privileged architecture specification for more details.
- If the value of wfe.WFE is set to 1, the core will be waited for an event to wake up. Please see the detailed description in the next section.

20.4.3 Wake Up by Event

Event can wake up the processor core when the following conditions are met:

- If the value of wfe.WFE is set to 1, then:
  - When the core detects that the input signal rx_evt (called the event signal) is asserted, the core will be woken up and continue to execute the previously interrupted instruction stream. Please refer to the specific databook of the Nuclei processor core for details about the signal rx_evt.

20.4.4 Wake Up by Debug Request

Debug requests can wake up the core. If the debugger is connected, it wakes up the core and lets core enter the debug mode.

21 Power Down Low-Power Mechanism

Last chapter introduces Nuclei processor core WFI/WFE mechanism, it uses clock off to save power consumption. Further to save power consumption, power off the logic is an option, this chapter introduces that the programming flow of power down Nuclei processor core.

21.1 Single Core Power Down Flow

As Nuclei N200/N300/N600/NX600/UX600 core is Single Core Configurations only, including 900 Single Core, it has only one power domain. The programing model/flow to power down is as following:

- Core finishes the application task, mask all interrupts or disable global interrupt.
- If there is L1-DCache, core flushes all L1-DCache and waits till it is done (refer csr ccm_pipe in CCM document).
- Core sets the value of CSR sleepvalue to 1, then execute the WFI instruction.
- The PMU of SoC observers the core's core_wfi_mode is 1 and the core_sleep_value is 1, then it can turn off all this core's input clocks and then turn off the core's power.

Note:

1. If user needs to save the status of core and let core to power on from the point where core powers down (called software retention), user needs to define the scope of "status" (including core status and SoC status), and implement an always on memory in the SoC, then core can save the "status" to the always on memory and leave a flag of warm reset in the memory, when core powers on , the first thing is to read the flag in the always on memory and then decide it is a warm reset or code reset, if warm reset, store the status of memory and begin to run. If user needs more consultant, please contact Nuclei Support.

21.2 Power Down One Core in the Cluster

As Nuclei 900 Series support Cluster Configuration, in the cluster it has N SMP cores, so it can only power down one core in this cluster. The programing model/flow to power down one core in a cluster is as following:

- The core finishes the application task, mask all interrupts or disable global interrupt.
- If there is L1-DCache, core flushes all L1-DCache and waits till it is done (refer csr ccm_pipe in CCM document).
- If the core has enabled smp, then core can query and wait the smp transaction done; and query and wait the smp snoop by other agents done.
- The core disable smp.
- The core sets the value of CSR sleepvalue to 1, then execute the WFI instruction.

- The PMU of SoC observers the core's core_wfi_mode is 1 and the core_sleep_value is 1, then it can turn off the core's input clocks and then turn off the core's power.

21.3 Power Down One Cluster

As Nuclei 900 Series support Cluster Configuration, it can power down the whole cluster. The programing model/flow to power down one cluster is as following:

- Define a master core in the cluster (usually core0), then all other cores in the cluster belongs to slave core (core1 ~ coren).
- All slave cores follows the flow of *Power Down One Core in the Cluster* (page 106).
- Master core can query the PMU of SoC and know that all slave cores has entered into WFI mode.
- Master core mask all interrupts/disable global interrupt , flush its all L1 D-Cache ,then flush all cluster cache and wait till it is done.
- Master core sets the value of CSR sleepvalue to 1, then execute the WFI instruction.
- The PMU of SoC observers all cores' core_wfi_mode are 1 and the core_sleep_value are 1, then it can turn off the cluster (or all core's) input clocks and then turn off the cluster's power.

22 Nuclei processor core CSRs Descriptions

22.1 CSR Overview

CSR (Control and Status Registers) is part of RISC-V standard privileged architecture. Basically, Nuclei processor core are following and compatible to RISC-V standard CSR definitions, but there are some additions and enhancements to the original standard spec.

To respect the RISC-V standard, this document may not repeat the contents of original RISC-V standard, but will highlight the additions and enhancements of Nuclei defined.

22.2 Nuclei processor core CSRs List

In this CSRs List of Nuclei processor core, there are RISC-V standard CSRs and customized CSRs. Following table only describes the customized CSRs.

Table 22.1: Customized CSRs in the Nuclei processor core

Address	R&W	Name	Description
0x307	MRW	mtvt	ECLIC Interrupt Vector Table Base Address
0x320	MRW	mcountinhibit	Customized register used to control the on & off of counters
0x346	MRO	mintstatus	Current Interrupt Level
0x348	MRW	mscratchcsw	Scratch swap register for privileged mode
0x349	MRW	mscratchcswl	Scratch swap register for interrupt mode and normal mode
0x390	MRW	mlwid	WorldGuard for security
0x5f0	S-SRW	sattri0_base	Base address of Secure Region 0 to set attribute
0x5f1	S-SRW	sattri0_mask	Mask(size) of Secure Region 0 to set attribute
0x5f2	S-SRW	sattri1_base	Base address of Secure Region 1 to set attribute
0x5f3	S-SRW	sattri1_mask	Mask(size) of Secure Region 1 to set attribute
0x5f4	S-SRW	sattri2_base	Base address of Secure Region 2 to set attribute
0x5f5	S-SRW	sattri2_mask	Mask(size) of Secure Region 2 to set attribute
0x5f6	S-SRW	sattri3_base	Base address of Secure Region 3 to set attribute
0x5f7	S-SRW	sattri3_mask	Mask(size) of Secure Region 3 to set attribute
0x5f8	S-SRW	sattri4_base	Base address of Secure Region 4 to set attribute
0x5f9	S-SRW	sattri4_mask	Mask(size) of Secure Region 4 to set attribute
0x5fa	S-SRW	sattri5_base	Base address of Secure Region 5 to set attribute
0x5fb	S-SRW	sattri5_mask	Mask(size) of Secure Region 5 to set attribute
0x5fc	S-SRW	sattri6_base	Base address of Secure Region 6 to set attribute
0x5fd	S-SRW	sattri6_mask	Mask(size) of Secure Region 6 to set attribute
0x5fe	S-SRW	sattri7_base	Base address of Secure Region 7 to set attribute
0x5ff	S-SRW	sattri7_mask	Mask(size) of Secure Region 7 to set attribute
0x7c0	MRW	milm_crtl	Enable/Disable the ILM address space

continues on next page

Table 22.1 – continued from previous page

Address	R&W	Name	Description
0x7c1	MRW	mdlm_crtl	Enable/Disable the DLM address space
0x7c2	MRW	mecc_code	ECC code injection register, can be used to simulate ECC error
0x7c3	MRO	mnvec	Customized register used to indicate the NMI handler entry address
0x7c4	MRW	msubm	Customized register storing current trap type and the previous trap type before trapped
0x7c6	MRW	mstack_ctrl	Customized register to control Stack Check function
0x7c7	MRW	mstack_bound	Customized register to store the bound of or track stack top for Stack Check.
0x7c8	MRW	mstack_base	Customized register to store the stack base for Stack Check.
0x7c9	MRW	mdcause	Customized register storing current trap's detailed cause
0x7ca	MRW	mcache_ctl	Customized register to control the Cache features
0x7d0	MRW	mmisc_ctl	Customized register controlling the selection of the NMI Handler Entry Address
0x7dd	MRW	mtlb_ctl	Customized register to control the TLB
0x7de	MRW	mecc_lock	To lock ECC configure registers, then all the ECC related CSRs cannot be modified, unless by reset
0x7e2	MRW	mfp16mode	Customized register used to set 16 bit float precision mode
0x7eb	MRW	pushmsubm	Customized register used to push the value of msubm into the stack memory
0x7ec	MRW	mtvt2	Customized register used to indicate the common handler entry address of non-vectored interrupts
0x7ed	MRW	jalmnxti	Customized register used to enable the ECLIC interrupt. The read operation of this register will take the next interrupt,return the entry address of next interrupt handler, and jump to the corresponding handler at the same time
0x7ee	MRW	pushmcause	Customized register used to push the value of mcause into the stack memory
0x7ef	MRW	pushmepc	Customized register used to push the value of mepc into the stack memory
0x7f0	MRW	mppicfg_info	PPI Configuration Information
0x7f1	MRO	mfiocfg_info	FIO Configuration Information
0x7f3	MRW	mattri0_base	Base address of Region 0 to set attribute
0x7f4	MRW	mattri0_mask	Mask(size) of Region 0 to set attribute
0x7f5	MRW	mattri1_base	Base address of Region 1 to set attribute
0x7f6	MRW	mattri1_mask	Mask(size) of Region 1 to set attribute
0x7f9	MRW	mattri2_base	Base address of Region 2 to set attribute
0x7fa	MRW	mattri2_mask	Mask(size) of Region 2 to set attribute
0x7fb	MRW	mattri3_base	Base address of Region 3 to set attribute
0x7fc	MRW	mattri3_mask	Mask(size) of Region 3 to set attribute
0x7fd	MRW	mattri4_base	Base address of Region 4 to set attribute
0x7fe	MRW	mattri4_mask	Mask(size) of Region 4 to set attribute
0x7f7	MRO	mirgb_info	IREGION Configuration Information
0x810	URW	wfe	Customized register used to control the WFE mode
0x811	URW	sleepvalue	Customized register used to indicate the WFI sleep mode
0x812	URW	txevt	Customized register used to send an event
0xbc0	MRW	mecc_ctrl	ECC Control Register
0xbc4	MRW	mecc_status	ECC Status and Control Register
0xbc8	MRW	mmacro_dev_en	Enable Bit for Dev Region setting in RTL Configuration Stage
0xbc9	MRW	mmacro_noc_en	Enable Bit for Non-Cacheable Region setting in RTL Configuration Stage
0xbca	MRW	mmacro_ca_en	Enable Bit for Cacheable Region setting in RTL Configuration Stage
0xbe0	MRW	mattri5_base	Base address of Region 5 to set attribute
0xbe1	MRW	mattri5_mask	Mask(size) of Region 5 to set attribute
0xbe2	MRW	mattri6_base	Base address of Region 6 to set attribute

continues on next page

Table 22.1 – continued from previous page

Address	R&W	Name	Description
0xbe3	MRW	mattri6_mask	Mask(size) of Region 6 to set attribute
0xbe4	MRW	mattri7_base	Base address of Region 7 to set attribute
0xbe5	MRW	mattri7_mask	Mask(size) of Region 7 to set attribute
0xdc0	SRO	shartid	HARTID for S-Mode.
0xfc1	MRO	mdcfg_info	DLM and D-Cache configuration information
0xfc0	MRO	micfg_info	ILM and I-Cache configuration information
0xfc1	MRO	mdcfg_info	DLM and D-Cache configuration information
0xfc2	MRO	mcfg_info	Processor configuration information
0x1A0	SRW	spmpcfg0	Supervisor physical memory protection configuration
0x1A1	SRW	spmpcfg1	Supervisor physical memory protection configuration
0x1A2	SRW	spmpcfg2	Supervisor physical memory protection configuration
0x1A3	SRW	spmpcfg3	Supervisor physical memory protection configuration
0x1B0	SRW	spmpaddr0	Supervisor physical memory protection address register
0x1B1	SRW	spmpaddr1	Supervisor physical memory protection address register
…	SRW	…	…
(0x1B0+n)	SRW	spmpaddrn	Supervisor physical memory protection address register
0x107	SRW	stvt	Supervisor Trap-handler vector table base address
0x146	SRW	sintstatus	Supervisor current interrupt level
0x148	SRW	sscratchcsw	Scratch swap register for multiple privilege modes
0x947	SRW	jalsnxti	Jumping to next supervisor interrupt handler address and interrupt-enable register
0x948	SRW	stvt2	ECLIC non-vectored supervisor interrupt handler address register
0x949	SRW	pushscause	Push scause to stack
0x94a	SRW	pushsepc	Push sepc to stack
0x9c0	SRW	sdcause	Detail information for scause

Note:

- MRW: Machine Mode Readable/Writeable.
- MRO: Machine Mode Read-Only.
- URW: User Mode Readable/Writeable.
- URO: User Mode Read-Only.
- S-SRW: Secure State S Mode Readable/Writeable

22.3 Accessibility of CSRs in the Nuclei processor core

The CSRs Accessibility in the Nuclei processor core:

- Read or write to a non-existent CSR will raise an Illegal Instruction Exception.
- Write to RO CSRs will raise an Illegal Instruction Exception.
  - Note: According to the RISC-V standard, the URO registers like cycle, cycleh, time, timeh, instret, instreth are special, the read accessibility of which are determined by the corresponding field in mcounteren.
- Access the higher privilege mode CSR raise an Illegal Instruction Exception. For example, in User Mode accessing to MRO or MRW CSRs will raise an Illegal Instruction Exception.

22.4 RISC-V Standard CSRs Supported in the Nuclei processor core

These CSRs are following RISC-V standard privileged architecture specification. This document will not repeat its content here, please refer to RISC-V standard privileged architecture specification for more details.

This chapter only introduces the RISC-V Standard CSRs supported in the Nuclei processor core, but those CSRs who also have some unique differences implemented in Nuclei processor core.

22.4.1 mie

In Nuclei processor core, the format and features of CSR mie is same as RISC-V standard, please refer to RISC-V standard privileged architecture specification for more details of CSR mie.

But note: the mie CSR is not effective when the core is in the CLIC mode, and the readout of mie is always 0, the writing is ignored.

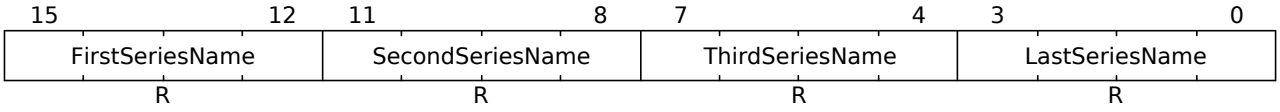
22.4.2 mip

In Nuclei processor core, the format and features of CSR mip is same as RISC-V standard, please refer to RISC-V standard privileged architecture specification for more details of CSR mip.

But note: the mip CSR is not effective when the core is in the CLIC mode, and the readout of mip is always 0, the writing is ignored.

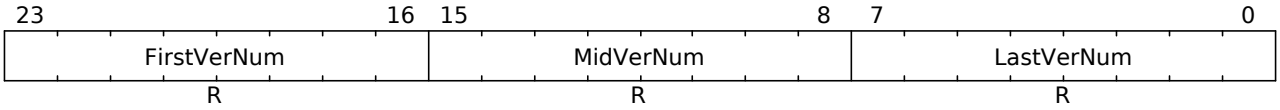
22.4.3 marchid

marchid is to indicate the core series . marchid[MXLEN-1:16] is reserved for future use. marchid[15:0] indicates the core series name, for example, N600 is 0x0600, NX607 is 0xb607, UX608 is 0xc608. The format is as followed:



22.4.4 mimpid

mimpid is to indicate the core product version (also the RTL version), mimpid[MXLEN-1:24] is reserved for future use , mimpid[23:0] indicates the core product version X.Y.Z, X/Y/Z are all hexadecimal number and each occupies 8 bits. The format is as followed:



22.4.5 mhartid

In Nuclei processor core, the format and features of CSR mhartid is same as RISC-V standard, please refer to RISC-V standard privileged architecture specification for more details of CSR mhartid.

In the Nuclei processor core, hart ID is controlled by input signal core_mhartid.

Note:

- According to RISC-V architecture, one hart is required to have a known hart ID of 0, other harts ID can be in 1~1023.
- In Nuclei Subsystem Design with multi cluster of Nuclei core, it uses the bits 0 ~ 7 to reflect the hart ID/Num and bits 8 ~ 15 to reflect the cluster ID/Num in this CSR.

22.4.6 mtvec

The mtvec register holds the entry address of the interrupt and exception handler. Field of mtvec register is shown in following table.

- When mtvec holds the exception entry address:
  - The value of the address field must always be aligned on a 64-byte boundary.
- When mtvec holds the interrupt entry address:
  - When mtvec.MODE != 6'b000011, the processor uses the CLINT interrupt mode.
  - When mtvec.MODE = 6'b000011, the processor uses the CLIC interrupt mode.
    - See *Non-Vectored Processing Mode* (page 34) for more information about non-vectored mode interrupt handler entry address.
    - See *Vectored Processing Mode* (page 37) for more information about vectored mode interrupt handler entry address.

Table 22.2: mtvec register

Field	Bits	Description
ADDR	MXLEN-1:6	mtvec address
MODE	5:0	**MODE field determine interrupt mode:** 000011 means CLIC interrupt mode; others means CLINT interrupt mode.

22.4.7 mcause

The mcause is written with a code indicating the reason that caused the trap. The format and features of CSR mcause is basically same as RISC-V standard, please refer to RISC-V standard privileged architecture specification for more details. But in CLIC mode for Nuclei processor core, there some additional fields added to support CLIC mode interrupt handling.

The mcause register is formatted as shown in following table:

Table 22.3: mcause register

Field	Bits	Description
INTERRUPT	MXLEN-1	Current trap type: 0: Exception or NMI; 1: Interrupt.
Reserved	MXLEN-2:31	Reserved 0
MINHV	30	Indicate processer is reading interrupt vector table
MPP	29:28	Privilege mode before interrupt
MPIE	27	Interrupt enable before interrupt
Reserved	26:24	Reserved 0
MPIL	23:16	Previous Interrupt Level
Reserved	15:12	Reserved 0
EXCCODE	11:0	Exception/Interrupt Encoding

Note:

- Filed of MINHV, MPP, MPIE and MPIL are only effect in CLIC mode. When in CLINT mode, these field is masked read as zero, write ignored.
- In CLIC mode, the mstatus.MPIE and mstatus.MPP are the mirror images of mcause.MPIE and mcause.MPP.
- For overflow of Stack Check, the mcause.EXCCODE is 0x18; for underflow of Stack Check, the mcause.EXCCODE is 0x19.
- The mcause.EXCCODE of NMI can be 0x1 or 0xfff, the value is controlled by mmisc_ctl, see more detail in

mmisc_ctl (page 120).

22.4.8 mcycle and mcycleh

The RISC-V architecture defines a 64-bits width cycle counter which indicates how many cycles the processor has executed. Whenever the processor is working, the counter will increase automatically.

The mcycle register records the lower 32-bits of counter and mcycleh records the higher 32-bits. The format and features of CSR mcycle/mcycleh are basically same as RISC-V standard, please refer to RISC-V standard privileged architecture specification for more details.

But in Nuclei processor core, considering the counter increases the power consumption, there is an extra bit in the customized CSR mcountinhibit that can pause the counter to save power when users don't need to monitor the performance through the counter. See Section *mcountinhibit* (page 113) for more information.

22.4.9 minstret and minstreth

The RISC-V architecture defines a 64-bits width counter which records how many instructions have been executed successfully.

The minstret register records the low 32-bits of counter and minstreth records the high 32-bits. The format and features of CSR minstret/minstreth are basically same as RISC-V standard, please refer to RISC-V standard privileged architecture specification for more details.

But in Nuclei processor core, considering the counter has power consumption, there is an extra bit in the customized CSR mcountinhibit that can turn off the counter to save power when users don't need to learn the performance through the counter. See Section *mcountinhibit* (page 113) for more information.

22.5 Customized CSRs supported in Nuclei processor core

This section introduces customized CSRs in the Nuclei processor core.

22.5.1 mcountinhibit

The mcountinhibit register controls the counting of mcycle/mcycleh and minstret/minstreth registers to save power when users don't need them.

Table 22.4: mcountinhibit register

Field	Bits	Description
Reserved	MXLEN-1:3	Reserved 0
IR	2	When IR is 1, minstret/minstreth is stop counting.
Reserved	1	Reserved 0
CY	0	When CY is 1, mcycle/mcycleh is is stop counting.

22.5.2 milm_ctl

The milm_ctl register controls the ILM (Instruction Local Memory) address space to enable or disable it based on user¡¯s application scenarios.

Table 22.5: milm_ctl register

Field	Bits	Description
ILM_BPA	MXLEN-1:10	Read-Only. The base physical address of ILM. For example, the instruction local memory of size 4KB starts at address 12KB (0x3000), then the ILM_BPA is 0xC (take the upper 22 bits of 0x3000).
Reserved	9:5	Reserved 0
ILM_ECC_CHK_EN	4	Controls to check the ECC Code or not when core access to ILM. This bit can be turned on for injecting ECC errors to test the ECC handler. 0: Disable to check the ECC codes (Default) 1: Enable to check the ECC codes Note: When ILM or ECC is not configured, this field is tied to 0.
ILM_ECC_INJ_EN	3	Controls to inject the ECC Code in CSR mecc_code to ILM. 0: Disable to inject the ECC codes (Default) 1: Enable to inject the ECC codes Note: When ILM or ECC is not configured, this field is tied to 0
ILM_ECC_EXCP_EN	2	ILM double bit ECC exception enable control: 0: ECC error will not trigger exception (Default) 1: ECC error will trigger exception Note: When ILM or ECC is not configured, this field is tied to 0
ILM_ECC_EN	1	ILM ECC feature enable control. 0: Disable ECC 1: Enable ECC (default) Note: When ILM or ECC is not configured, this field is tied to 0.
ILM_ENABLE	0	Instruction Local Memory enable bit: 0: ILM is disabled 1: ILM is enabled (default) Note: This bit only effects core access ILM, dose not effect master of Slave Port.

22.5.3 mdlm_ctl

The mdlm_ctl register controls the DLM (Data Local Memory) address space to enable or disable it based on user's application scenarios.

Table 22.6: mdlm_ctl register

Field	Bits	Description
DLM_BPA	MXLEN-1:10	Read-Only. The base physical address of DLM. It has to be aligned to multiple of DLM size. For example, data local memory of size 4KB starts at address 12KB (0x3000), then the DLM_BPA is 0xC (take the upper 22 bits of 0x3000).
Reserved	9:5	Reserved 0
DLM_ECC_CHK_EN	4	Controls to check the ECC Code or not when core access to DLM. This bit can be turned on for injecting ECC errors to test the ECC handler. 0: Disable to check the ECC codes (Default) 1: Enable to check the ECC codes Note: When DLM or ECC is not configured, this field is tied to 0.

continues on next page

Table 22.6 – continued from previous page

Field	Bits	Description
DLM_ECC_INJ_EN	3	Controls to inject the ECC Code in CSR mecc_code to DLM. 0: Disable to inject the ECC codes (Default) 1: Enable to inject the ECC codes Note: When DLM or ECC is not configured, this field is tied to 0
DLM_ECC_EXCP_EN	2	DLM double bit ECC exception enable control. 0: Disable double bit ECC exception (default) 1: Enable double bit ECC exception Note: When DLM or ECC is not configured, this field is tied to 0
DLM_ECC_EN	1	DLM ECC enable feature control. 0: Disable ECC ; 1: Enable ECC (default). Note: When DLM or ECC is not configured, this field is tied to 0.
DLM_ENABLE	0	Data Local Memory enable bit. 0: DLM is disabled; 1: DLM is enabled (default). Note: This bit only effects core access ILM, dose not effect master of Slave Port.

22.5.4 mnvec

The Nuclei processor core customized CSR mnvec register holds the NMI entry address.

In order to understand this register, please see Chapter *NMI Handling in Nuclei processor core* (page 19) for more information about NMI.

During a processor running a program, the program will be forced to jump into a new PC address when an NMI is triggered. The PC address is determined by mnvec.The value of mnvec is controlled by mmisc_ctl, see more information in Section *mmisc_ctl* (page 120).

22.5.5 msubm

The Nuclei processor core customized CSR msubm register holds the current machine sub-mode and the machine sub-mode before the current trap. See Section *Machine Sub-Mode added by Nuclei* (page 15) for details.

Table 22.7: msubm register

Field	Bits	Description
Reserved	MXLEN-1:10	Reserved 0
PTYP	9:8	Machine sub-mode before entering the trap: 0: Normal Machine Mode; 1: Interrupt Handling Mode; 2: Exception Handing Mode; 3: NMI Handing Mode.
TYP	7:6	Current sub-mode: 0: Normal Machine Mode; 1: Interrupt Handling Mode; 2: Exception Handing Mode; 3: NMI Handing Mode.
Reserved	5:0	Reserved 0.

22.5.6 mstack_ctrl

The Nuclei processor core customized CSR mstack_ctrl register is to control the Stack Check realted function.

Table 22.8: mstack_ctrl register

Field	Bits	Description
Reserved	MXLEN-1:3	Reserved 0
MODE	2	Mode of stack checking: 0: Overflow and Underflow check (Default) ; 1: Track the stack top.
UDF_EN	1	Stack underflow check enable: 0: Disable (Default); 1: Enable; When MODE is 0, and UDF_EN is 1, the hardware will check stack underflow.
OVF_TRACK_EN	0	Stack overflow check or track enable: 0: Disable (Default); 1: Enable; When MODE is 0, and OVF_TRACK_EN is 1, the hardware will check stack overflow; when MODE is 1, and OVF_TRACK_EN is 1, the hardware will track the stack top.

Note:

- Currently only 300 Series v4.2.0 or later support this Stack Check function.
- The Stack Check can work as excpect only when the Stack downward growth.

22.5.7 mstack_bound

The Nuclei processor core customized CSR mstack_bound register is check stack overflow or track the stack top.

Table 22.9: mstack_bound register

Field	Bits	Description
BOUND	MXLEN-1:0	Stack top's value for stack overflow check or track (Default value is all ones): When MODE is 0, and OVF_TRACK_EN is 1, if the sp is smaller than BOUND, the core will report Stack Overflow exception and the mcause.EXCCODE is 0x18. When MODE is 1, and OVF_TRACK_EN is 1, if the sp is smaller than BOUND, the BOUND's value will be updated to sp.

Note:

- User should set the mstack_bound before mstack_ctrl to check stack overflow.

22.5.8 mstack_base

The Nuclei processor core customized CSR mstack_base register is check stack underflow.

Table 22.10: mstack_base register

Field	Bits	Description
BASE	MXLEN-1:0	Stack base's value for stack underflow check (Default value is all ones): When MODE is 0, and UDF_TRACK_EN is 1, if the sp is bigger than BASE, the core will report Stack Underflow exception and the mcause.EXCCODE is 0x19.

Note:

- User should set the mstack_base before mstack_ctrl to check stack underflow.

22.5.9 mdcause

Since there might be some exceptions share the same mcause.EXCCODE value. To further record the differences, Nuclei processor core customizes CSR mdcause register to record the detailed information about the exception.

Table 22.11: mdcause register

Field	Bits	Description
Reserved	MXLEN-1:3	Reserved 0

continues on next page

Table 22.11 – continued from previous page

Field	Bits	Description
MDCAUSE	2:0	Further record the detailed information about the exception. When mcause.EXCCODE = 1 (Instruction access fault): 0: Reserved 1: PMP permission violation 2: Bus error 3-6: Reserved 7: ECC error When mcause.EXCCODE = 5 (Load access fault): 0: Reserved 1: PMP permission violation bus error 2: Bus error caused by core memory read 3: Bus error caused by NICE write back. 4-6: Reserved 7: ECC error Note: although this error ideally is nothing to do with the Load access fault, but they just shared the same mcause.EXCCODE to simplify the hardware implementation. When mcause.EXCCODE = 7 (Store/AMO access fault): 0: Reserved 1: PMP permission violation bus error 2: Bus error caused by core memory write 3-6: Reserved 7: ECC error When mcause.EXCCODE = 12 (Instruction page fault): 0-4: Reserved 5: Instruction page fault caused by MMU 6: Instruction page fault caused by SPMP 7: ECC error When mcause.EXCCODE = 13 (Load page fault): 0-4: Reserved 5: Load page fault caused by MMU 6: Load page fault bus error caused by SPMP 7: ECC error When mcause.EXCCODE = 15 (Store page fault): 0-4: Reserved 5: Store page fault caused by MMU 6: Store page fault bus error caused by SPMP 7: ECC error

22.5.10 mcache_ctl

Nuclei processor core customizesd CSR mcache_ctl register to control the I-Cache and D-Cache features.

Table 22.12: mcache_ctl register

Field	Bits	Description
Reserved	MXLEN-1:23	Reserved 0.
DC_PREFETCH_EN	22	D-Cache CMO prefetch enable control: 0: Disable (default) 1: Enable
DC_ECC_CHK_EN	21	Controls to check the ECC Code when core access to D-Cache. 0: Disable to check the ECC codes (Default) 1: Enable to check the ECC codes Note: When D-Cache or ECC is not configured, this field is tied to 0

continues on next page

Table 22.12 – continued from previous page

Field	Bits	Description
DC_DRAM_ECC_INJ_EN	20	Controls to inject the ECC Code in CSR mecc_code or not to D-Cache data rams. This bit can be turned on for injecting ECC errors to test the ECC handler. 0: Disable to inject the ECC codes (Default) 1: Enable to inject the ECC codes Note: When D-Cache or ECC is not configured, this field is tied to 0.
DC_TRAM_ECC_INJ_EN	19	Controls to inject the ECC Code in CSR mecc_code or not to D-Cache tag rams. This bit can be turned on for injecting ECC errors to test the ECC handler. 0: Disable to inject the ECC codes (Default) 1: Enable to inject the ECC codes Note: When D-Cache or ECC is not configured, this field is tied to 0.
DC_ECC_EXCP_EN	18	D-Cache double bit ECC exception enable control: 0: ECC error will not trigger exception (Default) 1: ECC error will trigger exception Note: When D-Cache or ECC is not configured, this field is tied to 0.
DC_ECC_EN	17	D-Cache ECC enable control: 0: Disable ECC 1: Enable ECC (Default) Note: When D-Cache or ECC is not configured, this field is tied to 0.
DC_EN	16	D-Cache enable control: 0: D-Cache Disable (default) 1: D-Cache Enable
Reserved	15:10	Reserved 0.
IC_PREFETCH_EN	9	I-Cache CMO prefetch enable control: 0: Disable (default) 1: Enable
IC_ECC_CHK_EN	8	Controls to check the ECC when core access to I-Cache. 0: Disable to check the ECC codes (Default) 1: Enable to check the ECC codes Note: When I-Cache or ECC is not configured, this field is tied to 0
IC_CANCEL_EN	7	Supported only in 900 series, I-Cache change flow canceling enable control: 0: Disable canceling current accesses in icache e1 stage when change flow happens 1: Enable canceling previous accesses in icache e1 stage when change flow happens Note: When I-Cache is not configured, this field is tied to 0.
IC_PF_EN	6	Supported only in 900 series, I-Cache prefetch enable control: 0: Disable prefetching 1: Enable prefetching, when I-Cache miss, prefetch next-line into I-Cache when not cross 4K Byte. Note: When I-Cache is not configured, this field is tied to 0.
IC_DRAM_ECC_INJ_EN	5	Controls to inject the ECC Code in CSR mecc_code to I-Cache data rams. This bit can be turned on for injecting ECC errors to test the ECC handler. 0: Disable to inject the ECC codes (Default) 1: Enable to inject the ECC codes Note: When I-Cache or ECC is not configured, this field is tied to 0.

continues on next page

Table 22.12 – continued from previous page

Field	Bits	Description
IC_TRAM_ECC_INJ_EN	4	Controls to inject the ECC Code in CSR mecc_code to I-Cache tag rams. This bit can be turned on for injecting ECC errors to test the ECC handler. 0: Disable to inject the ECC codes (Default) 1: Enable to inject the ECC codes Note: When I-Cache or ECC is not configured, this field is tied to 0.
IC_ECC_EXCP_EN	3	I-Cache double bit ECC exception enable control: 0: ECC error will not trigger exception (Default) 1: ECC error will trigger exception Note: When I-Cache or ECC is not configured, this field is tied to 0.
IC_ECC_EN	2	I-Cache ECC enable control: 0: Disable ECC 1: Enable ECC (Default) Note: When I-Cache or ECC is not configured, this field is tied to 0.
IC_SCPD_MOD	1	I-Cache Scratchpad Mode enable control: 0: I-Cache works in the normal mode (default) 1: I-Cache works in the Scratchpad mode. When under this mode, the Data SRAM of I-Cache will be reused and downgraded to memory mapped SRAM which can be accessed by instruction fetch and load/store access, like ILM and DLM, but called as Scratchpad here. Note: this mode only take effect when IC_EN bit is disabled by software. (default)
IC_EN	0	I-Cache enable control: 0: I-Cache Disable (default) 1: I-Cache Enable

Note: only when the I-Cache is disabled (IC_EN bit as 0) and scratchpad mode is enabled, i.e., mcache_ctl[1:0] is 2'b10 (default value after reset), I-Cache really works as the Scratchpad.

22.5.11 mmisc_ctl

The Nuclei processor core customized CSR mmisc_ctl controls many Nuclei micro-architecture implementation related features.

Table 22.13: mmisc_ctl register

Field	Bits	Description
Reserved	MXLEN-1:17	Reserved 0.
CSR_EXCL_ENABLE	17	Exclusive instruction(lr,sc) on Non-cacheable/Device memory can send exclusive flag in memory bus. 0: Disable exclusive flag goes to Memory Interface (Reset Value). 1: Enable exclusive flag goes to Memory Interface.
Reserved	16:13	Reserved 0.
LDSPEC_ENABLE	12	Enable or disable the Load Speculative goes to Mem Interface: 0: Disable Load Speculative goes to Memory Interface (Reset Value). 1: Enable Load Speculative goes to Memory Interface.
SIJUMP_ENABLE	11	Control the SIJUMP mode of trace: 0: SIJUMP mode is off(Reset Value); 1: SIJUMP mode is on.

continues on next page

Table 22.13 – continued from previous page

Field	Bits	Description
IMRETURN_ENABLE	10	Control the IMRETURN mode of trace: 0: IMRETURN mode is off(Reset Value). 1: IMRETURN mode is on.
NMI_CAUSE_FFF	9	Control mnvec and mcause.EXCCODE of NMI: 0: The value of mnvec equals the PC address after reset, mcause.EXCCODE of NMI is 0x1; 1: The value of mnvec is the same as the value of mtvec , mcause.EXCCODE of NMI is 0xfff.
CORE_BUS_ERR	8	Control the bus error caused by core is exception or interrupt: 0: Core Bus Error caused by core is an exception; When Core Bus Error is an excetpion, please check *Core Bus Error Exception* (page 121) for more details. 1: Core Bus Error caused by core is a interrupt. When Core Bus Error is a interrupt, please check *Core Bus Error Interrupt* (page 122) for more details.
ZCMT_ZCMP_EN	7	Control the Zc Ext uses the cfdsp of D Ext's encoding or not. When there is Zc and no D Ext, this bit is always 1 and read only. When there is Zc and D, the reset value is 0; and if FPU is off by software, then software can write 1 to enable Zc.
UNALGN_ENA_BLE	6	Enable or disable misaligned load/store access, if disabled, accessing misaligned memory locations will trigger an Address Misaligned exception: 0: Disable misaligned load/store access; 1: Enable misaligned load/store access. Note: This field only takes effects for load/store specified in I/F/D Extension, for load/store specified in A Extension, misaligned accesses always trigger an Address Misaligned exception.
Reserved	5:4	Reserved 0.
BPU_ENABLE	3	Enable or disable the BPU Unit: 0: Disable the BPU Unit; 1: Enable the BPU Unit. Note: BPU is on by default after reset. If BPU is disable by software, then all branches are predicted as jump statically until the BPU is enable again. So software should set this bit to 1 to enable BPU.
Reserved	2:0	Reserved 0.

22.5.11.1 Core Bus Error Exception

Currently Nuclei processor core implments 7 types Core Bus Error, when it is configured to be exception (by CSR *mmisc_ctl*), the mapping of Core Bus Error type and CSR coding of *mcause* and *mdcause* is defined in below table.

Table 22.14: Core Bus Err Type and Exception Coding Mapping

Core Bus Err Type	*mcause* Exception Coding	*mdcause*
Bus Err caused by core memory read	5 (load access fault)	2
Bus Err caused by core memory write	7 (store access fault)	2
Bus Err caused by core memory read PMP violation	5 (load access fault)	1
Bus Err caused by core memory write PMP violation	7 (store access fault)	1
Bus Err caused by core memory read ECC error	5 (load access fault)	7
Bus Err caused by core memory write ECC error	7 (store access fault)	7

continues on next page

Table 22.14 – continued from previous page

Core Bus Err Type	*mcause* Exception Coding	*mdcause*
Bus Err caused by core memory read SPMP violation	13 (load page fault)	6
Bus Err caused by core memory write SPMP violation	15 (store page fault)	6
Bus Err caused by core's NICE write back	5 (load access fault)	3

22.5.11.2 Core Bus Error Interrupt

Currently Nuclei processor core implments 7 types Core Bus Error, when it is configured to be interrupt (by CSR *mmisc_ctl*), the mapping of Core Bus Error type and CSR coding of *mcause* and *mdcause* is defined in below table. And it belongs to Internal Interrupt, and Interrupt ID is fixed to 18.

Table 22.15: Core Bus Err Type and Interrupt Coding Mapping

Core Bus Err Type	*mcause* Interrupt Coding	*mdcause*
Bus Err casued by core memory read	18	2
Bus Err casued by core memory write	18	2
Bus Err casued by core memory read PMP violation	18	1
Bus Err casued by core memory write PMP violation	18	1
Bus Err casued by core memory read ECC error	18	7
Bus Err casued by core memory write ECC error	18	7
Bus Err casued by core memory read SPMP violation	18	6
Bus Err casued by core memory write SPMP violation	18	6
Bus Err casued by core's NICE write back	18	3

Note:

- As Core Bus Error Interrupt belongs to Internal Interrupt, both CLIC Mode or CLINT Mode (with PLIC), the Interrupt ID is the same (18).
- In CLIC Mode, as Nuclei ECLIC can support 3 types of interrupt trigger and Core Bus Error Interrupt is edge type, so besides to set clicintie[18] = 1'b1 to enable it , user should also set clicintattr[18].trig = 2'b01.
- In CLINT Mode (with PLIC), the Core Bus Error is controlled by CSR *mie* and *mip*. And user has no need to consider the interrupt trigger type.

22.5.12 mtvt

This reigister is from the CLIC draft of RISC-V fast interrupt task group.

The mtvt register holds the base address of interrupt vector table (in CLIC mode), and the base address is aligned at least 64-byte/128-byte boundary. See Section *(CLIC mode) Interrupt Vector Table* (page 31) for more details.

In order to improve the performance and reduce the gate count, the alignment of the base address in mtvt is determined by the actual number of interrupts, which is shown in the following table.

Table 22.16: mtvt alignment

Max Interrupt Number	mtvt alignment in RV32	mtvt alignment in RV64
0 to 16	64-byte	128-byte
17 to 32	128-byte	256-byte
33 to 64	256-byte	512-byte
65 to 128	512-byte	1KB
129 to 256	1KB	2KB
257 to 512	2KB	4KB
513 to 1024	4KB	8KB
1025 to 2048	8KB	16KB
2049 to 4096	16KB	32KB

22.5.13 mintstatus

This reigister is from the CLIC draft of RISC-V fast interrupt task group.

The mintstatus register holds the active interrupt level for all the privilege mode.

Table 22.17: mintstatus register

Field	Bits	Description
Reserved	MXLEN-1:32	Reserved 0
MIL	31:24	The active interrupt level in machine mode.
Reserved	23:8	Reserved 0.
UIL	7:0	The active interrupt level in user mode.

22.5.14 mtvt2

mtvt2 is used to indicate the entry address of the common base handler shared by all ECLIC non-vectored interrupts. See more information about mtvt2 in Section *Non-Vectored Processing Mode* (page 34).

Table 22.18: mtvt2 register

Field	Bits	Description
CMMON-CODE-ENTRY	MXLEN-1:2	When mtvt2.MTVT2EN=1, this field determines the entry address of interrupt common code in ECLIC non-vector mode.
Reserved	1	Reserved 0.
MTVT2EN	0	mtvt2 enable: 0: the entry address of interrupt common code in ECLIC non-vector mode is determined by mtvec; 1: the entry address of interrupt common code in ECLIC non-vector mode is determined by mtvt2.COMMON-CODE-ENTRY.

22.5.15 jalmnxti

The Nuclei processor core customized CSR jalmnxti to reduce the delay for interrupt and accelerates interrupt tail-chaining.

The jalmnxti included all functionality of mnxti, besides it also include enabling the interrupt, handling the next interrupt, jumping to the next interrupt entry address, and jumping to the interrupt handler. So, the jalmnxti can decrease the instruction numbers to speed up the interrupt handling and tail-chaining.

See more information related tail-chaining in Section *Interrupt Handling in Nuclei processor core* (page 23).

22.5.16 pushmsubm

The Nuclei processor core customized CSR pushmsubm provides a method to store the value of msubm in memory space which base address is SP with CSR instruction csrrwi.

For example:

```
csrrwi x0, PUSHMSUBM, 1
```

This instruction stores the value of msubm in SP+1*4 address.

22.5.17 pushmcause

The Nuclei processor core customized CSR pushmcause provides a method to store the value of mcause in memory space which base address is SP with CSR instruction csrrwi.

For example:

```
csrrwi x0, PUSHMCAUSE, 1
```

This instruction stores the value of mcause in SP+1*4 address.

22.5.18 pushmepc

The Nuclei processor core customized CSR pushmepc provides a method to store the value of mepc in memory space which base address is SP with CSR instruction csrrwi.

For example:

```
csrrwi x0, PUSHMEPC, 1
```

This instruction stores the value of mepc in SP+1*4 address.

22.5.19 mscratchcsw

This reigister is from the CLIC draft of RISC-V fast interrupt task group.

The mscratchcsw register is useful to swap the value between the target register and mscratch when privilege mode change.

Using a CSR read instruction to perform mscratchcsw, when the privilege mode is changed after taking an interrupt, following pseudo instruction operations are performed:

```
csrrw rd, mscratchcsw, rs1

// Pseudocode operation.
if (mcause.mpp!= M-mode) then {
   t = rs1; rd = mscratch; mscratch = t;
} else {
   rd = rs1; // mscratch unchanged.
}

// Usual use: csrrw sp, mscratchcsw, sp
```

When the processor takes an interrupt in a lower privilege mode, the processor enters a higher privilege mode to handle the interrupt and need to store the status of processor into the stack before taking the interrupt. If the processor continues to use SP in low privilege mode, data in the higher privilege mode will be saved in the memory space which is accessible in the lower privilege mode.

RISC-V defines that when the processor is in a lower privilege mode, then data in SP of the higher privilege mode can be stored in mscratch. And in this way, the value of SP can be recovered from mscratch when the processor goes back to the higher privilege mode.

It will cost a lot of cycles to execute the program above using standard instructions, so RISC-V defines mscratchcsw register. After entering an interrupt, the processor executes one mscratchcsw CSR instruction to swap the value between mscratch and SP to recover the value of SP of the higher privilege mode. At the same time, copy the value of SP of the lower privilege to mscratch. Before the mret instruction to exit interrupt, add a mscratchcsw instruction to swap value between mscratch and SP. It will recover the SP value of the lower privilege mode and store the higher privilege mode SP to mscratch again. In this way, only two instructions are needed to solve the stack pointer (SP) switching problem of different privileged modes, which speeds up interrupt processing.

22.5.20 mscratchcswl

This reigister is from the CLIC draft of RISC-V fast interrupt task group.

The mscratchcswl register is used to exchange the destination register with the value of mscratch to speed up interrupt processing when switching between interrupt mode (not including exception) and normal mode.

Using the CSR instruction to read the register mscratchcswl, with unchanged privilege mode, the following register operations are performed when there is a switch between the interrupt handler and the application program:

```
csrrw rd, mscratchcswl, rs1

// Pseudocode operation.
if ( (0 == mcause.mpil) != (0 == mintstatus.mil) ) then {
   t = rs1; rd = mscratch; mscratch = t;
} else {
   rd = rs1; // mscratch unchanged.
}

// Usual use: csrrw sp, mscratchcswl, sp
```

In the same privilege mode, separating the interrupt handler task from the task space of the application task can increase robustness, reduce space usage, and facilitate system debugging. The interrupt handler has a non-zero interrupt level while the application task has a zero interrupt level. According to this feature, the RISC-V architecture defines the mscratchcswl register. Similar to mscratchcsw, adding a CSR instruction of mscratchcswl to the beginning and the end of the interrupt service routine enables a fast stack pointer switch between the interrupt handler and the regular application, ensuring the separation of the stack space between the interrupt handler and the regular application.

22.5.21 sleepvalue

The Nuclei processor core customized CSR sleepvalue controls different sleep modes. See Section *Enter the Sleep Mode* (page 103) for more information.

Table 22.19: sleepvalue register

Field	Bits	Description
Reserved	MXLEN-1:1	Reserved 0.
SLEEPVALUE	0	Control WFI sleep mode: 0: shallow sleep mode (After WFI, it recommends SoC to turn core_clk off); 1: deep sleep mode (After WFI, it recommends SoC to turn core_clk and core_aon_clk both off). Reset default value is 0.

22.5.22 txevt

The Nuclei processor core customized CSR txevt controls output events.

Table 22.20: txevt register

Field	Bits	Description
Reserved	MXLEN-1:1	Reserved 0.
TXEVT	0	Event control: 0: No action; 1: The core will trigger a single-cycle pulse output signal tx_evt as event signal. This bit will be automatically cleared to 0 in the next cycle. Reset default value is 0.

22.5.23 wfe

The Nuclei processor core customized CSR wfe controls whether the processor should be awakened by interrupt or event. See Section *Wait for Event* (page 104) for more information.

Table 22.21: wfe register

Field	Bits	Description
Reserved	MXLEN-1:1	Reserved 0.
WFE	0	Control whether the processor should be awakened by interrupt or event. 0: The processor should be awakened by interrupt and NMI in sleep mode; 1: The processor should be awakened by event and NMI in sleep mode. Reset default value is 0.

22.5.24 ucode

This register is from the P-extension draft of RISC-V P-extension task group.

This register only exists when the core has been configured to support the P extension. The CSR register ucode is used to record if there is overflow happened in DSP instructions.

Table 22.22: ucode register

Field	Bits	Description
Reserved	MXLEN-1:1	Reserved 0.
OV	0	Record if there is overflow happened in DSP instructions. If there is overflow then this field OV is set as 1, software can write 0 to this register to clear it. Reset default value is 0.

22.5.25 mcfg_info

This CSR is used to show the processor core's configuration information.

Table 22.23: mcfg_info register

Field	Bits	Description
Reserved	MXLEN-1:25	Reserved 0.
XLCZ	24	Indicate Core supports XLCZ or not, only active when ZC_XLCZ_EXT is 1. 0: XLCZ Support; 1: No XLCZ support.
VNICE	23	Indicate Core supports VNICE or not: 0: No VNICE support; 1: VNICE support.
SAFETY_MECHA	22:21	Indicate Core's safety mechanism: 00: No Safety Mechanism; 01: Lockstep; 10: Lockstep + Split Mode; 11: ASIL-B.
ETRACE	20	Indicate Core supports Etrace or not: 0: No ETrace support; 1: ETrace support.

continues on next page

Table 22.23 – continued from previous page

Field	Bits	Description
SEC_MODE	19	Indicate Core supports Smwg Extension or not: 0: No Smwg Extension support; 1: Smwg Extension support.
VPU_DEGREE	18:17	Indicate the VPU degree of parallel: 00: DLEN = 1/2 VLEN; 01: DLEN = VLEN; others are reserved.
IREGION_EXIST	16	Shows the IREGION exist or not. It is Read Only. 0: IREGION not exist; 1: IREGION exists. Note: IREGION (Internal Region) means the address space of all Private Peripherals in core is continuous and base address is defined by IREGION. If no IREGION, user can seperately configures the address space of each Private Peripheral. Please check Private Peripherals chapter of each series databook for more details. New version of Nuclei Core supports IREGION and it is fixed in these new versions. Please check Configuration Options chapter of related databook for more details.
ZC_XLCZ_EXT	15	Indicate Core supports Zc and Xlcz Extension or not: 0: No Zc and Xlcz Extensions support; 1: Zc or Zc+Xlcz Extensions support.
DSP_N3	14	Global configuration for DSP N3 Extension Support: 0: No DSP N3 Extension support; 1: Has DSP N3 Extension support. Note: Only take effect when DSP is configured.
DSP_N2	13	Global configuration for DSP N2 Extension Support: 0: No DSP N2 Extension support; 1: Has DSP N2 Extension support. Note: Only take effect when DSP is configured.
DSP_N1	12	Global configuration for DSP N1 Extension Support: 0: No DSP N1 Extension support; 1: Has DSP N1 Extension support. Note: Only take effect when DSP is configured.
SMP	11	Global configuration for SMP support: 0: No SMP support; 1: Has SMP support.
DCACHE	10	Global configuration for D-Cache support: 0: No D-Cache support; 1: Has D-Cache support.
ICACHE	9	Global configuration for I-Cache support: 0: No I-Cache support; 1: Has I-Cache support.
DLM	8	Global configuration for IDLM support: 0: No DLM support; 1: Has DLM support.

continues on next page

Table 22.23 – continued from previous page

Field	Bits	Description
ILM	7	Global configuration for ILMsupport: 0: No ILM support; 1: Has ILM support.
NICE	6	Global configuration for NICE support: 0: No NICE support; 1: Has NICE support.
PPI	5	Global configuration for PPI support: 0: No PPI support; 1: Has PPI support.
FIO	4	Global configuration for FIO support: 0: No FIO support; 1: Has FIO support.
PLIC	3	Global configuration for PLIC support: 0: No PLIC support; 1: Has PLIC support.
CLIC	2	Global configuration for CLIC support: 0: No CLIC support; 1: Has CLIC support.
ECC	1	Global configuration for ECC support: 0: No ECC support; 1: Has ECC support.
TEE	0	Global configuration for TEE support: 0: No TEE support; 1: Has TEE support.

22.5.26 micfg_info

This CSR is used to show the ILM and I-Cache configuration information.

Table 22.24: micfg_info register

Field	Bits	Description
Reserved	MXLEN-1:23	Reserved 0.
ILM_ECC	22	ECC support for the instruction local memory (ILM): 0: No ECC support 1: Has ECC support
ILM_XONLY	21	Indicates if ILM is execute-only. If ILM is execute-only, load/store instructions cannot access the ILM region: 0: ILM is not execute-only 1: ILM is execute-only

continues on next page

Table 22.24 – continued from previous page

Field	Bits	Description
ILM_SIZE	20:16	Indicates the size of ILM and the size should be power of 2: 0: 0Byte 1: 256 Bytes 2: 512 Bytes 3: 1KB 4: 2KB 5: 4KB 6: 8KB 7: 16KB 8: 32KB 9: 64KB 10: 128KB 11: 256KB 12: 512KB 13: 1MB 14: 2MB 15: 4MB 16: 8MB 17: 16MB 18: 32MB 19: 64MB 20: 128MB 21: 256MB 22: 512MB others: Reserved
Reserved	15:11	Reserved 0.
IC_ECC	10	ECC support for the I-Cache 0: No ECC support 1: Has ECC support
IC_LSIZE	9:7	I-Cache line size: 0: No I-Cache 1: 8 Bytes 2: 16 Bytes 3: 32 Bytes 4: 64 Bytes 5: 128 Bytes others: Reserved
IC_WAY	6:4	I-Cache ways: 0: Direct-mapped 1: 2 way 2: 3 way 3: 4 way 4: 5 way 5: 6 way 6: 7 way 7: 8 way

continues on next page

Table 22.24 – continued from previous page

Field	Bits	Description
IC_SET	3:0	I-Cache sets per way: 0: 8 1: 16 2: 32 3: 64 4: 128 5: 256 6: 512 7: 1024 8: 2048 9: 4096 10: 8192 others: Reserved

22.5.27 mdcfg_info

This CSR is used to show the DLM and D-Cache configuration information.

Table 22.25: mdcfg_info register

Field	Bits	Description
Reserved	MXLEN-1:22	Reserved 0.
DLM_ECC	21	ECC support for the data local memory (DLM): 0: No ECC support 1: Has ECC support
DLM_SIZE	20:16	Indicates the size of DLM and the size should be power of 2: 0: 0Byte 1: 256 Bytes 2: 512 Bytes 3: 1KB 4: 2KB 5: 4KB 6: 8KB 7: 16KB 8: 32KB 9: 64KB 10: 128KB 11: 256KB 12: 512KB 13: 1MB 14: 2MB 15: 4MB 16: 8MB 17: 16MB 18: 32MB 19: 64MB 20: 128MB 21: 256MB 22: 512MB others: Reserved
Reserved	15:11	Reserved 0.

continues on next page

Table 22.25 – continued from previous page

Field	Bits	Description
DC_ECC	10	ECC support for the D-Cache 0: No ECC support 1: Has ECC support
DC_LSIZE	9:7	D-Cache line size: 0: No D-Cache 1: 8 Bytes 2: 16 Bytes 3: 32 Bytes 4: 64 Bytes 5: 128 Bytes others: Reserved
DC_WAY	6:4	D-Cache ways: 0: Direct-mapped 1: 2 way 2: 3 way 3: 4 way 4: 5 way 5: 6 way 6: 7 way 7: 8 way
DC_SET	3:0	D-Cache sets per way: 0: 8 1: 16 2: 32 3: 64 4: 128 5: 256 6: 512 7: 1024 8: 2048 9: 4096 10: 8192 others: Reserved

22.5.28 mtlbcfg_info

This CSR is used to show the TLB configuration information.

Table 22.26: mtlbcfg_info register

Field	Bits	Description
Reserved	MXLEN-1:22	Reserved 0.
DTLB_SIZE	21:19	DTLB size: 0: No DTLB 1: Reserved 2: Reserved 3: Reserved 4: 8 entry 5: 16 entry 6: 32 entry 7: 64 entry

continues on next page

Table 22.26 – continued from previous page

Field	Bits	Description
ITLB_SIZE	18:16	ITLB size: 0: No ITLB 1: Reserved 2: Reserved 3: Reserved 4: 8 entry 5: 16 entry 6: 32 entry 7: 64 entry
Reserved	15:12	Reserved 0.
MTLB_NAPOT	11	TLB supports Svnapot or not: 0: No support 1: Support
MTLB_ECC	10	Main TLB supports ECC or not: 0: No ECC support; 1: Has ECC support.
Reserved	9:7	Reserved 0.
MTLB_WAY	6:4	Main TLB ways: 0: Direct-mapped 1: 2 way 2: 3 way 3: 4 way 4: 5 way 5: 6 way 6: 7 way 7: 8 way
MTLB_WAY_ENTRY	3:0	Main TLB Entry per way: 0: 8 1: 16 2: 32 3: 64 4: 128 5: 256 6: 512 7: 1024 8: 2048 9: 4096 10: 8192 others: Reserved

22.5.29 mppicfg_info

This CSR is used to show the PPI configuration information.

Table 22.27: mppicfg_info register

Field	Bits	Description
PPI_BPA	MXLEN-1:10	The base physical address of PPI. It has to be aligned to multiple of PPI size. For example, to set up PPI of size 4KB starting at address 12KB (0x3000), we simply program DLM_BPA to 0xC (take the upper 22 bits of 0x3000).
Reserved	9:6	Reserved 0.

continues on next page

Table 22.27 – continued from previous page

Field	Bits	Description
PPI_SIZE	5:1	Indicates the size of PPI and it should be power of 2: 0: Reserved 1: 1KB 2: 2KB 3: 4KB 4: 8KB 5: 16KB 6: 32KB 7: 64KB 8: 128KB 9: 256KB 10: 512KB 11: 1MB 12: 2MB 13: 4MB 14: 8MB 15: 16MB 16: 32MB 17: 64MB 18: 128MB 19: 256MB 20: 512MB 21: 1GB 22: 2GB others: Reserved
PPI_EN	0	PPI enable 0: PPI disable 1: PPI enable(Default)

Note: this CSR only exists when PPI is configured.

22.5.30 mfiocfg_info

This CSR is used to show the FIO configuration information.

Table 22.28: mfiocfg_info register

Field	Bits	Description
FIO_BPA	MXLEN-1:10	The base physical address of FIO. It has to be aligned to multiple of FIO size. For example, to set up the FIO of size 4KB starting at address 12KB (0x3000), we simply program DLM_BPA to 0xC (take the upper 22 bits of 0x3000).
Reserved	9:6	Reserved 0.

continues on next page

Table 22.28 – continued from previous page

Field	Bits	Description
FIO_SIZE	5:1	Indicates the size of FIO and it should be power of 2: 0: Reserved 1: 1KB 2: 2KB 3: 4KB 4: 8KB 5: 16KB 6: 32KB 7: 64KB 8: 128KB 9: 256KB 10: 512KB 11: 1MB 12: 2MB 13: 4MB 14: 8MB 15: 16MB 16: 32MB 17: 64MB 18: 128MB 19: 256MB 20: 512MB 21: 1GB 22: 2GB others: Reserved
Reserved	0	Reserved 1.

Note: This CSR only exists when FIO is configured.

22.5.31 sattrib(n)

This CSR is used to set region *n* (0 ~ 7) with base address for Secure S-Mode world to share with Non-Secure S-Mode world. The base address needs to be 4K Byte aligned.

Table 22.29: sattribn register

Field	Bits	Description
Reserved	SXLEN-1:PA_SIZE	Reserved 0.
BPA	PA_SIZE-1:12	Base physical address, the address is 4K byte aligned.
Reserved	11:1	Reserved.
ENA	0	Region enable control. 0: Disable (default); 1: Enable.

22.5.32 sattrim(n)

This CSR is used to set region *n* (0 ~ 7) with address mask, it is used as a set of sattrib(n).

Table 22.30: sattrimn register

Field	Bits	Description
Reserved	MXLEN-1:PA_SIZE	Reserved 0.
MASK	PA_SIZE-1:12	The address mask of this region and it is 4K byte aligned. Reset value is 0.
Reserved	11:0	Reserved 0.

Note:

- Currenlty only 900 v3.2.0 and later versions support this feature, and the number of CSR sets is 8.
- As ENA of *sattribn* is to enable the region, if at this time value of *sattrimn* is 0, it means all address space after BPA is with specific attributes. So user should program *sattrimn* before *sattribn*.
- The higher bits of *sattrimn* should be continuously 1, the left bits should be all 0. Technically the number of 0 means the size of this region.
- The BPA of *sattribn* should be integer multiples of the size of this region. A tip to check a new address is in the region or not: New Address & *sattrimn* == *sattribn* & *sattrimn*? Y, N.

22.5.33 mattrib(n)

This CSR is used to set region *n* with base address and specific attribute. The base address needs to be 4K Byte aligned.

Table 22.31: mattribn register

Field	Bits	Description
Reserved	MXLEN-1:PA_SIZE	Reserved 0.
BPA	PA_SIZE-1:12	Base physical address, the address is 4K byte aligned.
Reserved	11:4	Reserved.
SecShare	3	This region is shareable between secure world and non-secure world or not. 0: Invalid (default); 1: shareable Region.
NOC	2	This region is Noncacheable Region or not. 0: Invalid (default); 1: Noncacheable Region.
Dev	1	This region is Device Region or not. 0: Invalid (default); 1: Device Region. Note: only entry 0 can be set to Device Region.
ENA	0	Region enable control. 0: Disable (default); 1: Enable.

22.5.34 mattrim(n)

This CSR is used to set region *n* with address mask, it is used as a set of mattrib(n).

Table 22.32: mattrimn register

Field	Bits	Description
Reserved	MXLEN-1:PA_SIZE	Reserved 0.
MASK	PA_SIZE-1:12	The address mask of this region and it is 4K byte aligned. Reset value is 0.
Reserved	11:0	Reserved 0.

Note:

- 900 v3.1.0 and later versions support this feature and it has 5 sets CSRs, only entry0 can be used to be set to Device Region. And 900 v3.10.0 and later version, the 5 sets can be used to set to Device/Non-Cacheable/Cacheable Region freely, also the regions can overlap as the prority is Non-Cacheable > Cacheable > Device. 900 v3.11.0 the CSR sets is configurable and up to 8.
- 600 v3.0.0 and later versions support this feature and it has 5 sets CSRs, only entry0 can be used to be set to Device Region. And 900 v3.2.0 and later version, the 5 sets can be used to set to Device/Non-Cacheable/Cacheable Region freely, also the regions can overlap as the prority is Non-Cacheable > Cacheable > Device. 900 v3.3.0 the CSR sets is configurable and up to 8.
- 300 v4.1.0 and later has 2 sets CSR (also called mdevb & mdevm, mnocb & mnocm), entry0 is to set Device Region only and entry 1 is to set Noncacheable Region Only, so the base CSR has only two fileds (BPA and ENA). And 300 v4.9.0 can support up to 8 CSR sets, all sets can be used to set to Device/Non-Cacheable/Cacheable Region freely, also the regions can overlap as the prority is Non-Cacheable > Cacheable > Device.
- As ENA of *mattribn* is to enable the region, if at this time value of *mattrimn* is 0, it means all address space after BPA is with specific attributes. So user should program *mattrimn* before *mattribn*.
- The higher bits of *mattrimn* should be continuously 1, the remaining lower bits should be all 0. Technically the number (N) of 0 means the size of this region(2^N bytes).
- The BPA of *mattribn* should be integer multiples of the size of this region. A tip to check a new address is in the region or not: New Address & *mattrimn* == *mattribn* & *mattrimn*? Y, N.
- If the Dev & NOC bits are 0 and ENA is 1 of *mattribn*, then the region is cacheable.
- For 900 v3.9.1 or ealier version, the non-cacheable/device region defined by the CSR technially is from the cacheable region in Core's RTL Configuration stage, so it should not touch or overlap the IREGION's or other pre-defined regions' address map. Espcially the device region defined by the CSR can't overlap IREGION or software's instruction/data sections, please carefully set the *mattribn* and *mattrimn*.

22.5.35 mirgb_info

This CSR is used to show the IREGION configuration.

Table 22.33: mirgb_info register

Field	Bits	Description
Reserved	MXLEN-1:PA_SIZE+1	Reserved 0.
IRG_BASE_ADDR	PA_SIZE:10	IREGION Base Address
Reserved	9:6	Reserved 0.

continues on next page

Table 22.33 – continued from previous page

Field	Bits	Description
IREGION_SIZE	5:1	Indicates the size of IREGION and it should be power of 2: 0: Reserved 1: 1KB 2: 2KB 3: 4KB 4: 8KB 5: 16KB 6: 32KB 7: 64KB 8: 128KB 9: 256KB 10: 512KB 11: 1MB 12: 2MB 13: 4MB 14: 8MB 15: 16MB 16: 32MB 17: 64MB 18: 128MB 19: 256MB 20: 512MB 21: 1GB 22: 2GB others: Reserved
Reserved	0	Reserved 1.

Note:

- This CSR always exists in Nuclei new verison Core IP, please check related Databook.
- Previously there is a CSR named msmpcfg_info using this CSR ID, as SMP configuration information is inside the IREGION, so msmpcfg_info is discarded.

22.5.36 mecc_lock

This CSR is used to show ECC lock information.

Table 22.34: mecc_lock register

Field	Bits	Description
Reserved	MXLEN-1:1	Reserved 0.
ECC_LOCK	0	To lock ECC related CSRs (mecc_lock, mecc_code [RAMID and SRAMID excluded], ECC field in milm_ctl, mdlm_ctl, mcache_ctl, mtlb_ctl), then cannot be modified: 0: not locked 1: locked

22.5.37 mecc_code

This CSR is used to ECC code injection and ECC error simulation.

Table 22.35: mecc_code register

Field	Bits	Description
Reserved	MXLEN-1:29	Reserved 0.
SRAMID[4:0]	28:24	The ID of RAM that has single bit ECC errors. One of these bits is updated when a single ECC error exception occurs, and can be cleared by software. SRAMID[0]: I-Cache has a single ECC error SRAMID[1]: D-Cache has a single ECC error SRAMID[2]: Main TLB has a single ECC error SRAMID[3]: ILM has a single ECC error SRAMID[4]: DLM has a single ECC error
Reserved	23:21	Reserved 0.
RAMID[4:0]	20:16	The ID of RAM that has double bit ECC errors. One of these bits is updated when a double ECC error exception occurs, and can be cleared by software. RAMID[0]: I-Cache has a doulbe ECC error RAMID[1]: D-Cache has a double ECC error RAMID[2]: Main TLB has a double ECC error RAMID[3]: ILM has a double ECC error RAMID[4]: DLM has a double ECC error
Reserved	[15:7] or [15:8] or [15:9]	Reserved 0.
CODE	[6:0] or [7:0] or [8:0]	ECC code for injection: Max(TLB Width, Data Bus Width) > 64; the width of CODE is 9. Max(TLB Width, Data Bus Width) > 32; the width of CODE is 8. Max(TLB Width, Data Bus Width) <= 32; the width of CODE is 7.

22.5.38 mecc_ctrl

This CSR is only for NA Class Core (just like NA900).

Table 22.36: mecc_ctrl register

Field	Bits	Description
Reserved	MXLEN-1:10	Reserved 0.
DC_CPBK_MSK	9	Write 1 to disable aggregate DCache CPBK ECC fatal error to safety_error output. Reset value is 1.
DC_CCM_MSK	8	Write 1 to disable aggregate DCache CCM ECC fatal error to safety_error output. Reset value is 1.
IC_CCM_MSK	7	Write 1 to disable aggregate ICache CCM ECC fatal error to safety_error output. Reset value is 1.
DLM_EXT_MSK	6	Write 1 to disable aggregate DLM external access ECC fatal error to safety_error output. Reset value is 1.

continues on next page

Table 22.36 – continued from previous page

Field	Bits	Description
ILM_EXT_MSK	5	Write 1 to disable aggregate ILM external access ECC fatal error to safety_error output. Reset value is 1.
DC_ACC_MSK	4	Write 1 to disable aggregate DCache access ECC fatal error to safety_error output. Reset value is 1.
IC_FCH_MSK	3	Write 1 to disable aggregate ICache fetch ECC fatal error to safety_error output. Reset value is 1.
DLM_ACC_MSK	2	Write 1 to disable aggregate DLM access ECC fatal error to safety_error output. Reset value is 1.
ILM_ACC_MSK	1	Write 1 to disable aggregate ILM load/store access ECC fatal error to safety_error output. Reset value is 1.
ILM_FCH_MSK	0	Write 1 to disable aggregate ILM fetch ECC fatal error to safety_error output. Reset value is 1.

22.5.39 mecc_status

This CSR is for ECC status and control.

Table 22.37: mecc_status register

Field	Bits	Description
Reserved	MXLEN-1:10	Reserved 0.
DC_CPBK_ERR	9	DCache CPBK ECC fatal error has occurred. Write 0 to clear, write 1 to set and generate a pulse to corresponding error output. Reset value is 0.
DC_CCM_ERR	8	DCache CCM ECC fatal error has occurred. Write 0 to clear, write 1 to set and generate a pulse to corresponding error output. Reset value is 0.
IC_CCM_ERR	7	ICache CCM ECC fatal error has occurred. Write 0 to clear, write 1 to set and generate a pulse to corresponding error output. Reset value is 0.
DLM_EXT_ERR	6	DLM external access ECC fatal error has occurred. Write 0 to clear, write 1 to set and generate a pulse to corresponding error output. Reset value is 0.
ILM_EXT_ERR	5	ILM external access ECC fatal error has occurred. Write 0 to clear, write 1 to set and generate a pulse to corresponding error output. Reset value is 0.
DC_ACC_ERR	4	DCache access ECC fatal error has occurred. Write 0 to clear, write 1 to set and generate a pulse to corresponding error output. Reset value is 0.
IC_FCH_ERR	3	ICache fetch ECC fatal error has occurred. Write 0 to clear, write 1 to set and generate a pulse to corresponding error output. Reset value is 0.
DLM_ACC_ERR	2	DLM access ECC fatal error has occurred. Write 0 to clear, write 1 to set and generate a pulse to corresponding error output. Reset value is 0.
ILM_ACC_ERR	1	ILM load/store access ECC fatal error has occurred. Write 0 to clear, write 1 to set and generate a pulse to corresponding error output. Reset value is 0.
ILM_FCH_ERR	0	ILM fetch ECC fatal error has occurred. Write 0 to clear, write 1 to set and generate a pulse to corresponding error output. Reset value is 0.

22.5.40 mmacro_dev_en

This CSR is used to disable or enable each Device Region in RTL Configuration Stage.

Table 22.38: mmacro_dev_en register

Field	Bits	Description
Reserved	MXLEN-1:8	Reserved 0
entry7_en	7	Controls to enable for Dev Region entry7 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry7, this field is tied to 0
entry6_en	6	Controls to enable for Dev Region entry6 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry6, this field is tied to 0
entry5_en	5	Controls to enable for Dev Region entry5 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry5, this field is tied to 0
entry4_en	4	Controls to enable for Dev Region entry4 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry4, this field is tied to 0
entry3_en	3	Controls to enable for Dev Region entry3 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry3, this field is tied to 0
entry2_en	2	Controls to enable for Dev Region entry2 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry2, this field is tied to 0
entry1_en	1	Controls to enable for Dev Region entry1 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry1, this field is tied to 0
entry0_en	0	Controls to enable for Dev Region entry0 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry0, this field is tied to 0

22.5.41 mmacro_noc_en

This CSR is used to disable or enable each Non-Cacheable Region in RTL Configuration Stage.

Table 22.39: mmacro_noc_en register

Field	Bits	Description
Reserved	MXLEN-1:8	Reserved 0
entry7_en	7	Controls to enable for Non-Cacheable Region entry7 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry7, this field is tied to 0
entry6_en	6	Controls to enable for Non-Cacheable Region entry6 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry6, this field is tied to 0

continues on next page

Table 22.39 – continued from previous page

Field	Bits	Description
entry5_en	5	Controls to enable for Non-Cacheable Region entry5 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry5, this field is tied to 0
entry4_en	4	Controls to enable for Non-Cacheable Region entry4 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry4, this field is tied to 0
entry3_en	3	Controls to enable for Non-Cacheable Region entry3 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry3, this field is tied to 0
entry2_en	2	Controls to enable for Non-Cacheable Region entry2 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry2, this field is tied to 0
entry1_en	1	Controls to enable for Non-Cacheable Region entry1 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry1, this field is tied to 0
entry0_en	0	Controls to enable for Non-Cacheable Region entry0 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry0, this field is tied to 0

22.5.42 mmacro_ca_en

This CSR is used to disable or enable each Cacheable Region in RTL Configuration Stage.

Table 22.40: mmacro_ca_en register

Field	Bits	Description
Reserved	MXLEN-1:8	Reserved 0
entry7_en	7	Controls to enable for Cacheable Region entry7 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry7, this field is tied to 0
entry6_en	6	Controls to enable for Cacheable Region entry6 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry6, this field is tied to 0
entry5_en	5	Controls to enable for Cacheable Region entry5 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry5, this field is tied to 0
entry4_en	4	Controls to enable for Cacheable Region entry4 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry4, this field is tied to 0
entry3_en	3	Controls to enable for Cacheable Region entry3 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry3, this field is tied to 0
entry2_en	2	Controls to enable for Cacheable Region entry2 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry2, this field is tied to 0

continues on next page

Table 22.40 – continued from previous page

Field	Bits	Description
entry1_en	1	Controls to enable for Cacheable Region entry1 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry1, this field is tied to 0
entry0_en	0	Controls to enable for Cacheable Region entry0 . 0: Disable this entry 1: Enable this entry (Default) Note: When there is no entry0, this field is tied to 0

22.5.43 mtlb_ctl

This CSR is used to control Main TLB related features.

Table 22.41: mtlb_ctl register

Field	Bits	Description
Reserved	MXLEN-1:8	Reserved 0
NAPOT_EN	7	Napot page enable: 0: Disable 1: Enable
TLB_ECC_CHK_EN	6	Controls to check the ECC when core access to MTLB. 0: Disable to check the ECC codes (Default) 1: Enable to check the ECC codes Note: When MMU or ECC is not configured, this field is tied to 0
Reserved	5:4	Reserved.
TLB_DRAM_ECC_INJ_EN	3	Controls to inject the ECC Code in CSR mecc_code to MTLB data rams. This bit can be turned on for injecting ECC errors to test the ECC handler. 0: Disable to inject the ECC codes (Default) 1: Enable to inject the ECC codes Note: When MMU or ECC is not configured, this field is tied to 0.
TLB_TRAM_ECC_INJ_EN	2	Controls to inject the ECC Code in CSR mecc_code to MTLB tag rams. This bit can be turned on for injecting ECC errors to test the ECC handler. 0: Disable to inject the ECC codes (Default) 1: Enable to inject the ECC codes Note: When MMU or ECC is not configured, this field is tied to 0.
TLB_ECC_EXCP_EN	1	MTLB double bit ECC exception enable control. 0: Disable double bit ECC exception 1: Enable double bit ECC exception Note: When MMU or ECC is not configured, this field is tied to 0
TLB_ECC_EN	0	MTLB ECC enable control. 0: Disable ECC 1: Enable ECC (default) Note: When MMU or ECC is not configured, this field is tied to 0.

22.5.44 mfp16mode

This CSR is used to set 16 bit floating precision mode.

Table 22.42: mfp16mode register

Field	Bits	Description
Reserved	MXLEN-1:1	Reserved 0.
fp16mode	0	16 bit float precision mode: 0: normal mode(IEEE-2008 half precision), default value. 1: bfloat 16 mode.

22.5.45 shartid

Nuclei UX900 v2.8.0 or later core implments CSR shartid, the format and features of CSR shartid is same as mhartid, it is convenient for S-mode software to distinguish each hart in a SMP cluster.

23 ECC Introduction

Nuclei processor core can optionally support the ECC protection on ILM, DLM, I-Cache, D-Cache , TLB and Cluster Cache, if configured. This chapter introduces the ECC on ILM, DLM, I-Cache, D-Cache, TLB. For ECC on Cluster Cache, please refer to *SMP and Cluster Cache* (page 333).

23.1 Nuclei ECC mechanism

- SECDED: Single Error Correction, Double Error Detection
- For RV32/64, ECC protection granularity:
  - DLM and D-Cache Data-Ram: 32-bit;
  - ILM and I-Cache Data-Ram: 64-bit;
  - CLM and Cluster Cache Data-RAM: 64-bit;
  - I/D-Cache Tag-Ram ,TLB, Cluster Cache other RAMs: their Actual Size;

Note: For ILM, DLM and CLM, if user wants to initialize them before using, please refer the granularity.

- ECC update policy:
  - Full Write: 32/64-bit data/instruction and corresponding ECC code will be updated simultaneously;
  - Partial Write: Read-Modify-Write sequency will be triggered when 8/16-bit data write operation;
- ECC control policy:
  - ECC_EN, ECC_Exception_EN and ECC_Check_EN can be enable/disable separately on ILM, DLM, I-Cache, D-Cache, TLB and Cluster Cache;
  - User should turn on ECC_EN firstly then turn on ECC_Exception or ECC_Check for a target module;
  - ECC_EN of all related module is default on while ECC_Check_EN is default off;
  - For I-Cache/D-Cache/Cluster Cache with ECC feature, the ECC_EN should be 1 before the related Cache_EN is 1 and not changed while the cache is on, or the result is undefined.
  - 1-bit ECC error will be corrected automatically by hardware and will not trigger ECC exception when ECC is enabled;
  - 2-bit ECC error will trigger ECC exception only when both ECC is enabled and ECC Exception is also enabled, or 2-bit ECC error exception will not be triggered;
- 2-bit ECC error exception cases:
  - Pipeline: IFU instruction fetch will report precise exception, LSU load/store data will report imprecise exception;

  - I-Cache CCM (Control and Maintenance): for by-ADDR operation, store access fault will be reported, mecc_code.RAMID will be set to 1, mtval/stval will be cleared to 0; for INV_ALL operation, I-Cache will be invalidated directly without checking ECC error;
  - D-Cache CCM (Control and Maintenance): for by-ADDR operation, store access fault will be reported, mecc_code.RAMID will be set to 1 (*RAMID.tlb will be set if in VA-PA translation stage, or RAMID.dcache will be set if in D-Cache accessing stage*), mtval/stval will be cleared to 0; the by-ALL operation will be abort if any cacheline has 2-bit ECC error, store access fault will be reported and mecc_code.RAMID.dcache will be set to 1, mtval/stval will be cleared to 0;
  - D-Cache eviction (cpbk): imprecise exception will be reported, mecc_code.RAMID.dcache will be set to 1;
  - Slave Port: 2-bit ECC error will be reported as Bus Error;
  - Debug SBA: 2-bit ECC error will be reported as Bus Error;
  - SFENCE: for by-ADDR operation, store access fault will be reported, mecc_code.RAMID.tlb will be set to 1 and mtval/stval will be cleared to 0; for by-ALL operation, TLB will be invalided directly without checking ECC error;
  - mecc_code.RAMID will set to 1 when 2-bit ECC error occurs on ILM/DLM/I-Cache/D-Cache/TLB; However RAMID is set to 1 does not indicate that ECC exception will be triggered, such as, 2-bit ECC error occurs in Slave Port or Debug SBA, or an instruction fetching has 2-bit ECC error but be flushed later in pipeline;
  - 2-bit ECC error will be indicated in mecc_code.RAMID but not in mdcause, and ECC exception will be reported as access fault but not page fault, so the access fault handler needs to check the mecc_code.RAMID for 2-bit ECC error;
  - 2-bit ECC error signals will be list as output in the CORE top module for SoC integration;
  - ECC exception will be reported if 2-bit ECC error occurs on any way of I-Cache/D-Cache/TLB when in Tag Ram comparing stage, even there is a hit way and this hit way has no 2-bit ECC error;
  - Access fault will be reported other than page fault when TLB has 2-bit ECC error, to make this ECC exception handled in M-mode;

- 1-bit ECC error cases:
  - 1-bit ECC error will be corrected automatically by hardware without triggering exception, and the corrected data will be also updated into the memory (1-bit ECC Scrubbing);
  - mecc_code.SRAMID will be set to 1 when 1-bit ECC error occurs on ILM/DLM/I-Cache/D-Cache/TLB;
  - 1-bit ECC error signals will be list as output in the CORE top module for SoC integration;
- ECC error injection:
  - mecc_code.Code can be selected to update the ILM (ILM is accessible by LSU) and DLM by STORE instruction, without using the ECC code generated by hardware;
  - mecc_code.Code can be selected to update the I-Cache/D-Cache Tag Ram or Data Ram by Linefill when cache miss, without using the ECC code generated by hardware;
  - mecc_code.Code can be selected to update the TLB Tag Ram or Data Ram by Refill when TLB miss, without using the ECC code generated by hardware;
- ECC lock:
  - ECC related CSRs cannot be modified after ECC is locked unless reset, for security;

23.2 Nuclei ECC CSRs

ECC CSRs are all Nuclei customized ones, see below:

Table 23.1: Nuclei ECC CSRs

CSR ADDR	R/W	Name	Description
0xFC0	MRO	micfg_info	ILM/I-Cache configuration info
0xFC1	MRO	mdcfg_info	DLM/D-Cache configuration info
0xFC2	MRO	mcfg_info	Core configuration info
0xFC3	MRO	mtlbcfg_info	TLB configuration info
0x7C0	MRW	milm_ctl	ILM control
0x7C1	MRW	mdlm_ctl	DLM control
0x7C2	MRW	mecc_code	ECC code injection
0x7DD	MRW	mtlb_ctl	TLB control
0x7DE	MRW	mecc_lock	ECC lock
0xbc0	MRW	mecc_ctrl	ECC Control Register
0xbc4	MRW	mecc_satus	ECC Status and Control Register
0x7CA	MRW	mcache_ctl	Cache control

24 Performance Monitor Introduction

Nuclei processor core supports to configure Performance Monitor(called PMU) which is to count on various events for performance profiling. And our implementation of Performance Monitor exactly follows RISC-V Privilege Spec, users also can refer that for details.

PMU implement the RISC-V Sscofpmf extension.

Note: The width of PMU counter is 32bit.

24.1 Performance Monitor CSRs

Table 24.1: Performance Monitor CSRs list

Type	CSR ADDR	RW	Name	Description
CSR	0xB03	MRW	mhpmcounter3	Machine performance monitor counter 3.
	0xB04	MRW	mhpmcounter4	Machine performance monitor counter 4.
	0xB05	MRW	mhpmcounter5	Machine performance monitor counter 5.
	0xB06	MRW	mhpmcounter6	Machine performance monitor counter 6.
	0xB07	MRW	mhpmcounter7	Machine performance monitor counter 7 (tie 0 in our implementation).
	…	…	…	…
	0xB1F	MRW	mhpmcounter31	Machine performance monitor counter 31 (tie 0 in our implementation).
	0xB83	MRW	mhpmcounter3h	Upper 32 bits of mhpmcounter 3 (tie 0 in our implementation), RV32 only.
	0xB84	MRW	mhpmcounter4h	Upper 32 bits of mhpmcounter 4 (tie 0 in our implementation), RV32 only.
	…	…	…	…
	0xB9F	MRW	mhpmcounter31h	Upper 32 bits of mhpmcounter 31 (tie 0 in our implementation), RV32 only.
	0x320	MRW	mcountinhibit	Machine counter-inhibit register.
	0x323	MRW	mhpmevent3	Machine performance monitor event selector 3.
	0x324	MRW	mhpmevent4	Machine performance monitor event selector 4.
	0x325	MRW	mhpmevent5	Machine performance monitor event selector 5.
	0x326	MRW	mhpmevent6	Machine performance monitor event selector 6.
	0x327	MRW	mhpmevent7	Machine performance monitor event selector 7 (tie 0 in our implementation).

continues on next page

Table 24.1 – continued from previous page

Type	CSR ADDR	RW	Name	Description
	…	…	…	…
	0x33F	MRW	mhpmevent31	Machine performance monitor event selector 31 (tie 0 in our implementation).
	0x723	MRW	mhpmevent3h	Upper 32 bits of mhpmevent3, RV32 only.
	0x724	MRW	mhpmevent4h	Upper 32 bits of mhpmevent4, RV32 only.
	0x725	MRW	mhpmevent5h	Upper 32 bits of mhpmevent5, RV32 only.
	0x726	MRW	mhpmevent6	Upper 32 bits of mhpmevent6, RV32 only.
	0x727	MRW	mhpmevent7h	Upper 32 bits of mhpmevent7, RV32 only, tie 0.
	…	…	…	…
	0x73f	MRW	mhpmevent31h	Upper 32 bits of mhpmevent31, RV32 only, tie 0.
	0xC03	URO	hpmcounter3	Supervisor/User mode performance monitor counter 3.
	0xC04	URO	hpmcounter4	Supervisor/User mode performance monitor counter 4.
	…	…	…	…
	0xC1F	URO	hpmcounter31	Supervisor/User mode performance monitor counter 31.
	0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter 3, RV32 only.
	0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter 4, RV32 only.
	…	…	…	…
	0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter 31, RV32 only.

24.1.1 mhpmcounterx

This CSR is used to record specific micro-architecture event number, Note that the width is different for RV32 and RV64.

Table 24.2: mhpmcounterx

Field Name	Bits	RW	Reset Value	Description
Reserved	MXLEN-1:32	RW	0	Tie 0 for RV32, RV64 only.
mhpmcounterx	31:0	RW	0	Machine performance monitor counter x. 3=< x <=6

Table 24.3: mhpmcounterx

Field Name	Bits	RW	Reset Value	Description
Reserved	MXLEN-1:32	R	0	Tie 0, RV64 only.
mhpmcounterx	31:0	RW	0	Tie 0. 7=< x <=31

24.1.2 mhpmcounterhx

This CSR is used to record specific micro-architecture event upper 32 bit number, it is only for RV32.

Table 24.4: mhpmcounterhx

Field Name	Bits	RW	Reset Value	Description
mhpmcounterhx	MXLEN-1:0	R	0	Tie 0, RV32 only. 0=< x <=31

24.1.3 mcountinhibit

The counter-inhibit register mcountinhibit is a XLEN-bit WARL register that controls which of the hardware performance-monitoring counters increment. The settings in this register only control whether the counters increment; Their accessibility is not affected by the setting of this register.

When the CY, IR, or HPMn bit in the mcountinhibit register is clear, the cycle, instret, or hpmcountern register increments as usual. When the CY, IR, or HPMn bit is set, the corresponding counter does not increment.

Table 24.5: mcountinhibit

Field Name	Bits	RW	Reset Value	Description
Reserved	MXLEN-1:7	R	0	-
HPM6	6	RW	0	If set, stop increase performance monitor counter 6.
HPM5	5	RW	0	If set, stop increase performance monitor counter 5.
HPM4	4	RW	0	If set, stop increase performance monitor counter 4.
HPM3	3	RW	0	If set, stop increase performance monitor counter 3.
IR	2	RW	0	If set, stop increase instret counter.
Reserved	1	R	0	-
CY	0	RW	0	If set, stop increase cycle counter.

24.1.4 mhpmeventx

The event selector CSRs, mhpmevent3 - mhpmevent31, are XLEN-bit WARL registers that control which event causes the corresponding counter to increment.

Table 24.6: mhpmevent 3-6

Field Name	Bits	RW	Reset Value	Description
OF	63	RW	0	Overflow status and interrupt disable bit that is set when counter overflows. RV64 Only.
MINH	62	RW	0	If set, the counting of events in M-mode is inhibited. RV64 Only.
SINH	61	RW	0	If set, the counting of events in S/HS-mode is inhibited. RV64 Only.
UINH	60	RW	0	If set, the counting of events in U-mode is inhibited. RV64 Only.
Reserved	59:32	R	0	RV64 Only.
Reserved	31:9	R	0	-
event_idx	8:4	RW	0	Detailed event selector. For instruction commit events please see *Event Selection Value for Instruction Commit Events* (page 150) for more details. For memory access events please see *Event Selection Value for Memory Access Events* (page 150) for more details.
event_sel	3:0	RW	0	0: Select the instruction commit events, such as load, store, bjp ect. See *Event Selection Value for Instruction Commit Events* (page 150) for more details. 1: Select the memory access events, such as icache miss, dcache miss ect. See *Event Selection Value for Memory Access Events* (page 150) for more details.

Note: If XLEN is 32, the high 32bit of hpmevent3-6 are in hpmeventh3-6.

Table 24.7: mhpmevent7-31

Field Name	Bits	RW	Reset Value	Description
Reserved	MXLEN-1:0	R	0	-

Table 24.8: Event Selection Value for Instruction Commit Events

event_idx (mhpeventx[3:0]==0)	Event Name
1	Cycle count
2	Retired instruction count
3	Integer load instruction (includes LR)
4	Integer store instruction (includes SC)
5	Atomic memory operation (do not include LR and SC)
6	System instruction
7	Integer computational instruction(excluding multiplication/division/remainder)
8	Conditional branch
9	Taken conditional branch
10	JAL instruction
11	JALR instruction
12	Return instruction
13	Control transfer instruction (CBR+JAL+JALR)
14	fence instruction(Not include fence.i)
15	Integer multiplication instruction
16	Integer division/remainder instruction
17	Floating-point load instruction
18	Floating-point store instruction
19	Floating-point addition/subtraction
20	Floating-point multiplication
21	Floating-point fused multiply-add (FMADD, FMSUB, FNMSUB, FNMADD)
22	Floating-point division or square-root
23	Other floating-point instruction
24	Conditional branch prediction fail
25	JAL prediction fail
26	JALR prediction fail

Table 24.9: Event Selection Value for Memory Access Events

event_idx (mhpeventx[3:0]==1)	Event Name
1	Icache miss
2	Dcache miss
3	ITLB miss
4	DTLB miss
5	Main TLB miss

25 CCM Mechanism

25.1 Revision History

Rev.	Rev. Date	Revised Section	Revised Content
1.0	2020/4/20	N/A	1.First English version
1.1.0	2021/8/13	1	1.ADD Cluster Cache CCM operations
1.2.0	2021/10/20	1, 3, 4	1.Update the ccm_suen description, which is different in Single Core Configuration and Cluster Configuration.
1.3.0	2022/1/6	4	1.Update the ccm_suen description, which is consistent in Single Core Configuration and Cluster Configuration.
1.4.0	2022/2/11	4	1.Update the ccm_suen description, which clarifies that: a ccm operation can trig an exception only when sen/uen is disabled. 2.Update the INVAL_ALL description: D-Cache INVAL_ALL CMD will be upgraded to be WBINVAL_ALL in U mode if uinvallen is set.

25.2 CCM Mechanism Introduction

CCM (Cache Control and Maintenance) is defined for software to control and maintain the internal L1 I/D-Cache and external Cluster Cache of the core, software can manage the Cache states flexibly to meet the actual application scenarios.

Note: The CCM behavior is valid only if the target L1 I-Cache or D-Cache is enable, or the result is unpredictable.

25.2.1 Nuclei CCM Implementation

Nuclei CCM mechanism uses specific CSRs to control and maintain the Cache. CCM operations have 3 types: by single Address, by ALL and Flush pipeline.

For the 'by single Address' operation, one complete CCM operation flow is:

- Using CSRW to write CSR 'ccm_*x*beginaddr', to specify the ADDR of the CCM operation.
- Using CSRW to write CSR 'ccm_xcommand', to specify the CMD type of the CCM operation, CMD type will be listed in details in below chapter.
- CCM operation will be triggered at the next cycle of the CMD CSR write operation.
- For some operations, such as Lock, using CSRR to read CSR 'ccm_*x*data' to get the result of the CCM operation.

For the 'by ALL' operation, one complete CCM operation flow is:

- Using CSRW to write CSR 'ccm_*x*command', to specify the CMD type of the CCM operation, CMD type will be listed in details in below chapter.
- CCM operation will be triggered at the next cycle of the CMD CSR write operation.

For the 'Flush pipeline' operation, in some senarios it is needed after above two operations to make sure latest instructions or data can be seen by pipeline.

Note: the *x* in ccm_*x*beginaddr, ccm_*x*command and ccm_*x*data can be M/S/U mode, each mode will have its own CSRs.

25.2.2 Nuclei CCM CSRs

Nuclei defines extended CSRs for CCM, which are listed in below table, M/S/U mode has its own CSRs, and the permission of the CCM operations in S/U mode is defined in 'ccm_suen'.

Table 25.2: Nuclei CCM CSRs List

CSR Addr	R/W	Name	Description
0x7CB	MRW	ccm_mbeginaddr	Machine Mode CCM operation ADDR
0x7CC	MRW	ccm_mcommand	Machine Mode CCM operation CMD
0x7CD	MRW	ccm_mdata	Machine Mode CCM operation ReadBack DATA
0x7CE	MRW	ccm_suen	Supervisor/User mode CCM Control
0x5CB	SRW	ccm_sbeginaddr	Supervisor Mode CCM operation ADDR
0x5CC	SRW	ccm_scommand	Supervisor Mode CCM operation CMD
0x5CD	SRW	ccm_sdata	Supervisor Mode CCM operation ReadBack DATA
0x4CB	URW	ccm_ubeginaddr	User Mode CCM operation ADDR
0x4CC	URW	ccm_ucommand	User Mode CCM operation CMD
0x4CD	URW	ccm_udata	User Mode CCM operation ReadBack DATA
0x4CF	URW	ccm_fpipe	Flush Pipeline CMD

25.2.2.1 ccm_*x*beginaddr

ccm_*x*beginaddr is to define the Address of the CCM operation, this Address is treated as VA (Virtual Address), which will be translated to be PA (Physical Address) by MMU (if configured) in pipeline, before accessing Cache.

Note, this CSR's content will INCR one cacheline in Byte automatically by HW after each CCM operation. ($ccm\_xbeginaddr = ccm\_xbeginaddr + One\text{-}Cacheline\text{-}Bytes$)

ccm_*x*beginaddr CSR description is listed below:

Table 25.3: ccm_*x*beginaddr CSR

Field Name	Bits	RW	Reset Value	Description
beginaddr	XLEN-1:0	RW	0	CCM operation ADDR.

25.2.2.2 ccm_*x*command

ccm_*x*command is to define the Command of the CCM operation, after an valid Command is defined, the corresponding operation will be triggered at the next cycle, till a completion sent from Cache. ccm_*x*command CSR description is listed below:

Table 25.4: ccm_*x*command CSR

Field Name	Bits	RW	Reset Value	Description
Reserved	XLEN-1:5	R	0	Reserved
command	4:0	RW	0	CCM operation command.

Command types list in *ccm_command CMD Types* (page 153):

Table 25.5: ccm_command CMD Types

Type	Operation	Codes	Description
I-Cache operation	INVAL	5b'01_000	When I-Cache hit, Unlock and Invalid the specific cacheline in I-Cache; Ignored when I-Cache miss.
	LOCK	5b'01_011	When I-Cache hit, Lock the specific cacheline in I-Cache; When I-Cache miss, Refill the specific cacheline then Lock it in I-Cache. Check ccm_data to know the Lock succeed or not.
	UNLOCK	5b'01_100	When I-Cache hit, Unlock the specific cacheline in I-Cache; Ignored when I-Cache miss.
	INVAL_ALL	5b'01_101	Unlock and Invalid ALL the cacheline in I-Cache.
D-Cache operation	INVAL	5b'00_000	When cache hit, Unlock and Invalid the specific cacheline; Ignored when cache miss. Work on both D-Cache and Cluster Cache, if cluster smp is enable, it will invalid the specified cacheline in the cluster.
	WB	5b'00_001	When cache hit and Dirty, Flush the specific cacheline; When cache miss or hit but not dirty, ignored. Lock bit is not affected. Work on both D-Cache and Cluster Cache, if cluster smp is enable, it will writeback the specified cacheline in the cluster.
	WBINVAL	5b'00_010	When cache hit, Unlock and Flush and Invalid the specific cacheline; Ignored when cache miss. Work on both D-Cache and Cluster Cache, if cluster smp is enable, it will writeback and invalid the specified cacheline in the cluster.
	LOCK	5b'00_011	When cache hit, Lock the specific cacheline; When cache miss, Refill the specific cacheline then Lock it. Check ccm_data to know the Lock succeed or not. Work on D-Cache only.
	UNLOCK	5b'00_100	When cache hit, Unlock the specific cacheline; Ignored when cache miss. Work on D-Cache only.
	INVAL_ALL	5b'10_111	Unlock and Invalid ALL the cacheline. Work on D-Cache only.
	WB_ALL	5b'00_111	Flush ALL the Valid and Dirty cachelines; Lock bit is not affected. Work on D-Cache only.
	WBINVAL_ALL	5b'00_110	Unlock and Flush and Invalid ALL the Valid and Dirty cachelines. Work on D-Cache only.
Cluster Cache operation	Cluster_LOCK	5b'10_011	When Cluster Cache hit, Lock the specific cacheline; When CLuster Cache miss, Refill the specific cacheline then Lock it. Check ccm_data to know the Lock succeed or not. Work on Cluster Cache only.
	Cluster_UNLOCK	5b'10_010	When CLuster Cache hit, Unlock the specific cacheline; Ignored when Cluster Cache miss. Work on Cluster Cache only.

continues on next page

Table 25.5 – continued from previous page

Type	Operation	Codes	Description
	WB_ALL WBINVAL_ALL		Please refer to the Nuclei ISA Spec for more details for the 'WB_ALL/WBINVAL_ALL' CCM operations on Cluster Cache.

All the above list operations are for M mode. Notes:

- ADDR is needed to be specified except for the 'ALL' operations.
- Cacheline is the one hit by the ADDR, the ADDR can point to the head, middle or end of the cacheline.
- S/U mode has the same CMD types as M mode, but for security, D-Cache INVAL_ALL CMD will be upgraded to be WBINVAL_ALL in U mode if uinvallen is set.
- CCM operations can still work on the Disabled Cache.
- Higher privileged mode can operate on the lower privileged mode CCM CSRs to trigger the CCM operations; But 'illegal instruction' exception will trigger when the lower privileged mode operates on higher privileged mode CCM CSRs.

Some special cases for I-Cache CCM operations:

- Permission checking will be done for those 'by ADDR' CCM operations. Permission checking includes: X checking in Page table if VA-PA translation needed, X checking in PMP, X checking in sPMP, Device attribute checking and Non-Cacheable attribute checking. This CCM operation will be ignored if any permission checking fails.
- For LOCK operation, Refill will be triggered if Permission checking passes but Miss in I-Cache, LOCK will fail if Bus Error occurs during Refill and fail info will update into 'ccm_data' CSR register. ECC error (if configured) may occur in Tag Ram when LOCK operation, then LOCK will fail and ECC error info will update into 'sramid' CSR register.
- Software can access the 'ccm_data' CSR register to check the Fail Info details of the LOCK operation, ECC fail info needs to check the 'sramid' CSR register.

Table 25.6: I-Cache Lock Operation Fail Info

Type	Code	Fail Info
I-Cache CCM Operation	0	Lock Succeed
	1	Exceed the Upper entry Num of Lockable way (N-Way I-Cache, Lockable is N-1)
	2	PMP/sPMP/Page-Table X permission check fail, or is Device/Non-Cacheable attribute
	3	Refill has Bus Error
	4	reserved

- No Permission checking will be done on 'INVAL_ALL', the whole I-Cache will be invalidated directly.

Some special cases for D-Cache and Cluster Cache CCM operations:

- Permission checking will be done for those 'by ADDR' CCM operations. Permission checking includes (except for LOCK/UNLOCK/CLuster_LOCK/Cluster_UNLOCK): R checking in Page table if VA-PA translation needed, W checking in PMP, W checking in sPMP, Device attribute checking and Non-Cacheable attribute checking. This CCM operation will be ignored if any permission checking fails or permission checking pass but miss in Cache.
- For LOCK/UNLOCK operation, permission checking will include: R checking in Page table if VA-PA translation needed, R checking in PMP, R checking in sPMP, Device attribute checking and Non-Cacheable attribute checking. This CCM operation will be ignored if any permission checking fails.
- For Cluster_LOCK/Cluster_UNLOCK operation, permission checking will include: R checking in Page table if VA-PA translation needed, X or R checking in PMP, X or R checking in sPMP, Device attribute checking and Non-Cacheable attribute checking. This CCM operation will be ignored if any permission checking fails.
- For LOCK/Cluster_LOCK operation, Refill will be triggered if Permission checking passes but Miss in D-Cache/Cluster Cache, LOCK will fail if Bus Error occurs during Refill and fail info will update into 'ccm_data' CSR register. ECC error (if configured) may occur in Tag Ram when LOCK/Cluster_LOCK operation, then LOCK/Cluster_LOCK will fail and fail info will update into 'sramid' CSR register for LOCK operation or

'CC_FATAL_CNT' register for Cluster_LOCK operation but not 'sramid' CSR register.

- Software can access the 'ccm_data' CSR register to check the Fail Info details of the LOCK/Cluster_LOCK operation.

Table 25.7: D-Cache Lock Operation Fail Info

Type	Code	Fail Info
D-Cache CCM Operation	0	Lock Succeed
	1	Exceed the Upper entry Num of Lockable way (N-Way D-Cache, Lockable is N-1)
	2	PMP/sPMP/Page-Table X permission check fail, or is Device/Non-Cacheable attribute
	3	Refill has Bus Error
	4 ~ 7	Reserved

- For 'INVAL_ALL' operation, PMP/sPMP W checking will be done on each cacheline when 'ALL' operation is enabled under the corresponding mode, if failed, 'INV' will be upgraded to be 'WB+INV' operation. Note: the "INVAL_ALL" in U mode will be automatically upgraded to be "WBINVAL_ALL" if uinvallen is set.
- No permission checking will be done on other 'by ALL' CCM operations if 'ALL' operation is enabled under the corresponding mode.
- WB/Flush will be triggered during some of the CCM operations, if Bus error occurs during WB/Flush, error info will be recorded and report to Core asynchronously; (for 'by ALL' operations, the cacheline in which bus error occurs will be skipped)

25.2.2.3 ccm_*x*data

ccm_*x*data is to record the result of the LOCK operation.

For LOCK operation, 0 in ccm_*x*data indicates succeed, non-zero value indicates failed. I-Cache Lock fail info can refer to *I-Cache Lock Operation Fail Info* (page 154),D-Cache and Cluster Cache Lock fail info can refer to *D-Cache Lock Operation Fail Info* (page 155).

Table 25.8: ccm_*x*data CSR

Field Name	Bits	RW	Reset Value	Description
Reserved	XLEN-1:3	R	0	-
data	2:0	RW	0	CCM DATA.

25.2.2.4 ccm_suen

ccm_suen is to control CCM operations in S/U mode, CCM operations can be permitted only when corresponding bits in ccm_sen/ccm_uen is 1, or 'illegal instruction' exception will be reported. In addition, a specified CCM operation like WB_ALL/INVAL_ALL/WBINVAL_ALL/INVAL/WBINVAL in S/U mode can be enabled only when its corresponding enable bit is set, or this CCM operation will be ignored. This CSR will only exist in M mode when S/U mode is configured.

Table 25.9: ccm_suen CSR

Field Name	Bits	RW	Reset Value	Description
Reserved	XLEN-1:26	R	0	-
swballen	25	RW	0	S-mode WB_ALL enable
uwballen	24	RW	0	U-mode WB_ALL enable
Reserved	23:18	R	0	
sinvallen	17	RW	0	S-mode INVAL_ALL/WBINVAL_ALL enable
uinvallen	16	RW	0	U-mode INVAL_ALL/WBINVAL_ALL enable
Reserved	15:10	R	0	

continues on next page

Table 25.9 – continued from previous page

Field Name	Bits	RW	Reset Value	Description
sinven	9	RW	0	S-mode INVAL/WBINVAL enable
uinven	8	RW	0	U-mode INVAL/WBINVAL enable
Reserved	7:2	R	0	
sen	1	RW	0	S-mode CCM operations enable
uen	0	RW	0	U-mode CCM operations enable

25.2.2.5 ccm_fpipe

ccm_fpipe is to flush the pipeline after CCM operations on Cache, to ensure the latest instructions or data can be seen by pipeline.

ccm_fpipe CSR content in *ccm_fpipe content* (page 156).

Table 25.10: ccm_fpipe content

Field Name	Bits	RW	Reset Value	Description
fpipe	XLEN-1:0	RW	0	CCM Flush Pipeline, Wirte any value to this CSR will trigger pipeline flush, Read this CSR will return 0.

26 Packed-SIMD DSP Introduction

Nuclei processor core can optionally support the DSP features with P extension (Packed-SIMD) instructions.

26.1 Revision History

Rev.	Revision Date	Revised Section	Revised Content
1.5.0	2020/1/20	N/A	1.First version as the full English
2.0.0	2020/4/30	1	1.RISC-V P extension update from v0.5 to v0.5.4
2.5.0	2021/10/29	2	1.Add Nuclei N2 N3 RISC-V P extension instruction
2.5.5	2021/11/01	3	1.Implement EXPD instruction in RV64
2.6.0	2021/12/20	3	1.Update N3's 3 instructions (signed to unsigned)
2.7.0	2023/02/18	9,10	1.Add more N2's instruction (54, Chapter 9.11 ~ 9.64) 2.Add more N3's instruction (14, Chapter 10.37 ~ 10.50)

26.2 Overview of Nuclei SIMD DSP Instructions

The Packed-SIMD DSP of Nuclei Processor Core basically follows the RISC-V "P" Extension Proposal (Version 0.5.4).

Besides RISC-V P Extension, based on Nuclei's customer requests and our DSP domain experiences, Nuclei DSP implements one type of "expansion" default instruction and three configurable extensions.

- One type of "expansion" instruction is to expand a specific byte to a XLEN GPR.
- Three configurable extensions are called N1, N2 and N3. They can only apply to RV32 architecture. The primary advantage of the three subsets is to double the SIMD computation's parallelism as they can use paired 32-bit registers. Brief introduction is as follows:
  - N1 can use register-pairs to do some multiply and add operations. The operand of N1 additional instructions is 8 bit or 16 bit.
  - N2 can use register-pairs to do some multiply operations. The operand of N2 is 32 bit.
  - N3 can use register-pairs to do some multiply-add operations.

For the details of the Nuclei added SIMD DSP instructions, please refer to "Appendix" of this document for more details.

Note:

- The default "expansion" instruction applies to both RV32 and RV64 architectures. In other words, if customer configures DSP module for the core, it always implements the instructions.
- N1, N2, N3 only apply to RV32 architecture. So only N series core can support them. And N2 depends on N1.

- Please refer the Databook in the deliver package to know what extensions are supported, or please contact Nuclei Support.
- About the definition of "register-pairs", please refer the RISC-V P Extension Proposal (Version 0.5.4).

26.3 Introduction of NMSIS

26.3.1 Background

Many micro-controller-based applications can benefit from efficient digital signal processing libraries. In order to quickly and easily handle a variety of complex DSP functions, Nuclei have established a NMSIS DSP library, which is also compatible to ARM open-source CMSIS DSP library, helping users to handle complex DSP calculations on the processor more conveniently.

For more details of the NMSIS, please refer to its online doc from http://doc.nucleisys.com/nmsis.

26.3.2 DSP Library Functions

In NMSIS DSP library, it includes many practical DSP functions. The library is divided into a number of functions each covering a specific category:

- Basic Math Function: Support basic math function, e.g. add, sub, mul, div, etc.
- Fast Math Function: Mainly include sin, cos, sqrt functions, etc.
- Complex Math Function: Mainly include vector calculation and module operation.
- Filter Function: IIR, FIR, LMS, etc.
- Matrix Function: Support matrix calculation.
- Transform Function: Include cfft/ciff, rfft/rifft calculation, etc.
- Motor Control Function: Mainly include PID control functions.
- Statistics Function: Include average, RMS functions.
- Support Function: Include data-copy, transformation between integers and floating-point.
- Interpolation Function: Support interpolation calculation.

The library has separate functions for operating on 8-bit integer, 16-bit integer, 32-bit integer and 32-bit floating-point values. All the library functions are declared in the file riscv_math.h. The functions end with _f32 operating on 32-bit floating-point values. The functions end with _q7, _q15, _q31 operating on integers.

For more details of the library functions, please refer to NMSIS online doc from http://doc.nucleisys.com/nmsis.

26.3.3 DSP Intrinsic Functions

When doing calculation, users can directly call the functions in the NMSIS DSP library to perform efficiently and quickly. When the required functions are not found in the library, users can also directly call the DSP intrinsic functions to meet the requirements and handle related data processing.

For more details of the intrinsic functions, please refer to NMSIS online doc from http://doc.nucleisys.com/nmsis.

26.4 Example of DSP Program

This section will use a simple example to introduce how to set up a project, and operate by calling the NMSIS DSP library.

The example is named as "demo_dsp", in this project, the program want to calculate the average value of the arrays for different data types. As a demo, the program will use the native reference C program and call the DSP library function to calculate the result respectively, and show their results and performance cycles.

Please refer to "application/baremetal/demo_dsp" directory from Nuclei-SDK (https://github.com/Nuclei-Software/nuclei-sdk) for more details about the "demo_dsp" program.

The code structure of this program and the flow of this project are described in detail below.

- demo_dsp.c is the program source code file. The detail of the code is explained below:

```
#define BENCH_INIT()        enter_cycle=__get_rv_cycle(); \
                             printf("CSV, BENCH START, %llu\n", enter_cycle);

#define BENCH_START(func)   start_cycle=__get_rv_cycle();

#define BENCH_END(func)     end_cycle=__get_rv_cycle(); \
                             cycle=end_cycle-start_cycle; \
                             printf("CSV, %s, %llu\n", #func, cycle);

#define BENCH_FINISH()      exit_cycle=__get_rv_cycle(); \
                             cycle=exit_cycle-enter_cycle; \
                             printf("CSV, BENCH END, %llu\n", cycle);

// Defined a comparison function which compares "the result calculated by DSP library"
// with "the result of the reference native C Code".

// Use BENCH_START and BENCH_END macro to record the clock cycles required to execute
// the program, and print the final result.

// In the DSP library, riscv_mean_f32, riscv_mean_q7, riscv_mean_q15, and riscv_mean_q31
// are the averaging functions for 32-bit floating point, 8-bit, 16-bit, and 32-bit
// integers arrays respectively.

// At the same time, use the C Code program to perform the averaging operation, and also
// use BENCH_START and BENCH_END to record the clock cycles required to execute
// the program, and print the result.

// define f32_mean_compare function

void f32_mean_compare()

{

    BENCH_START(riscv_mean_f32);

    riscv_mean_f32(f32_array, ARRAY_SIZE, &f32_out);

    BENCH_END(riscv_mean_f32);

    BENCH_START(ref_mean_f32);

    ref_mean_f32(f32_array, ARRAY_SIZE, &f32_out_ref);

    BENCH_END(ref_mean_f32);

    printf("riscv vs ref: %f, %f\n", f32_out, f32_out_ref);
```

(continues on next page)

(continued from previous page)

```
}
void q7_mean_compare()
{
    BENCH_START(riscv_mean_q7);
    riscv_mean_q7(q7_array, ARRAY_SIZE, &q7_out);
    BENCH_END(riscv_mean_q7);
    BENCH_START(ref_mean_q7);
    ref_mean_q7(q7_array, ARRAY_SIZE, &q7_out_ref);
    BENCH_END(ref_mean_q7);
    printf("riscv vs ref: %d, %d\n", q7_out, q7_out_ref);
}
void q15_mean_compare()
{
    BENCH_START(riscv_mean_q15);
    riscv_mean_q15(q15_array, ARRAY_SIZE, &q15_out);
    BENCH_END(riscv_mean_q15);
    BENCH_START(ref_mean_q15);
    ref_mean_q15(q15_array, ARRAY_SIZE, &q15_out_ref);
    BENCH_END(ref_mean_q15);
    printf("riscv vs ref: %d, %d\n", q15_out, q15_out_ref);
}
void q31_mean_compare()
{
    BENCH_START(riscv_mean_q31);
    riscv_mean_q31(q31_array, ARRAY_SIZE, &q31_out);
    BENCH_END(riscv_mean_q31);
    BENCH_START(ref_mean_q31);
    ref_mean_q31(q31_array, ARRAY_SIZE, &q31_out_ref);
    BENCH_END(ref_mean_q31);
```

(continues on next page)

(continued from previous page)

```
    printf("riscv vs ref: %d, %d\n", q31_out, q31_out_ref);

}

//In main function, the comparison function defined in the previous code is called.
// It will compare the average result and speed calculated by the processor with
// the result and speed calculated by the C Code.

int main(int argc, char **argv)

{

    //......

    BENCH_INIT();

    f32_mean_compare();

    q7_mean_compare();

    q15_mean_compare();

    q31_mean_compare();

    BENCH_FINISH();

    return 0;

}
```

- The ref_mean.c file is an averaging operation program for different data types in C Code, which is used to compare with the results of processor. The code is explained as follows:

```
// Use C code to take the average of input data for different data types such as
// 32-bit floating point, 8-bit, 16-bit, and 32-bit integers value. The results
// will be compared with the results of processor.

// 32-bit floating point average function

void ref_mean_f32(

    float32_t * pSrc,

    uint32_t blockSize,

    float32_t * pResult)

{

    uint32_t i;

    float32_t sum=0;

    for(i=0;i<blockSize;i++)

    {

    sum += pSrc[i];
```

(continues on next page)

(continued from previous page)

```
    }

    *pResult = sum / (float32_t)blockSize;

}

// 32-bit interger average function

void ref_mean_q31(

    q31_t * pSrc,

    uint32_t blockSize,

    q31_t * pResult)

{

    uint32_t i;

    q63_t sum=0;

    for(i=0;i<blockSize;i++)

    {

    sum += pSrc[i];

    }

    *pResult = (q31_t) (sum / (int32_t) blockSize);

}

// 16-bit interger average function

void ref_mean_q15(

    q15_t * pSrc,

    uint32_t blockSize,

    q15_t * pResult)

{

    uint32_t i;

    q31_t sum=0;

    for(i=0;i<blockSize;i++)

    {

    sum += pSrc[i];

    }

    *pResult = (q15_t) (sum / (int32_t) blockSize);
```

(continues on next page)

(continued from previous page)

```
}

// 8-bit interger average function

void ref_mean_q7(

    q7_t * pSrc,

    uint32_t blockSize,

    q7_t * pResult)

{

    uint32_t i;

    q31_t sum=0;

    for(i=0;i<blockSize;i++)

    {

    sum += pSrc[i];

    }

    *pResult = (q7_t) (sum / (int32_t) blockSize);

}
```

- riscv_math.h includes all the functions supported by NMSIS DSP library and provide the name of these functions, the relevant codes are listed as follows:

```
// 8-bit interger average function

   /**

    * @brief Mean value of a Q7 vector.

    * @param[in] pSrc is input pointer

    * @param[in] blockSize is the number of samples to process

    * @param[out] pResult is output value.

    */

  void riscv_mean_q7(

  const q7_t * pSrc,

       uint32_t blockSize,

       q7_t * pResult);

// 16-bit interger average function

   /**
```

(continues on next page)

(continued from previous page)

```
    * @brief Mean value of a Q15 vector.
    * @param[in] pSrc is input pointer
    * @param[in] blockSize is the number of samples to process
    * @param[out] pResult is output value.
    */
  void riscv_mean_q15(
  const q15_t * pSrc,
       uint32_t blockSize,
       q15_t * pResult);
// 32-bit interger average function
   /**
    * @brief Mean value of a Q31 vector.
    * @param[in] pSrc is input pointer
    * @param[in] blockSize is the number of samples to process
    * @param[out] pResult is output value.
    */
  void riscv_mean_q31(
  const q31_t * pSrc,
       uint32_t blockSize,
       q31_t * pResult);
// 32-bit floating point average function
   /**
    * @brief Mean value of a floating-point vector.
    * @param[in] pSrc is input pointer
    * @param[in] blockSize is the number of samples to process
    * @param[out] pResult is output value.
    */
  void riscv_mean_f32(
  const float32_t * pSrc,
```

(continues on next page)

(continued from previous page)

```
        uint32_t blockSize,

        float32_t * pResult);
```

- The above part of code is the functions used in this example. riscv_mean_f32, riscv_mean_q7, riscv_mean_q15, riscv_mean_q31 are average functions for 32-bit floating point, 8-bit, 16-bit, and 32-bit fixed-point values. When users need to use other functions, users can also find their corresponding function names in the riscv_math.h header file.
- If there are no available functions to meet the requirements, the user can directly call the DSP intrinsic functions. The intrinsic functions can be found in the *core_feature_dsp.h* header file. The example segment code of the intrinsic functions is as below:

```
// kabs8 (simd 8-bit saturating absolute)

__STATIC_FORCEINLINE unsigned long __RV_KABS8(unsigned long a)

{

    unsigned long result;

    __ASM volatile("kabs8 %0, %1" : "=r"(result) : "r"(a));

    return result;

}
```

About how to run the *demo_dsp* program, please refer to Nuclei-SDK (https://github.com/Nuclei-Software/nuclei-sdk) for more details. After running the program, the printout message on the serial port is as shown in *Printout message after running demo_dsp program* (page 165), the terminal prints the result calculated by the averaging function from NMSIS DSP library and the result of the C Code.

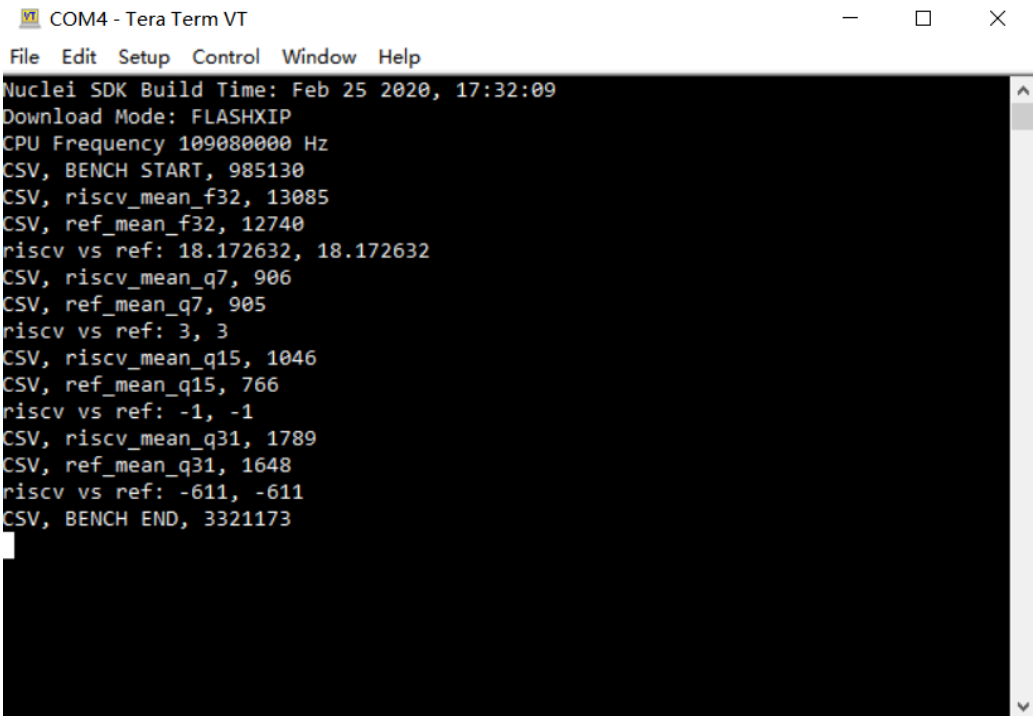


Fig. 26.1: Printout message after running demo_dsp program

26.5 Appendix A: Nuclei Default SIMD DSP Additional Instruction

26.5.1 EXPD80, EXPD81, EXPD82, EXPD83, EXPD84, EXPD85, EXPD86, EXPD87

Type: DSP

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
EXPD 0010010	xxxxx	Rs1	111	Rd	GE80B 1111111

Instr	xxxxx
EXPD80	00000
EXPD81	00001
EXPD82	00010
EXPD83	00011
EXPD84	00100
EXPD85	00101
EXPD86	00110
EXPD87	00111

Syntax:

```
EXPD80 Rd, Rs1

EXPD81 Rd, Rs1

EXPD82 Rd, Rs1

EXPD83 Rd, Rs1

EXPD84 Rd, Rs1

EXPD85 Rd, Rs1

EXPD86 Rd, Rs1

EXPD87 Rd, Rs1
```

Purpose:

When RV32, copy 8-bit data from 32-bit chunks into 4 bytes in a register; When RV64, copy 8-bit data from 64-bit chunks into 8 bytes in a register.

Description:

RV32:

EXPD80 Expand and Copy Byte 0 to 32 bit

EXPD81 Expand and Copy Byte 1 to 32 bit

EXPD82 Expand and Copy Byte 2 to 32 bit

EXPD83 Expand and Copy Byte 3 to 32 bit

RV64:

EXPD80 Expand and Copy Byte 0 to 64 bit

EXPD81 Expand and Copy Byte 1 to 64 bit

EXPD82 Expand and Copy Byte 2 to 64 bit

EXPD83 Expand and Copy Byte 3 to 64 bit

EXPD84 Expand and Copy Byte 4 to 64 bit

EXPD85 Expand and Copy Byte 5 to 64 bit

EXPD86 Expand and Copy Byte 6 to 64 bit

EXPD87 Expand and Copy Byte 7 to 64 bit

Operations:

```
RV32:

Rd.W[x][31:0] = CONCAT(Rs1.B[0][7:0], Rs1.B[0][7:0], Rs1.B[0][7:0],
Rs1.B[0][7:0]);//EXPD80

Rd.W[x][31:0] = CONCAT(Rs1.B[1][7:0], Rs1.B[1][7:0], Rs1.B[1][7:0],
Rs1.B[1][7:0]);//EXPD81

Rd.W[x][31:0] = CONCAT(Rs1.B[2][7:0], Rs1.B[2][7:0], Rs1.B[2][7:0],
Rs1.B[2][7:0]);//EXPD82

Rd.W[x][31:0] = CONCAT(Rs1.B[3][7:0], Rs1.B[3][7:0], Rs1.B[3][7:0],
Rs1.B[3][7:0]);//EXPD83

X=0

RV64:

Rd.W[x][31:0] = CONCAT(Rs1.B[0][7:0], Rs1.B[0][7:0], Rs1.B[0][7:0],
Rs1.B[0][7:0]);//EXPD80

Rd.W[x][31:0] = CONCAT(Rs1.B[1][7:0], Rs1.B[1][7:0], Rs1.B[1][7:0],
Rs1.B[1][7:0]);//EXPD81

Rd.W[x][31:0] = CONCAT(Rs1.B[2][7:0], Rs1.B[2][7:0], Rs1.B[2][7:0],
Rs1.B[2][7:0]);//EXPD82

Rd.W[x][31:0] = CONCAT(Rs1.B[3][7:0], Rs1.B[3][7:0], Rs1.B[3][7:0],
Rs1.B[3][7:0]);//EXPD83

Rd.W[x][31:0] = CONCAT(Rs1.B[4][7:0], Rs1.B[4][7:0], Rs1.B[4][7:0],
Rs1.B[4][7:0]);//EXPD84

Rd.W[x][31:0] = CONCAT(Rs1.B[5][7:0], Rs1.B[5][7:0], Rs1.B[5][7:0],
Rs1.B[5][7:0]);//EXPD85

Rd.W[x][31:0] = CONCAT(Rs1.B[6][7:0], Rs1.B[6][7:0], Rs1.B[6][7:0],
Rs1.B[6][7:0]);//EXPD86

Rd.W[x][31:0] = CONCAT(Rs1.B[7][7:0], Rs1.B[7][7:0], Rs1.B[7][7:0],
Rs1.B[7][7:0]);//EXPD87

X=0,1
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

EXPD80:

```
unsigned long __expd80(unsigned long a);

RV32:

uint8x4_t __v_expd80(uint8x4_t a);

RV64:

uint8x8_t __v_expd80(uint8x8_t a);
```

EXPD81:

```
unsigned long __expd81(unsigned long a);

// RV32:

uint8x4_t __v_expd81(uint8x4_t a);

// RV64:

uint8x8_t __v_expd81(uint8x8_t a);
```

EXPD82:

```
unsigned long __expd82(unsigned long a);

// RV32:

uint8x4_t __v_expd82(uint8x4_t a);

// RV64:

uint8x8_t __v_expd82(uint8x8_t a);
```

EXPD83:

```
unsigned long __expd83(unsigned long a);

// RV32:

uint8x4_t __v_expd83(uint8x4_t a);

// RV64:

uint8x8_t __v_expd83(uint8x8_t a);
```

EXPD84:

```
unsigned long __expd84(unsigned long a);

uint8x8_t __v_expd84(uint8x8_t a);
```

EXPD85:

```
unsigned long __expd85(unsigned long a);

uint8x8_t __v_expd85(uint8x8_t a);
```

EXPD86:

```
unsigned long __expd86(unsigned long a);

uint8x8_t __v_expd86(uint8x8_t a);
```

EXPD87:

```
unsigned long __expd87(unsigned long a);

uint8x8_t __v_expd87(uint8x8_t a);
```

26.6 Appendix B: Nuclei N1 SIMD DSP Additional Instruction

26.6.1 DKHM8 (64-bit SIMD Signed Saturating Q7 Multiply)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKHM8 1000111	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKHM8 Rd, Rs1, Rs2
```

Purpose:

Do Q7xQ7 element multiplications simultaneously. The Q14 results are then reduced to Q7 numbers again.

Description:

For the "DKHM8" instruction, multiply the top 8-bit Q7 content of 16-bit chunks in Rs1 with the top 8-bit Q7 content of 16-bit chunks in Rs2. At the same time, multiply the bottom 8-bit Q7 content of 16-bit chunks in Rs1 with the bottom 8-bit Q7 content of 16-bit chunks in Rs2.

The Q14 results are then right-shifted 7-bits and saturated into Q7 values. The Q7 results are then written into Rd. When both the two Q7 inputs of a multiplication are 0x80, saturation will happen. The result will be saturated to 0x7F and the overflow flag OV will be set.

Operations:

```
op1t = Rs1.B[x+1]; op2t = Rs2.B[x+1]; // top

op1b = Rs1.B[x]; op2b = Rs2.B[x]; // bottom


for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {

  if (0x80 != aop | 0x80 != bop) {

    res = (aop s* bop) >> 7;

  } else {

    res= 0x7F;

    OV = 1;

  }
```

(continues on next page)

(continued from previous page)

```
}

Rd.H[x/2] = concat(rest, resb);

x=0,2,4,6
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkhm8(unsigned long long a, unsigned long long b);

int8x8_t __v_dkhm8(int8x8_t a, int8x8_t b);
```

26.6.2 DKHM16 (64-bit SIMD Signed Saturating Q15 Multiply)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKHM16 1000011	RS2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKHM16 Rd, Rs1, Rs2
```

Purpose:

Do Q15xQ15 element multiplications simultaneously. The Q30 results are then reduced to Q15 numbers again.

Description:

Operations:

```
op1t = Rs1.H[x+1]; op2t = Rs2.H[x+1]; // top

op1b = Rs1.H[x]; op2b = Rs2.H[x]; // bottom


for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {

  if (0x8000 != aop | 0x8000 != bop) {

    res = (aop s* bop) >> 15;

  } else {

    res= 0x7FFF;

    OV = 1;

  }

}
```

(continues on next page)

(continued from previous page)

```
Rd.W[x/2] = concat(rest, resb);

x=0,2
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkhm16(unsigned long long a, unsigned long long b);

int16x4_t __v_dkhm16(int16x4_t a, int16x4_t b);
```

26.6.3 DKABS8 (64-bit SIMD 8-bit Saturating Absolute)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
ONEOP 1010110	DKABS8 10000	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKABS8 Rd, Rs1
```

Purpose:

Get the absolute value of 8-bit signed integer elements simultaneously.

Description:

This instruction calculates the absolute value of 8-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x80, this instruction generates 0x7f as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.B[x];

if (src == 0x80) {

  src = 0x7f;

  OV = 1;

} else if (src[7] == 1)

  src = -src;

}

Rd.B[x] = src;

x=7...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkabs8(unsigned long long a);

int8x8_t __v_dkabs8(int8x8_t a);
```

26.6.4 DKABS16 (64-bit SIMD 16-bit Saturating Absolute)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
ONEOP 1010110	DKABS16 10001	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKABS16 Rd, Rs1
```

Purpose:

Get the absolute value of 16-bit signed integer elements simultaneously.

Description:

This instruction calculates the absolute value of 16-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x8000, this instruction generates 0x7fff as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.H[x];

if (src == 0x8000) {

  src = 0x7fff;

  OV = 1;

} else if (src[15] == 1)

  src = -src;

}

Rd.H[x] = src;

x=3...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkabs16(unsigned long long a);

int16x4_t __v_dkabs16(int16x4_t a);
```

26.6.5 DKSLRA8 (64-bit SIMD 8-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKSLRA8 0101111	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKSLRA8 Rd, Rs1, Rs2
```

Purpose:

Do 8-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q7 saturation for the left shift.

Description:

The 8-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[3:0]. Rs2[3:0] is in the signed range of [-2^3 , 2^3-1]. A positive Rs2[3:0] means logical left shift and a negative Rs2[3:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[3:0]. However, the behavior of "Rs2[3:0]==-2^3 (0x8)" is defined to be equivalent to the behavior of "Rs2[3:0]==-(2^3-1) (0x9)".

Operations:

```
if (Rs2[3:0] < 0) {

  sa = -Rs2[3:0];

  sa = (sa == 8)? 7 : sa;

  Rd.B[x] = SE8(Rs1.B[x][7:sa]);

} else {

  sa = Rs2[2:0];

  res[(7+sa):0] = Rs1.B[x] <<(logic) sa;

  if (res > (2^7)-1) {

    res[7:0] = 0x7f; OV = 1;

  } else if (res < -2^7) {

    res[7:0] = 0x80; OV = 1;

  }

  Rd.B[x] = res[7:0];

}

x=7...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkslra8(unsigned long long a, int b);

int8x8_t __v_dkslra8(int8x8_t a, int b);
```

26.6.6 DKSLRA16 (64-bit SIMD 16-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKSLRA16 0101011	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKSLRA16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q15 saturation for the left shift.

Description:

The 16-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[4:0]. Rs2[4:0] is in the signed range of [-2^4, 2^4-1]. A positive Rs2[4:0] means logical left shift and a negative Rs2[4:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[4:0]. However, the behavior of "Rs2[4:0]==-2^4 (0x10)" is defined to be equivalent to the behavior of "Rs2[4:0]==-(2^4-1) (0x11)".

Operations:

```
if (Rs2[4:0] < 0) {

  sa = -Rs2[4:0];

  sa = (sa == 16)? 15 : sa;

  Rd.H[x] = SE16(Rs1.H[x][15:sa]);

} else {

  sa = Rs2[3:0];

  res[(15+sa):0] = Rs1.H[x] <<(logic) sa;

  if (res > (2^15)-1) {

    res[15:0] = 0x7fff; OV = 1;

  } else if (res < -2^15) {

    res[15:0] = 0x8000; OV = 1;

  }

  d.H[x] = res[15:0];

}
```

(continues on next page)

(continued from previous page)

```
x=3...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkslra16(unsigned long long a, int b);

int16x4_t __v_dkslra16(int16x4_t a, int b);
```

26.6.7 DKADD8 (64-bit SIMD 8-bit Signed Saturating Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKADD8 0001100	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKADD8 Rd, Rs1, Rs2
```

Purpose:

Do 8-bit signed integer element saturating additions simultaneously.

Description:

This instruction adds the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2. If any of the results are beyond the Q7 number range (-2^7 <= Q7 <= 2^7-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] + Rs2.B[x];

if (res[x] > 127) {

  res[x] = 127;

  OV = 1;

} else if (res[x] < -128) {

  res[x] = -128;

  OV = 1;

}

Rd.B[x] = res[x];

x=7...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkadd8(unsigned long long a, unsigned long long b);

int8x8_t __v_dkadd8(int8x8_t a, int8x8_t b);
```

26.6.8 DKADD16 (64-bit SIMD 16-bit Signed Saturating Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKADD16 0001000	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKADD16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit signed integer element saturating additions simultaneously.

Description:

This instruction adds the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2. If any of the results are beyond the Q15 number range (-2^15 <= Q15 <= 2^15-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] + Rs2.H[x];

if (res[x] > 32767){

  res[x] = 32767;

  OV = 1;

} else if (res[x] < -32768) {

  res[x] = -32768;

  OV = 1;

}

Rd.H[x] = res[x];

x=3...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkadd16(unsigned long long a, unsigned long long b);

int16x4_t __v_dkadd16(int16x4_t a, int16x4_t b);
```

26.6.9 DKSUB8 (64-bit SIMD 8-bit Signed Saturating Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKSUB8 0001101	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKSUB8 Rd, Rs1, Rs2
```

Purpose:

Do 8-bit signed elements saturating subtractions simultaneously.

Description:

This instruction subtracts the 8-bit signed integer elements in Rs2 from the 8-bit signed integer elements in Rs1. If any of the results are beyond the Q7 number range (-2^7 <= Q7 <= 2^7-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] - Rs2.B[x];

if (res[x] > (2^7)-1) {

  res[x] = (2^7)-1;

  OV = 1;

} else if (res[x] < -2^7) {

  res[x] = -2^7;

  OV = 1;

}

Rd.B[x] = res[x];

x=7...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dksub8(unsigned long long a, unsigned long long b);

int8x8_t __v_dksub8(int8x8_t a, int8x8_t b);
```

26.6.10 DKSUB16 (64-bit SIMD 16-bit Signed Saturating Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKSUB16 0001001	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKSUB16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit signed integer elements saturating subtractions simultaneously.

Description:

This instruction subtracts the 16-bit signed integer elements in Rs2 from the 16-bit signed integer elements in Rs1. If any of the results are beyond the Q15 number range (-2^15 <= Q15 <= 2^15-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] - Rs2.H[x];

if (res[x] > (2^15)-1) {

  res[x] = (2^15)-1;

  OV = 1;

} else if (res[x] < -2^15) {

  res[x] = -2^15;

  OV = 1;

}

Rd.H[x] = res[x];

x=3...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dksub16(unsigned long long a, unsigned long long b);

int16x4_t __v_dksub16(int16x4_t a, int16x4_t b);
```

26.7 Appendix C: Nuclei N2 SIMD DSP Additional Instruction

26.7.1 DKHMX8 (64-bit SIMD Signed Crossed Saturating Q7 Multiply)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKHMX8 0000000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKHMX8 Rd, Rs1, Rs2
```

Purpose:

Do Q7xQ7 element crossed multiplications simultaneously. The Q15 results are then reduced to Q7 numbers again.

Description:

Operations:

```
 op1t = Rs1.B[x+1]; op2t = Rs2.B[x]; // top

op1b = Rs1.B[x]; op2b = Rs2.B[x+1]; // bottom

for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {

   if (0x80 != aop | 0x80 != bop) {

       res = (aop s* bop) >> 7;

   } else {

       res= 0x7F;

       OV = 1;
   }
}

Rd.H[x/2] = concat(rest, resb);

x=0,2,4,6
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkhmx8(unsigned long long a, unsigned long long b);

int8x8_t __v_dkhmx8(int8x8_t a, int8x8_t b);
```

26.7.2 DKHMX16 (64-bit SIMD Signed Crossed Saturating Q15 Multiply)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKHMX16 0000001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKHMX16 Rd, Rs1, Rs2
```

Purpose:

Do Q15xQ15 element crossed multiplications simultaneously. The Q31 results are then reduced to Q15 numbers again.

Description:

Operations:

```
op1t = Rs1.H[x+1]; op2t = Rs2.H[x]; // top

op1b = Rs1.H[x]; op2b = Rs2.H[x+1]; // bottom

for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {

   if (0x8000 != aop | 0x8000 != bop) {

       res = (aop s* bop) >> 15;

   } else {

       res= 0x7FFF;

       OV = 1;
   }
}

Rd.W[x/2] = concat(rest, resb);

x=0,2
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkhmx16(unsigned long long a, unsigned long long b);

int16x4_t __v_dkhmx16(int16x4_t a, int16x4_t b);
```

26.7.3 DSMMUL (64-bit MSW 32x32 Signed Multiply)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMMUL 0000010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMMUL Rd, Rs1, Rs2
```

Purpose:

Do MSW 32x32 element signed multiplications simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {

  res = (aop s* bop)[63:32];

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsmmul(unsigned long long a, unsigned long long b);

int32x2_t __v_dsmmul(int32x2_t a, int32x2_t b);
```

26.7.4 DSMMULU (64-bit MSW 32x32 Unsigned Multiply)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMMULU 0000011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMMUL.U Rd, Rs1, Rs2
```

Purpose:

Do MSW 32x32 element unsigned multiplications simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {

  res = RUND(aop u* bop)[63:32];

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsmmulu(unsigned long long a, unsigned long long b);

int32x2_t __v_dsmmulu(int32x2_t a, int32x2_t b);
```

26.7.5 DKWMMUL (64-bit MSW 32x32 Signed Multiply & Double)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKWMMUL 0000100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKWMMUL Rd, Rs1, Rs2
```

Purpose:

Do MSW 32x32 element signed multiplications simultaneously and double. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
```

(continues on next page)

(continued from previous page)

```
  res = sat.q31((aop s* bop) << 1)[63:32];

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkwmmul(unsigned long long a, unsigned long long b);

int32x2_t __v_dkwmmul(int32x2_t a, int32x2_t b);
```

26.7.6 DKWMMULU (64-bit MSW 32x32 Unsigned Multiply & Double)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKWMMULU 0000101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKWMMUL.U Rd, Rs1, Rs2
```

Purpose:

Do MSW 32x32 element unsigned multiplications simultaneously and double. The results are written into Rd.

Descriptions:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {

  res = sat.q31(RUND(aop u* bop) << 1)[63:32];

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkwmmulu(unsigned long long a, unsigned long long b);

int32x2_t __v_dkwmmulu(int32x2_t a, int32x2_t b);
```

26.7.7 DKABS32 (64-bit SIMD 32-bit Saturating Absolute)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKABS32 0000110	DKABS32 00000	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKABS32 Rd, Rs1
```

Purpose:

Get the absolute value of 32-bit signed integer elements simultaneously.

Description:

This instruction calculates the absolute value of 32-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x8000_0000, this instruction generates 0x7fff_ffff as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.W[x];

if (src == 0x8000_0000) {

    src = 0x7fff_ffff;

    OV = 1;

} else if (src[31] == 1)

    src = -src;

}

Rd.W[x] = src;

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkabs32(unsigned long long a);

int32x2_t __v_dkabs32(int32x2_t a);
```

26.7.8 DKSLRA32 (64-bit SIMD 32-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKSLRA32 0000111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKSLRA32 Rd, Rs1, Rs2
```

Purpose:

Do 31-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q31 saturation for the left shift.

Description:

The 31-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[5:0]. Rs2[5:0] is in the signed range of [-2^5, 2^5-1]. A positive Rs2[5:0] means logical left shift and a negative Rs2[4:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[5:0]. However, the behavior of "Rs2[5:0]==-2^5 (0x20)" is defined to be equivalent to the behavior of "Rs2[5:0]==-(2^5-1) (0x21)".

Operations:

```
if (Rs2[5:0] < 0) {

    sa = -Rs2[5:0];

    sa = (sa == 32)? 31 : sa;

    Rd.W[x] = SE32(Rs1.W[x][31:sa]);

} else {

    sa = Rs2[4:0];

    res[(31+sa):0] = Rs1.W[x] <<(logic) sa;

if (res > (2^31)-1) {

    res[31:0] = 0x7fff_ffff; OV = 1;

} else if (res < -2^31) {

    res[31:0] = 0x8000_0000; OV = 1;

  }

  Rd.W[x] = res[31:0];

}

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkslra32(unsigned long long a, int b);

int32x2_t __v_dkslra32(int32x2_t a, int b);
```

26.7.9 DKADD32(64-bit SIMD 32-bit Signed Saturating Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKADD32 0001000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKADD32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit signed integer element saturating additions simultaneously.

Description:

This instruction adds the 32-bit signed integer elements in Rs1 with the 32-bit signed integer elements in Rs2. If any of the results are beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.W[x] + Rs2.W[x];

if (res[x] > 0x7fff_ffff) {

    res[x] = 0x7fff_ffff;

    OV = 1;

} else if (res[x] < 0x8000_0000) {

    res[x] = 0x8000_0000;

    OV = 1;

}

Rd.W[x] = res[x];

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkadd32(unsigned long long a, unsigned long long b);

int32x2_t __v_dkadd32(int32x2_t a, int32x2_t b);
```

26.7.10 DKSUB32(64-bit SIMD 32-bit Signed Saturating Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKSUB32 0001001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKSUB32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit signed integer elements saturating subtractions simultaneously.

Description:

This instruction subtracts the 32-bit signed integer elements in Rs2 from the 32-bit signed integer elements in Rs1. If any of the results are beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.W[x] - Rs2.W[x];

if (res[x] > (2^31)-1) {

    res[x] = (2^31)-1;

    OV = 1;

} else if (res[x] < -2^31) {

    res[x] = -2^31;

    OV = 1;

}

Rd.W[x] = res[x];

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dksub32(unsigned long long a, unsigned long long b);

int32x2_t __v_dksub32(int32x2_t a, int32x2_t b);
```

26.7.11 DRADD16(64-bit SIMD 16-bit Halving Signed Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DRADD16 0101110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DRADD16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit signed integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description:

This instruction adds the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.H[x] = [(Rs1.H[x]) + (Rs2.H[x])] s>> 1;

x=3...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dradd16(unsigned long long a, unsigned long long b);

int16x4_t __v_dradd16(int16x4_t a, int16x4_t b);
```

26.7.12 DSUB16(64-bit SIMD 16-bit Halving Signed Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSUB16 0101111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSUB16  Rd, Rs1, Rs2
```

Purpose:

Do 16-bit integer element subtractions simultaneously.

Description:

This instruction adds the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.H[x] = [(Rs1.H[x]) - (Rs2.H[x])] ;

x=3...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsub16(unsigned long long a, unsigned long long b);

int16x4_t __v_dsub16(int16x4_t a, int16x4_t b);
```

26.7.13 DRADD32(64-bit SIMD 32-bit Halving Signed Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DRADD32 0110000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DRADD32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit signed integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description:

This instruction adds the 32-bit signed integer elements in Rs1 with the 32-bit signed integer elements in Rs2. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.W[x] = [(Rs1.W[x]) + (Rs2.W[x])] s>> 1;

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dradd32(unsigned long long a, unsigned long long b);

int32x2_t __v_dradd32(int32x2_t a, int32x2_t b);
```

26.7.14 DSUB32(64-bit SIMD 32-bit Halving Signed Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSUB32 0110001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSUB32  Rd, Rs1, Rs2
```

Purpose:

Do 32-bit integer element subtractions simultaneously.

Description:

This instruction subtracts the 32-bit signed integer elements in Rs2 from the 32-bit signed integer elements in Rs1 . The results are written to Rd.

Operations:

```
Rd.W[x] = [(Rs1.E[x]) - (Rs2.E[x])] ;

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsub32(unsigned long long a, unsigned long long b);

int32x2_t __v_dsub32(int32x2_t a, int32x2_t b);
```

26.7.15 DMSR16(Signed Multiply Halfs with Right Shift 16-bit and Cross Multiply Halfs with Right Shift 16-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DMSR16 1011111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DMSR16  Rd, Rs1, Rs2
```

Purpose:

Do two signed 16-bit multiplications and cross multiplications from the 16-bit elements of two registers; and each multiplications performs a right shift operation.

Description:

For the "DMSR16" instruction, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the top 16-bit Q15 content of 32-bit chunks in Rs2, multiply the bottom 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom 16-bit Q15 content

of 32-bit chunks in Rs2.

At the same time, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom16-bit Q15 content of 32-bit chunks in Rs2 and multiply the bottom16-bit Q15 content of 32-bit chunks in Rs1 with the top16-bit Q15 content of 32-bit chunks in Rs2. The Q31 results are then right-shifted 16-bits and clipped to Q15 values. The Q15 results are then written into Rd.

Operations:

```
Rd.H[0] = (Rs1.H[0] s* Rs2.H[0]) s>> 16

Rd.H[1] = (Rs1.H[1] s* Rs2.H[1]) s>> 16

Rd.H[2] = (Rs1.H[1] s* Rs2.H[0]) s>> 16

Rd.H[3] = (Rs1.H[0] s* Rs2.H[1]) s>> 16
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __DMSR16(unsigned long a, unsigned long b);

int16x4_t __v_dmsr16(int16x2_t a, int16x2_t b);
```

26.7.16 DMSR17(Signed Multiply Halfs with Right Shift 17-bit and Cross Multiply Halfs with Right Shift 17-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DMSR17 1100000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DMSR17  Rd, Rs1, Rs2
```

Purpose:

Do two signed 16-bit multiplications and cross multiplications from the 16-bit elements of two registers; and each multiplications performs a right shift operation.

Description:

For the "DMSR17" instruction, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the top 16-bit Q15 content of 32-bit chunks in Rs2, multiply the bottom 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom 16-bit Q15 content of 32-bit chunks in Rs2.

At the same time, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom16-bit Q15 content of 32-bit chunks in Rs2 and multiply the bottom16-bit Q15 content of 32-bit chunks in Rs1 with the top16-bit Q15 content of 32-bit chunks in Rs2. The Q31 results are then right-shifted 17-bits and clipped to Q15 values. The Q15 results are then written into Rd.

Operations:

```
Rd.H[0] = (Rs1.H[0] s* Rs2.H[0]) s>> 17
```

(continues on next page)

(continued from previous page)

```
Rd.H[1] = (Rs1.H[1] s* Rs2.H[1]) s>> 17

Rd.H[2] = (Rs1.H[1] s* Rs2.H[0]) s>> 17

Rd.H[3] = (Rs1.H[0] s* Rs2.H[1]) s>> 17
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __DMSR17(unsigned long a, unsigned long b);

int16x4_t __v_dmsr17(int16x2_t a, int16x2_t b);
```

26.7.17 DMSR33(Signed Multiply with Right Shift 33-bit and Cross Multiply with Right Shift 33-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DMSR33 1100001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DMSR33  Rd, Rs1, Rs2
```

Purpose:

Do two signed 32-bit multiplications from the 32-bit elements of two registers, and each multiplications performs a right shift operation.

Description:

For the "DMSR33"instruction, multiply the top 32-bit Q31 content of 64-bit chunks in Rs1 with the top 32-bit Q31 content of 64-bit chunks in Rs2. At the same time, multiply the bottom 32-bit Q31 content of 64-bit chunks in Rs1 with the bottom 32-bit Q31 content of 64-bit chunks in Rs2.

The Q63 results are then right-shifted 33-bits and clipped to Q31 values. The Q31 results are then written into Rd.

Operations:

```
Rd.W[0] = (Rs1.W[0] s* Rs2.W[0]) s>> 33

Rd.W[1] = (Rs1.W[1] s* Rs2.W[1]) s>> 33
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dmsr33(unsigned long long a, unsigned long long b);

int32x2_t __v_dmsr33(int32x2_t a, int32x2_t b);
```

26.7.18 DMXSR33(Signed Multiply with Right Shift 33-bit and Cross Multiply with Right Shift 33-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DMXSR33 1110011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DMXSR33  Rd, Rs1, Rs2
```

Purpose:

Do two signed 32-bit cross multiplications from the 32-bit elements of two registers, and each multiplications performs a right shift operation.

Description:

For the "DMXSR33" instruction, multiply the top 32-bit Q31 content of 64-bit chunks in Rs1 with the bottom 32-bit Q31 content of 64-bit chunks in Rs2. At the same time, multiply the bottom 32-bit Q31 content of 64-bit chunks in Rs1 with the top 32-bit Q31 content of 64-bit chunks in Rs2.

The Q63 results are then right-shifted 33-bits and clipped to Q31 values. The Q31 results are then written into Rd.

Operations:

```
Rd.W[0] = (Rs1.W[0] s* Rs2.W[1]) s>> 33

Rd.W[1] = (Rs1.W[1] s* Rs2.W[0]) s>> 33
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dmxsr33(unsigned long long a, unsigned long long b);

int32x2_t __v_dmxsr33(int32x2_t a, int32x2_t b);
```

26.7.19 DREDAS16(Reduced Addition and Reduced Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DREDAS16 0000110	DREDAS16 00010	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DREDAS16  Rd, Rs1
```

Purpose:

Do halfs reduced subtraction and halfs reduced addition from a register. The result is written to Rd.

Description:

For the "DREDAS16" instruction, subtract the top 16-bit Q15 element from the bottom 16-bit Q15 element of the bottom 32-bit Q31 content of 64-bit chunks in Rs1. At the same time, add the the top16-bit Q15 element with the bottom16-bit Q15 element of the top 32-bit Q31 content of 64-bit chunks in Rs1. The two Q15 results are then written into Rd.

Operations:

```
Rd.H[0] = Rs1.H[0] - Rs1.H[1]

Rd.H[1] = Rs1.H[2] + Rs1.H[3]
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long __dredas16(unsigned long long a);

int16x2_t __v_dredas16(int16x4_t a);
```

26.7.20 DREDSA16(Reduced Subtraction and Reduced Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DREDSA16 0000110	DREDSA16 00011	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DREDSA16  Rd, Rs1
```

Purpose:

Do halfs reduced subtraction and halfs reduced addition from a register. The result is written to Rd.

Description:

For the "DREDSA16" instruction, add the top 16-bit Q15 element from the bottom 16-bit Q15 element of the bottom 32-bit Q31 content of 64-bit chunks in Rs1. At the same time, subtract the the top16-bit Q15 element with the bottom16-bit Q15 element of the top 32-bit Q31 content of 64-bit chunks in Rs1. The two Q15 results are then written into Rd.

Operations:

```
Rd.H[0] = Rs1.H[0] + Rs1.H[1]

Rd.H[1] = Rs1.H[2] - Rs1.H[3]
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long __dredsa16(unsigned long long a);

int16x2_t __v_dredsa16(int16x4_t a);
```

26.7.21 DKCLIP64(64-bit Clipped to 16-bit Saturation Value)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKCLIP64 0000110	DKCLIP64 00001	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKCLIP64  Rd, Rs
```

Purpose:

Do 15-bit element arithmetic right shift operations and limit result into 32-bit int,then do saturate operation to 16-bit and clip result to 16-bit Q15.

Description:

For the "DKCLIP64" instruction, shift the input 15 bits to the right and data convert the result to 32-bit int type, after which the input is saturated to limit the data to between 2^{15}-1 and -2^{15}. the result is converted to 16-bits q15 type. The final results are written to Rd.

Operations:

```
const int32_t max = (int32_t)((1U << 15U) - 1U);

const int32_t min = -1 - max ;

int32_t val = (int32_t)(Rs s>> 15);

if (val > max) {

    Rd = max;

} else if (val < min) {

    Rd = min;

} else {

    Rd = (q15_t)val;

}
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
int16_t __dkclip64(long long a);

q15_t __v_dkclip64(int64_t a);
```

26.7.22 DKMDA(Signed Multiply Two Halfs and Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMDA 0110010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMDA  Rd, Rs1, Rs2
```

Purpose:

Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then adds the two 32-bit results together. The addition result may be saturated.

DKMDA: top*top + bottom*bottom (per 32-bit element)

Description:

This instruction multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

The addition result is checked for saturation. If saturation happens, the result is saturated to 2^{31}-1 The final results are written to Rd. The 16-bit contents are treated as signed integers.

Operations:

```
if (Rs1.W[x] != 0x80008000) or (Rs2.W[x] != 0x80008000){

    Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[1]) + (Rs1.W[x].H[0] * Rs2.W[x].H[0]);

} else {

    Rd.W[x] = 0x7fffffff;

    OV = 1;

}

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmda(unsigned long long a, unsigned long long b);

int32x2_t __v_dkmda(int16x4_t a, int16x4_t b);
```

26.7.23 DKMXDA(Signed Crossed Multiply Two Halfs and Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMXDA 0110011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMXDA  Rd, Rs1, Rs2
```

Purpose:

Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then adds the two 32-bit results together. The addition result may be saturated.

DKMXDA: top*bottom + top*bottom (per 32-bit element)

Description:

This instruction multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

The addition result is checked for saturation.If saturation happens, the result is saturated to 2^{31}-1 The final results are written to Rd. The 16-bit contents are treated as signed integers.

Operations:

```
if (Rs1.W[x] != 0x80008000) or (Rs2.W[x] != 0x80008000){

    Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[0]) + (Rs1.W[x].H[0] * Rs2.W[x].H[1]);

} else {

    Rd.W[x] = 0x7fffffff;

    OV = 1;

}

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmxda(unsigned long long a, unsigned long long b);

int32x2_t __v_dkmxda(int16x4_t a, int16x4_t b);
```

26.7.24 DSMDRS(Signed Multiply Two Halfs and Reverse Subtract)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMDRS 0110100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMDRS  Rd, Rs1, Rs2
```

Purpose:

Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then perform a subtraction operation between the two 32-bit results.

DSMDRS: bottom*bottom - top*top (per 32-bit element)

Description:

This instruction multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

The subtraction result is written to the corresponding 32-bit element of Rd. The 16-bit contents of multiplication are treated as signed integers.

Operations:

```
Rd.W[x] = (Rs1.W[x].H[0] * Rs2.W[x].H[0]) - (Rs1.W[x].H[1] * Rs2.W[x].H[1]);

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsmdrs(unsigned long long a, unsigned long long b);

int32x2 __v_dsmdrs(int16x4_t a, int16x4_t b);
```

26.7.25 DSMXDS(Signed Crossed Multiply Two Halfs and Subtract)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMXDS 0110101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMXDS  Rd, Rs1, Rs2
```

Purpose:

Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then perform a subtraction operation

between the two 32-bit results.

DSMDRS: top*bottom - bottom*top (per 32-bit element)

Description:

This instruction multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

The subtraction result is written to the corresponding 32-bit element of Rd. The 16-bit contents of multiplication are treated as signed integers.

Operations:

```
Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.W[x].H[1]);

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsmxds(unsigned long long a, unsigned long long b);

int32x2 __v_dsmxds(int16x4_t a, int16x4_t b);
```

26.7.26 DSMBB32(Signed Multiply Bottom Word & Bottom Word)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMBB32 0110110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMBB32.sra14  Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element of a register with the signed 32-bit element of another register and write the 64-bit result to a third register.

DSMBB32: bottom*bottom

Description:

This instruction multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The 64-bit multiplication result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = (Rs1.W[0] * Rs2.W[0]);

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmbb32(unsigned long long a, unsigned long long b);

int64_t __v_dsmbb32(int32x2_t a, int32x2_t b);
```

26.7.27 DSMBB32.sra14(Signed Multiply Bottom Word & Bottom Word with Right Shift 14)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMBB32.sra14 0110111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMBB32.sra14  Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element of a register with the signed 32-bit element of another register, then right shift 14-bit,finally write the 64-bit result to a third register.

DSMBB32.sra14: bottom*bottom s>> 14

Description:

This instruction multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The 64-bit multiplication result is written to Rd after right shift 14-bit. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = (Rs1.W[0] * Rs2.W[0]) s>> 14;

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmbb32.sra14(unsigned long long a, unsigned long long b);

int64_t __v_dsmbb32.sra14(int32x2_t a, int32x2_t b);
```

26.7.28 DSMBB32.sra32(Signed Multiply Bottom Word & Bottom Word with Right Shift 32)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMBB32.sra32 0111000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMBB32.sra32  Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element of a register with the signed 32-bit element of another register, then right shift 14-bit,finally write the 64-bit result to a third register.

DSMBB32.sra32: bottom*bottom s>> 32

Description:

This instruction multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The 64-bit multiplication result is written to Rd after right shift 14-bit. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = (Rs1.W[0] * Rs2.W[0]) s>> 32;

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmbb32.sra32(unsigned long long a, unsigned long long b);

int64_t __v_dsmbb32.sra32(int32x2_t a, int32x2_t b);
```

26.7.29 DSMBT32(Signed Multiply Bottom Word & Top Word)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMBT32 0111001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMBT32 Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element of a register with the signed 32-bit element of another register and write the 64-bit result to a third register.

DSMBT32: bottom*top

Description:

This instruction multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2. The 64-bit multiplication result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = (Rs1.W[0] * Rs2.W[0]);

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmbt32(unsigned long long a, unsigned long long b);

int64_t __v_dsmbt32(int32x2_t a, int32x2_t b);
```

26.7.30 DSMBT32.sra14(Signed Multiply Bottom Word & Top Word with Right Shift 14)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMBT32.sra14 0111010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMBT32.sra14  Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element of a register with the signed 32-bit element of another register, then right shift 14-bit,finally write the 64-bit result to a third register.

DSMBT32.sra14: bottom*bottom s>> 14

Description:

This instruction multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2. The 64-bit multiplication result is written to Rd after right shift 14-bit. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = (Rs1.W[0] * Rs2.W[0]) s>> 14;

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmbt32.sra14(unsigned long long a, unsigned long long b);

int64_t __v_dsmbt32.sra14(int32x2_t a, int32x2_t b);
```

26.7.31 DSMBT32.sra32(Signed Crossed Multiply Two Halfs and Subtract with Right Shift 32)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMBT32.sra32 0111011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMBT32.sra32  Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element of a register with the signed 32-bit element of another register, then right shift 32-bit,finally write the 64-bit result to a third register.

DSMBT32.sra32: bottom*bottom s>> 32

Description:

This instruction multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The 64-bit multiplication result is written to Rd after right shift 32-bit. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = (Rs1.W[0] * Rs2.W[0]) s>> 32;

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmbt32.sra32(unsigned long long a, unsigned long long b);

int64_t __v_dsmbt32.sra32(int32x2_t a, int32x2_t b);
```

26.7.32 DSMTT32(Signed Multiply Top Word & Top Word)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMTT32 0111100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMTT32  Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element of a register with the signed 32-bit element of another register and write the 64-bit result to a third register.

DSMTT32: top*top

Description:

This instruction multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2. The 64-bit multiplication result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rs1.W[1] * Rs2.W[1];

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmtt32(unsigned long long a, unsigned long long b);

int64_t __v_dsmtt32(int32x2_t a, int32x2_t b);
```

26.7.33 DSMTT32.sra14(Signed Multiply Top Word & Top Word with Right Shift 14-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMTT32.sra14 0111101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMTT32.sra14  Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element of a register with the signed 32-bit element of another register,then right shift 14-bit,finally write the 64-bit result to a third register.

DSMTT32.sra14: top*top s>> 14

Description:

This instruction multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2. The 64-bit multiplication result is written to Rd after right shift 14-bit. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rs1.W[1] * Rs2.W[1] >> 14;

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmtt32.sra14(unsigned long long a, unsigned long long b);

int64_t __v_dsmtt32.sra14(int32x2_t a, int32x2_t b);
```

26.7.34 DSMTT32.sra32(Signed Multiply Top Word & Top Word with Right Shift 32-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMTT32.sra32 0111110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMTT32.sra32  Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element of a register with the signed 32-bit element of another register,then right shift 32-bit,finally write the 64-bit result to a third register.

DSMTT32.sra32: top*top s>> 32

Description:

This instruction multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2. The 64-bit multiplication result is written to Rd after right shift 32-bit. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rs1.W[1] * Rs2.W[1] >> 32;

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmtt32.sra32(unsigned long long a, unsigned long long b);

int64_t __v_dsmtt32.sra32(int32x2_t a, int32x2_t b);
```

26.7.35 DPKBB32(Pack Two 32-bit Data from Both Bottom Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DPKBB32 0111111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DPKBB32  Rd, Rs1, Rs2
```

Purpose:

Pack 32-bit data from 64-bit chunks in two registers.

DPKBB32: bottom.bottom

Description:

This instruction moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0].

Operations:

```
Rd = CONCAT(Rs1.W[0], Rs2.W[0]);
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dpkbb32(unsigned long long a, unsigned long long b);

uint32x2_t __v_dpkbb32(uint32x2_t a, uint32x2_t b);
```

26.7.36 DPKBT32(Pack Two 32-bit Data from Bottom and Top Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DPKBT32 1000000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DPKBT32  Rd, Rs1, Rs2
```

Purpose:

Pack 32-bit data from 64-bit chunks in two registers.

DPKBT32: bottom.top

Description:

This instruction moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0].

Operations:

```
Rd = CONCAT(Rs1.W[0], Rs2.W[1]);
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dpkbt32(unsigned long long a, unsigned long long b);

uint32x2_t __v_dpkbt32(uint32x2_t a, uint32x2_t b);
```

26.7.37 DPKTT32(Pack Two 32-bit Data from Both Top Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DPKTT32 1000001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DPKTT32  Rd, Rs1, Rs2
```

Purpose:

Pack 32-bit data from 64-bit chunks in two registers.

DPKTT32: top.top

Description:

This instruction moves Rs1.W[1] to Rd.W[0] and moves Rs2.W[1] to Rd.W[0].

Operations:

```
Rd = CONCAT(Rs1.W[1], Rs2.W[1]);
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dpktt32(unsigned long long a, unsigned long long b);

uint32x2_t __v_dpktt32(uint32x2_t a, uint32x2_t b);
```

26.7.38 DPKTB32(Pack Two 32-bit Data from Top and Bottom Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DPKTB32 1000010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DPKTB32 Rd, Rs1, Rs2
```

Purpose:

Pack 32-bit data from 64-bit chunks in two registers.

DPKTB32: top.bottom

Description:

This instruction moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0].

Operations:

```
Rd = CONCAT(Rs1.W[1], Rs2.W[0]);
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dpktb32(unsigned long long a, unsigned long long b);

uint32x2_t __v_dpktb32(uint32x2_t a, uint32x2_t b);
```

26.7.39 DPKTB16(Pack Two 16-bit Data from Both Bottom Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DPKTB16 1000011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DPKTB16 Rd, Rs1, Rs2
```

Purpose:

Pack 16-bit data from 32-bit chunks in two registers.

DPKTB16: top.bottom

Description:

This instruction moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0].

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][15:0]);

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dpktb16(unsigned long long a, unsigned long long b);

uint16x4_t __v_dpktb16(uint16x4_t a, uint16x4_t b);
```

26.7.40 DPKBB16(Pack Two 16-bit Data from Both Bottom Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DPKBB16 1000100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DPKBB16 Rd, Rs1, Rs2
```

Purpose:

Pack 16-bit data from 32-bit chunks in two registers.

PKBB16: bottom.bottom

Description:

This instruction moves Rs1.W[x][15:0] to Rd.W[x][31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0].

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][15:0]);

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dpkbb16(unsigned long long a, unsigned long long b);

uint16x4_t __v_dpkbb16(uint16x4_t a, uint16x4_t b);
```

26.7.41 DPKBT16(Pack Two 16-bit Data from Bottom and Top Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DPKBT16 1000101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
PKBT16 Rd, Rs1, Rs2
```

Purpose:

Pack 16-bit data from 32-bit chunks in two registers.

PKBT16: bottom.top

Description:

This instruction moves Rs1.W[x] [15:0] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0].

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][31:16]);
x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dpkbt16(unsigned long long a, unsigned long long b);

uint16x4_t __v_dpkbt16(uint16x4_t a, uint16x4_t b);
```

26.7.42 DPKTT16(Pack Two 16-bit Data from Both Top Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DPKTT16 1000110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DPKTT16 Rd, Rs1, Rs2
```

Purpose:

Pack 16-bit data from 32-bit chunks in two registers.

PKTT16: top.top

Description:

This instruction moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0].

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][31:16]);

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dpktt16(unsigned long long a, unsigned long long b);

uint16x4_t __v_dpktt16(uint16x4_t a, uint16x4_t b);
```

26.7.43 DSRA16(SIMD 16-bit Shift Right Arithmetic)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSRA16 1000111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSRA16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit element arithmetic right shift operations simultaneously. The shift amount is a variable from a GPR.

Description:

The 16-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the low-order 4-bits of the value in the Rs2 register. And the results are written to Rd.

Operations:

```
sa = Rs2[3:0];

if (sa != 0)

{

    Rd.H[x] = SE16(Rs1.H[x][15:sa]);

} else {

    Rd = Rs1;

}

x=3...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsra16(unsigned long long a, unsigned long b);

int16x4_t __v_dsra16(int16x4_t a, unsigned int b);
```

26.7.44 DADD16(16-bit Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DADD16 1001000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DADD16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit integer element additions simultaneously.

Description:

This instruction adds the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2. And the results are written to Rd.

Operations:

```
Rd.H[x] = Rs1.H[x] + Rs2.H[x];

x=3...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dadd16(unsigned long long a, unsigned long long b);

int16x4_t __v_dadd16(int16x4_t a, int16x4_t b);
```

26.7.45 DADD32(32-bit Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DADD32 1001001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DADD32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit integer element additions simultaneously.

Description:

This instruction adds the 32-bit integer elements in Rs1 with the 32-bit integer elements in Rs2, and then writes the 32-bit element results to Rd.

Operations:

```
Rd.W[x] = Rs1.W[x] + Rs2.W[x];

x=1...0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned addition.

Intrinsic functions:

```
unsigned long long __dadd32(unsigned long long a, unsigned long long b);

int32x2_t __v_dadd32(int32x2_t a, int32x2_t b);
```

26.7.46 DSMBB16(Signed Multiply Bottom Half & Bottom Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMBB16 1001010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMBB16 Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 16-bit content of the 32-bit elements of a register with the signed 16-bit content of the 32-bit elements of another register and write the result to a third register.

DSMBB16: W[x].bottom*W[x].bottom

Description:

For the "DSMBB16" instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

The multiplication results are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rd.W[x] = Rs1.W[x].H[0] * Rs2.W[x].H[0];

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsmbb16(unsigned long long a, unsigned long long b);

int32x2_t __v_dsmbb16(int16x4_t a, int16x4_t b);
```

26.7.47 DSMBT16(Signed Multiply Bottom Half & Top Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMBT16 1001011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMBT16 Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 16-bit content of the 32-bit elements of a register with the signed 16-bit content of the 32-bit elements of another register and write the result to a third register.

DSMBT16:W[x].bottom*W[x].top

Description:

For the "DSMBT16" instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

The multiplication results are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rd.W[x] = Rs1.W[x].H[0] * Rs2.W[x].H[1];

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsmbt16(unsigned long long a, unsigned long long b);

int32x2_t __v_dsmbt16(int16x4_t a, int16x4_t b);
```

26.7.48 DSMTT16(Signed Multiply Top Half & Top Half)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMTT16 1001100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMTT16 Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 16-bit content of the 32-bit elements of a register with the signed 16-bit content of the 32-bit elements of another register and write the result to a third register.

DSMTT16: W[x].top * W[x].top

Description:

For the "DSMBB16" instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

The multiplication results are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rd.W[x] = Rs1.W[x].H[1] * Rs2.W[x].H[1];

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsmtt16(unsigned long long a, unsigned long long b);

int32x2_t __v_dsmtt16(int16x4_t a, int16x4_t b);
```

26.7.49 DRCRSA16(16-bit Signed Halving Cross Subtraction & Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DRCRSA16 1001101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DRCRSA16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit signed integer element subtraction and 16-bit signed integer element addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description:

This instruction subtracts the 16-bit signed integer in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer in [15:0] of 32-bit chunks in Rs2, and adds the 16-bit signed integer in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer in [15:0] of 32-bit chunks in Rs1. The element results are first logically right-shifted by 1 bit and then written to [31:16] of 32- bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] - Rs2.W[x][15:0]) s>> 1;

Rd.W[x][15:0] = (Rs1.W[x][15:0] + Rs2.W[x][31:16]) s>> 1;

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __drcrsa16(unsigned long long a, unsigned long long b);

int16x4_t __v_drcrsa16(int16x4_t a, int16x4_t b);
```

26.7.50 DRCRSA32(32-bit Signed Halving Cross Subtraction & Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DRCRSA32 1011110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DRCRSA32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit signed integer element subtraction and 32-bit signed integer element addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. The results are halved to avoid overflow or saturation.

Description:

This instruction subtracts the 32-bit signed integer element in [63:32] of Rs1 with the 32-bit signed integer element in [31:0] of Rs2, and adds the 32-bit signed integer element in [63:32] of Rs2 from the 32-bit signed integer element in [31:0] of Rs1. The element results are first arithmetically right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
Rd.W[1] = (Rs1.W[1] - Rs2.W[0]) s>> 1;

Rd.W[0] = (Rs1.W[0] + Rs2.W[1]) s>> 1;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __drcrsa32(unsigned long long a, unsigned long long b);

int32x2_t __v_drcrsa32(int32x2_t a, int32x2_t b);
```

26.7.51 DRCRAS16(16-bit Signed Halving Cross Addition & Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DRCRAS16 1001110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DRCRAS16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit signed integer element subtraction and 16-bit signed integer element addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description:

This instruction adds the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs1 with the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs2, and subtracts the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs2 from the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs1. The element results are first logically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] + Rs2.W[x][15:0]) s>> 1;

Rd.W[x][15:0] = (Rs1.W[x][15:0] - Rs2.W[x][31:16]) s>> 1;

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __drcras16(unsigned long long a, unsigned long long b);

int16x4_t __v_drcras16(int16x4_t a, int16x4_t b);
```

26.7.52 DRCRAS32(32-bit Signed Cross Addition & Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DRCRAS32 1110010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DRCRAS32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit signed integer element subtraction and 32-bit signed integer element addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. The results are halved to avoid overflow or saturation.

Description:

This instruction add the 32-bit signed integer element in [63:32] of Rs1 with the 32-bit signed integer element in [31:0] of Rs2, and subtract the 32-bit signed integer element in [63:32] of Rs2 from the 32-bit signed integer element in [31:0] of Rs1. The element results are first arithmetically right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
Rd.W[1] = (Rs1.W[1] + Rs2.W[0]) s>> 1;

Rd.W[0] = (Rs1.W[0] - Rs2.W[1]) s>> 1;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __drcrsa32(unsigned long long a, unsigned long long b);

int32x2_t __v_drcrsa32(int32x2_t a, int32x2_t b);
```

26.7.53 DKCRAS16(16-bit Signed Saturating Cross Addition & Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKCRAS16 1010000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKCRAS16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit signed integer element saturating addition and 16-bit signed integer element saturating subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description:

This instruction adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2; at the same time, it subtracts the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1.

If any of the results are beyond the Q15 number range (-2^{15} <= Q15 <= 2^{15}-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for addition and [15:0] of 32-bit chunks in Rd for subtraction.

Operations:

```
res1 = Rs1.W[x][31:16] + Rs2.W[x][15:0];

res2 = Rs1.W[x][15:0]  - Rs2.W[x][31:16];

for (res in [res1, res2]) {

   if (res > (2^15)-1) {

      res = (2^15)-1;

      OV = 1;

   } else if (res < -2^15) {

      res = -2^15;

      OV = 1;

   }
```

(continues on next page)

(continued from previous page)

```
}

Rd.W[x][31:16] = res1;

Rd.W[x][15:0] = res2;

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkcras16(unsigned long long a, unsigned long long b);

int16x4_t __v_dkcras16(int16x4_t a, int16x4_t b);
```

26.7.54 DKCRSA16(16-bit Signed Saturating Cross Subtraction & Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKCRSA16 1001111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKCRSA16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit signed integer element saturating subtraction and 16-bit signed integer element saturating addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description:

This instruction subtracts the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1; at the same time, it adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1.

If any of the results are beyond the Q15 number range (-2^{15} <= Q15 <= $2^{15}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] - Rs2.W[x][15:0];

res2 = Rs1.W[x][15:0]  + Rs2.W[x][31:16];

for (res in [res1, res2]) {

   if (res > (2^15)-1) {

      res = (2^15)-1;

      OV = 1;
```

(continues on next page)

(continued from previous page)

```
   } else if (res < -2^15) {

      res = -2^15;

      OV = 1;

   }

}

Rd.W[x][31:16] = res1;

Rd.W[x][15:0] = res2;

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkcrsa16(unsigned long long a, unsigned long long b);

int16x4_t __v_dkcrsa16(int16x4_t a, int16x4_t b);
```

26.7.55 DRSUB16(16-bit Signed Halving Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DRSUB16 1010001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DRSUB16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit signed integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description:

This instruction subtracts the 16-bit signed integer elements in Rs2 from the 16-bit signed integer elements in Rs1. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] - Rs2.H[x]) s>> 1;

x=3...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __drsub16(unsigned long long a, unsigned long long b);

int16x4_t __v_drsub16(int16x4_t a, int16x4_t b);
```

26.7.56 DSTSA32(32-bit Straight Subtraction & Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSTSA32 1010011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSTSA32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit integer element subtraction and 32-bit integer element addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

Description:

This instruction subtracts the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [63:32] of Rs1, and writes the result to [63:32] of Rd; at the same time, it adds the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [31:0] of Rs2, and writes the result to [31:0] of Rd.

Operations:

```
Rd.W[1] = Rs1.W[1] - Rs2.W[1];

Rd.W[0] = Rs1.W[0] + Rs2.W[0];
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations

Intrinsic functions:

```
unsigned long long __dstsa32(unsigned long long a, unsigned long long b);

uint32x2_t __v_dstsa32(uint32x2_t a, uint32x2_t b);
```

26.7.57 DSTAS32(32-bit Straight Addition & Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSTAS32 1010100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSTAS32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit integer element subtraction and 32-bit integer element addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

Description:

This instruction adds the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [63:32] of Rs1, and writes the result to [63:32] of Rd; at the same time, it subtracts the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [31:0] of Rs2, and writes the result to [31:0] of Rd.

Operations:

```
Rd.W[1] = Rs1.W[1] + Rs2.W[1];

Rd.W[0] = Rs1.W[0] - Rs2.W[0];
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations

Intrinsic functions:

```
unsigned long long __dstas32(unsigned long long a, unsigned long long b);

uint32x2_t __v_dstas32(uint32x2_t a, uint32x2_t b);
```

26.7.58 DKCRSA32(32-bit Signed Saturating Cross Subtraction & Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKCRSA32 1010110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKCRSA32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit signed integer element saturating subtraction and 32-bit signed integer element saturating addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description:

This instruction subtracts the 32-bit integer element in [31:0] of Rs2 from the 32-bit integer element in [63:32] of Rs1; at the same time, it adds the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [63:32] of Rs2. If any of the results are beyond the Q31 number range (-2^{31} <= Q31 <= 2^{31}-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res[1] = Rs1.W[1] - Rs2.W[0];

res[0] = Rs1.W[0] + Rs2.W[1];

if (res[x] > (2^31)-1) {
```

(continues on next page)

(continued from previous page)

```
    res[x] = (2^31)-1;

    OV = 1;

} else if (res < -2^31) {

    res[x] = -2^31;

    OV = 1;

}

Rd.W[1] = res[1];

Rd.W[0] = res[0];

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkcrsa32(unsigned long long a, unsigned long long b);

int32x2_t __v_dkcrsa32(int32x2_t a, int32x2_t b);
```

26.7.59 DKCRAS32(32-bit Signed Saturating Cross Addition & Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKCRAS32 1010101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKCRAS32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit signed integer element saturating subtraction and 32-bit signed integer element saturating addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description:

This instruction adds the 32-bit integer element in [31:0] of Rs2 from the 32-bit integer element in [63:32] of Rs1; at the same time, it subtracts the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [63:32] of Rs2. If any of the results are beyond the Q31 number range (-2^{31} <= Q31 <= 2^{31}-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res[1] = Rs1.W[1] + Rs2.W[0];
```

(continues on next page)

(continued from previous page)

```
res[0] = Rs1.W[0] - Rs2.W[1];

if (res[x] > (2^31)-1) {

    res[x] = (2^31)-1;

    OV = 1;

} else if (res < -2^31) {

    res[x] = -2^31;

    OV = 1;

}

Rd.W[1] = res[1];

Rd.W[0] = res[0];

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkcras32(unsigned long long a, unsigned long long b);

int32x2_t __v_dkcras32(int32x2_t a, int32x2_t b);
```

26.7.60 DCRSA32(32-bit Cross Subtraction & Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DCRSA32 1010111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DCRSA32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit integer element subtraction and 32-bit integer element addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description:

This instruction subtracts the 32-bit integer element in [63:32] of Rs1 with the 32-bit integer element in [31:0] of Rs2, and writes the result to [63:32] of Rd; at the same time, it adds the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [31:0] of Rs1, and writes the result to [31:0] of Rd.

Operations:

```
res[1] = Rs1.W[1] - Rs2.W[0];

res[0] = Rs1.W[0] + Rs2.W[1];
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations.

Intrinsic functions:

```
unsigned long long __dcrsa32(unsigned long long a, unsigned long long b);

int32x2_t __v_dcrsa32(int32x2_t a, int32x2_t b);
```

26.7.61 DCRAS32(32-bit Cross Addition & Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DCRAS32 1011000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DCRAS32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit integer element subtraction and 32-bit integer element addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description:

This instruction subtracts the 32-bit integer element in [63:32] of Rs1 with the 32-bit integer element in [31:0] of Rs2, and writes the result to [63:32] of Rd; at the same time, it adds the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [31:0] of Rs1, and writes the result to [31:0] of Rd.

Operations:

```
res[1] = Rs1.W[1] + Rs2.W[0];

res[0] = Rs1.W[0] - Rs2.W[1];
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations.

Intrinsic functions:

```
unsigned long long __dcras32(unsigned long long a, unsigned long long b);

int32x2_t __v_dcras32(int32x2_t a, int32x2_t b);
```

26.7.62 DKSTSA16(16-bit Signed Saturating Straight Subtraction & Addition)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKSTSA16 1011001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKSTSA16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit signed integer element saturating subtraction and 16-bit signed integer element saturating addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description:

This instruction subtracts the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1; at the same time, it adds the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1.

If any of the results are beyond the Q15 number range (-2^{15} <= Q15 <= 2^{15}-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] - Rs2.W[x][31:16];

res2 = Rs1.W[x][15:0] + Rs2.W[x][15:0];

for (res in [res1, res2]) {

   if (res > (2^15)-1) {

      res = (2^15)-1;

      OV = 1;

   } else if (res < -2^15) {

      res = -2^15;

      OV = 1;

   }

}

Rd.W[x][31:16] = res1;

Rd.W[x][15:0] = res2;

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkstsa16(unsigned long long a, unsigned long long b);

int16x4_t __v_dkstsa16(int16x4_t a, int16x4_t b);
```

26.7.63 DKSTAS16(16-bit Signed Saturating Straight Addition & Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKSTAS16 1011010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKSTAS16 Rd, Rs1, Rs2
```

Purpose:

Do 16-bit signed integer element saturating subtraction and 16-bit signed integer element saturating addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description:

This instruction adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2; at the same time, it subtracts the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1.

If any of the results are beyond the Q15 number range (-2^{15} <= Q15 <= $2^{15}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] + Rs2.W[x][31:16];

res2 = Rs1.W[x][15:0] - Rs2.W[x][15:0];

for (res in [res1, res2]) {

   if (res > (2^15)-1) {

      res = (2^15)-1;

      OV = 1;

   } else if (res < -2^15) {

      res = -2^15;

      OV = 1;

   }

}

Rd.W[x][31:16] = res1;

Rd.W[x][15:0] = res2;
```

(continues on next page)

(continued from previous page)

```
x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkstas16(unsigned long long a, unsigned long long b);

int16x4_t __v_dkstas16(int16x4_t a, int16x4_t b);
```

26.7.64 DSCLIP8(8-bit Signed Saturation and Clip)

Type: SIMD

Format:

31 - 25	24 - 23	22 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSCLIP8 1011011	00	imm3u[2:0]	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSCLIP8 Rd, Rs1, imm3u[2:0]
```

Purpose:

Limit the 8-bit signed integer elements of a register into a signed range simultaneously.

Description:

This instruction limits the 8-bit signed integer elements stored in Rs1 into a signed integer range between -2^{imm3u} and $2^{imm3u}-1$, and writes the limited results to Rd. For example, if imm3u is 3, the 8-bit input values should be saturated between 7 and -8. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.B[x];

if (src > (2^imm3u)-1) {

    src = (2^imm3u)-1;

    OV = 1;

} else if (src < -2^imm3u) {

    src = -2^imm3u;

    OV = 1;

}

Rd.B[x] = src

x=7...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsclip8(unsigned long long a, unsigned long b);

int8x8_t __v_dsclip8(int8x8_t a, unsigned long b);
```

26.7.65 DSCLIP16(16-bit Signed Saturation and Clip)

Type: SIMD

Format:

31 - 25	24	23 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSCLIP16 1011100	0	imm4u[3:0]	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSCLIP16 Rd, Rs1, imm4u[3:0]
```

Purpose:

Limit the 16-bit signed integer elements of a register into a signed range simultaneously.

Description:

This instruction limits the 16-bit signed integer elements stored in Rs1 into a signed integer range between -2^{imm4u} and $2^{imm4u}-1$, and writes the limited results to Rd. For example, if imm4u is 3, the 16-bit input values should be saturated between 7 and -8. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.H[x];

if (src > (2^imm4u)-1) {

    src = (2^imm4u)-1;

    OV = 1;

} else if (src < -2^imm4u) {

    src = -2^imm4u;

    OV = 1;

}

Rd.H[x] = src

x=3...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsclip16(unsigned long long a, unsigned long b);

int16x4_t __v_dsclip16(int16x4_t a, unsigned long b);
```

26.7.66 DSCLIP32 (32-bit Signed Saturation and Clip)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSCLIP32 1011101	imm5u[4:0]	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSCLIP32 Rd, Rs1, imm5u[4:0]
```

Purpose:

Limit the 32-bit signed integer elements of a register into a signed range simultaneously.

Description:

This instruction limits the 32-bit signed integer elements stored in Rs1 into a signed integer range between -2^{imm5u} and $2^{imm5u}-1$, and writes the limited results to Rd. For example, if imm5u is 3, the 32-bit input values should be saturated between 7 and -8. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.W[x];

if (src > (2^imm5u)-1) {

    src = (2^imm5u)-1;

    OV = 1;

} else if (src < -2^imm5u) {

    src = -2^imm5u;

     OV = 1;

}

Rd.W[x] = src

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsclip32(unsigned long long a, unsigned long b);

int32x2_t __v_dsclip32(int32x2_t a, unsigned long b);
```

26.7.67 DRSUB32 (32-bit Signed Halving Subtraction)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DRSUB32 1010010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DRSUB32 Rd, Rs1, Rs2
```

Purpose:

Do 32-bit signed integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description:

This instruction subtracts the 32-bit signed integer elements in Rs2 from the 32-bit signed integer elements in Rs1. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] - Rs2.W[x]) s>> 1;

x=1...0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __drsub32(unsigned long long a, unsigned long long b);

int32x2_t __v_drsub32(int32x2_t a, int32x2_t b);
```

26.7.68 DPACK32 (SIMD Pack Two 32-bit Data To 64-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DPACK32 1100110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DPACK32 Rd, Rs1, Rs2
```

Purpose:

Pack two 32-bit datas which from two registers into a 64-bit data.

Description:

This instruction moves 32-bit Rs1 to Rd.W[1] and moves 32-bit Rs2 to Rd.W[0].

Operations:

```
Rd = CONCAT(Rs1.W , Rs2.W);
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dpack32(signed long a, signed long b);

int32x2_t __v_dpack32(int32_t a, int32_t b);
```

26.7.69 DSUNPKD810,DSUNPKD820,DSUNPKD830,DSUNPKD831,DSUNPKD832

26.7.69.1 DSUNPKD810 (Signed Unpacking Bytes 1 & 0)

26.7.69.2 DSUNPKD820 (Signed Unpacking Bytes 2 & 0)

26.7.69.3 DSUNPKD830 (Signed Unpacking Bytes 3 & 0)

26.7.69.4 DSUNPKD831 (Signed Unpacking Bytes 3 & 1)

26.7.69.5 DSUNPKD832 (Signed Unpacking Bytes 3 & 2)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSUNPKD8 **xy** 0000110	xxxxx	Rs1	000	Rd	custom-3 1111011

xy	xxxxx
10	00100
20	00101
30	00110
31	00111
32	01000

Syntax:

```
DSUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose:

Unpack byte x and byte y of 32-bit chunks in a register into two 16-bit signed halfwordsof 32-bit chunks in a register.

Description:

For the "DSUNPKD8xy" instruction, it unpacks byte x and byte y of 32-bit chunks in Rs1 into two 16-bit signed halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = SE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = SE16(Rs1.W[m].B[y])
//DSUNPKD810, x=1,y=0
```

(continues on next page)

(continued from previous page)

```
//DSUNPKD820, x=2,y=0
//DSUNPKD830, x=3,y=0
//DSUNPKD831, x=3,y=1
//DSUNPKD832, x=3,y=2
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
DSUNPKD810:

unsigned long long __dsunpkd810(unsigned long long a);

int16x4_t __v_dsunpkd810(int8x8_t a);

DSUNPKD820:

unsigned long long __dsunpkd820(unsigned long long a);

int16x4_t __v_dsunpkd820(int8x8_t a);

DSUNPKD830:

unsigned long long __dsunpkd830(unsigned long long a);

int16x4_t __v_dsunpkd830(int8x8_t a);

DSUNPKD831:

unsigned long long __dsunpkd831(unsigned long long a);

int16x4_t __v_dsunpkd831(int8x8_t a);

DSUNPKD832:

unsigned long long __dsunpkd832(unsigned long long a);

int16x4_t __v_dsunpkd832(int8x8_t a);
```

26.7.70 DZUNPKD810,DZUNPKD820,DZUNPKD830,DZUNPKD831,DZUNPKD832

26.7.70.1 DZUNPKD810 (Signed Unpacking Bytes 1 & 0)

26.7.70.2 DZUNPKD820 (Signed Unpacking Bytes 2 & 0)

26.7.70.3 DZUNPKD830 (Signed Unpacking Bytes 3 & 0)

26.7.70.4 DZUNPKD831 (Signed Unpacking Bytes 3 & 1)

26.7.70.5 DZUNPKD832 (Signed Unpacking Bytes 3 & 2)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DZUNPKD8 **xy** 0000110	xxxxx	Rs1	000	Rd	custom-3 1111011

xy	xxxxx
10	01001
20	01010
30	01011
31	01100
32	01101

Syntax:

```
DZUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose:

Unpack byte x and byte y of 32-bit chunks in a register into two 16-bit unsigned halfwordsof 32-bit chunks in a register.

Description:

For the "DZUNPKD8xy" instruction, it unpacks byte x and byte y of 32-bit chunks in Rs1 into two 16-bit unsigned halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = SE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = SE16(Rs1.W[m].B[y])
//DZUNPKD810, x=1,y=0
//DZUNPKD820, x=2,y=0
//DZUNPKD830, x=3,y=0
//DZUNPKD831, x=3,y=1
//DZUNPKD832, x=3,y=2
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
DZUNPKD810:

unsigned long long __dzunpkd810(unsigned long long a);

uint16x4_t __v_dzunpkd810(uint8x8_t a);

DZUNPKD820:

unsigned long long __dzunpkd820(unsigned long long a);

uint16x4_t __v_dzunpkd820(uint8x8_t a);

DZUNPKD830:

unsigned long long __dzunpkd830(unsigned long long a);

uint16x4_t __v_dzunpkd830(uint8x8_t a);

DZUNPKD831:
```

(continues on next page)

(continued from previous page)

```
unsigned long long __dzunpkd831(unsigned long long a);

uint16x4_t __v_dzunpkd831(uint8x8_t a);

DZUNPKD832:

unsigned long long __dzunpkd832(unsigned long long a);

uint16x4_t __v_dzunpkd832(uint8x8_t a);
```

26.8 Appendix D: Nuclei N3 SIMD DSP Additional Instruction

26.8.1 DKMMAC (64-bit MSW 32x32 Signed Multiply and Saturating Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMMAC 0001010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMMAC Rd, Rs1, Rs2
```

Purpose:

Do MSW 32x32 element signed multiplications and saturating addition simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   res = sat.q31(dop + (aop s* bop)[63:32]);

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmmac(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dkmmac(int32x2_t c, int32x2_t a, int32x2_t b);
```

26.8.2 DKMMAC.u (64-bit MSW 32x32 Unsigned Multiply and Saturating Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMMAC.u 0001011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMMAC.u Rd, Rs1, Rs2
```

Purpose:

Do MSW 32x32 element unsigned multiplications and saturating addition simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   res = sat.q31(dop + RUND(aop u* bop)[63:32]);

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmmac_u(unsigned long long c, unsigned long long a, unsigned long␣
↪long b);

int32x2_t __v_dkmmac_u(int32x2_t c, int32x2_t a, int32x2_t b);
```

26.8.3 DKMMSB (64-bit MSW 32x32 Signed Multiply and Saturating Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMMSB 0001100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMMSB Rd, Rs1, Rs2
```

Purpose:

Do MSW 32x32 element signed multiplications and saturating subtraction simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   res = sat.q31(dop - (aop s* bop)[63:32]);

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmmsb(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dkmmsb(int32x2_t c, int32x2_t a, int32x2_t b);
```

26.8.4 DKMMSB.u (64-bit MSW 32x32 Unsigned Multiply and Saturating Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMMSB.u 0001101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMMSB.u Rd, Rs1, Rs2
```

Purpose:

Do MSW 32x32 element unsigned multiplications and saturating subtraction simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   res = sat.q31(dop - (aop u* bop)[63:32]);

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmmsb_u(unsigned long long c, unsigned long long a, unsigned long␣
↪long b);

int32x2_t __v_dkmmsb_u(int32x2_t c, int32x2_t a, int32x2_t b);
```

26.8.5 DKMADA (Two 16x16 with 32-bit Signed Double Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMADA 0001110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMADA Rd, Rs1, Rs2
```

Purpose:

Do two 16x16 with 32-bit signed double addition simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top
```

(continues on next page)

(continued from previous page)

```
op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   mul1 = aop.H[1] s* bop.H[1];

   mul2 = aop.H[0] s* bop.H[0];

   res = sat.q31(dop + mul1 + mul2);

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmada(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dkmada(int32x2_t c, int16x4_t a, int16x4_t b);
```

26.8.6 DKMAXDA (Two Cross 16x16 with 32-bit Signed Double Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMAXDA 0001111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMAXDA Rd, Rs1, Rs2
```

Purpose:

Do two cross 16x16 with 32-bit signed double addition simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   mul1 = aop.H[1] s* bop.H[0];

```

(continues on next page)

(continued from previous page)

```
   mul2 = aop.H[0] s* bop.H[1];

   res = sat.q31(dop + mul1 + mul2);

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmaxda(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dkmaxda(int32x2_t c, int16x4_t a, int16x4_t b);
```

26.8.7 DKMADS (Two 16x16 with 32-bit Signed Add and Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMADS 0010000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMADS Rd, Rs1, Rs2
```

Purpose:

Do two 16x16 with 32-bit signed addition and subtraction simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   mul1 = aop.H[1] s* bop.H[1];

   mul2 = aop.H[0] s* bop.H[0];

   res = sat.q31(dop + mul1 - mul2);

}

Rd = concat(rest, resb);
```

(continues on next page)

(continued from previous page)

```
x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmads(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dkmads(int32x2_t c, int16x4_t a, int16x4_t b);
```

26.8.8 DKMADRS (Two 16x16 with 32-bit Signed Add and Reversed Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMADRS 0010001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMADRS Rd, Rs1, Rs2
```

Purpose:

Do two 16x16 with 32-bit signed addition and revered subtraction simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   mul1 = aop.H[1] s* bop.H[1];

   mul2 = aop.H[0] s* bop.H[0];

   res = sat.q31(dop - mul1 + mul2);

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmadrs(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dkmadrs(int32x2_t c, int16x4_t a, int16x4_t b);
```

26.8.9 DKMAXDS (Two Cross 16x16 with 32-bit Signed Add and Sub)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
DKMAXDS 0010010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMAXDS Rd, Rs1, Rs2
```

Purpose:

Do two cross 16x16 with 32-bit signed addition and subtraction simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   mul1 = aop.H[1] s* bop.H[0];

   mul2 = aop.H[0] s* bop.H[1];

   res = sat.q31(dop + mul1 - mul2);

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmaxds(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dkmaxds(int32x2_t c, int16x4_t a, int16x4_t b);
```

26.8.10 DKMSDA (Two 16x16 with 32-bit Signed Double Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMSDA 0010011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMSDA Rd, Rs1, Rs2
```

Purpose:

Do two 16x16 with 32-bit signed double subtraction simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   mul1 = aop.H[1] s* bop.H[1];

   mul2 = aop.H[0] s* bop.H[0];

   res = sat.q31(dop - mul1 - mul2);

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmsda(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dkmsda(int32x2_t c, int16x4_t a, int16x4_t b);
```

26.8.11 DKMSXDA (Two Cross 16x16 with 32-bit Signed Double Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMSXDA 0010100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMSXDA Rd, Rs1, Rs2
```

Purpose:

Do two cross 16x16 with 32-bit signed double subtraction simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   mul1 = aop.H[1] s* bop.H[0];

   mul2 = aop.H[1] s* bop.H[0];

   res = sat.q31(dop - mul1 - mul2);

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkmsxda(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dkmsxda(int32x2_t c, int16x4_t a, int16x4_t b);
```

26.8.12 DSMAQA (Four Signed 8x8 with 32-bit Signed Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMAQA 0010101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMAQA Rd, Rs1, Rs2
```

Purpose:

Do four signed 8x8 with 32-bit signed addition simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   m0 = aop.B[0] s* bop.B[0];

   m1 = aop.B[1] s* bop.B[1];

   m2 = aop.B[2] s* bop.B[2];

   m3 = aop.B[3] s* bop.B[3];

   res = dop + m0 + m1 + m2 + m3;

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsmaqa(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dsmaqa(int32x2_t c, int8x8_t a, int8x8_t b);
```

26.8.13 DSMAQA.SU (Four Signed 8 x Unsigned 8 with 32-bit Signed Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMAQA.SU 0010110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMAQA.SU Rd, Rs1, Rs2
```

Purpose:

Do four signed 8 x unsigned 8 with 32-bit signed addition simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   m0 = aop.B[0] su* bop.B[0];

   m1 = aop.B[1] su* bop.B[1];

   m2 = aop.B[2] su* bop.B[2];

   m3 = aop.B[3] su* bop.B[3];

   res = dop + m0 + m1 + m2 + m3;

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dsmaqa_su(unsigned long long c, unsigned long long a, unsigned long␣
↪long b);

int32x2_t __v_dsmaqa_su(int32x2_t c, int8x8_t a, int8x8_t b);
```

26.8.14 DUMAQA (Four Unsigned 8x8 with 32-bit Unsigned Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMAQA 0010111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DUMAQA Rd, Rs1, Rs2
```

Purpose:

Do four unsigned 8x8 with 32-bit unsigned addition simultaneously. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; op3t = Rd.W[x+1] // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; op3b = Rd.W[x] // bottom


for ((aop,bop,dop,res) in [(op1t,op2t,op3t,rest), (op1b,op2b,op3b,resb)]) {

   m0 = aop.B[0] u* bop.B[0];

   m1 = aop.B[1] u* bop.B[1];

   m2 = aop.B[2] u* bop.B[2];

   m3 = aop.B[3] u* bop.B[3];

   res = dop + m0 + m1 + m2 + m3;

}

Rd = concat(rest, resb);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dumaqa(unsigned long long c, unsigned long long a, unsigned long long␣
↪b);

int32x2_t __v_dumaqa(int32x2_t c, int8x8_t a, int8x8_t b);
```

26.8.15 DKMDA32 (Two Signed 32x32 with 64-bit Saturation Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMDA32 0011000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMDA32 Rd, Rs1, Rs2
```

Purpose:

Do two signed 32x32 add the signed multiplication results with Q63 saturation. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t0 = op1b s* op2b;

t1 = op1t s* op2t;

Rd = sat.q63(t0 + t1);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmda32(unsigned long long a, unsigned long long b);

int64_t __v_dkmda32(int32x2_t a, int32x2_t b);
```

26.8.16 DKMXDA32 (Two Cross Signed 32x32 with 64-bit Saturation Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMXDA32 0011001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMXDA32 Rd, Rs1, Rs2
```

Purpose:

Do two cross signed 32x32 and add the signed multiplication results with Q63 saturation. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t01 = op1b s* op2t;

t10 = op1t s* op2b;

Rd = sat.q63(t01 + t10);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmxda32(unsigned long long a, unsigned long long b);

int64_t __v_dkmxda32(int32x2_t a, int32x2_t b);
```

26.8.17 DKMADA32 (Two Signed 32x32 with 64-bit Saturation Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMADA32 0011010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMADA32 Rd, Rs1, Rs2
```

Purpose:

Do two signed 32x32 and add the signed multiplication results and a third register with Q63 saturation. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t0 = op1b s* op2b;

t1 = op1t s* op2t;
```

(continues on next page)

(continued from previous page)

```
Rd = sat.q63(Rd + t0 + t1);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmada32(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dkmada32(int64_t c, int32x2_t a, int32x2_t b);
```

26.8.18 DKMAXDA32 (Two Cross Signed 32x32 with 64-bit Saturation Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMAXDA32 0011011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMAXDA32 Rd, Rs1, Rs2
```

Purpose:

Do two cross signed 32x32 and add the signed multiplication results and a third register with Q63 saturation. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t01 = op1b s* op2t;

t10 = op1t s* op2b;

Rd = sat.q63(Rd + t01 + t10);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmaxda32(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dkmaxda32(int64_t c, int32x2_t a, int32x2_t b);
```

26.8.19 DKMADS32 (Two Signed 32x32 with 64-bit Saturation Add and Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMADS32 0011100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMADS32 Rd, Rs1, Rs2
```

Purpose:

Do two signed 32x32 and add the top signed multiplication results and subtraction bottom signed multiplication results and add a third register with Q63 saturation. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t0 = op1b s* op2b;

t1 = op1t s* op2t;

Rd = sat.q63(Rd - t0 + t1);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmads32(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dkmads32(int64_t c, int32x2_t a, int32x2_t b);
```

26.8.20 DKMADRS32 (Two Signed 32x32 with 64-bit Saturation Revered Add and Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMADRS32 0011101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMADRS32 Rd, Rs1, Rs2
```

Purpose:

Do two signed 32x32 and add the signed multiplication results and a third register with Q63 saturation. The results are written into Rd.Do two signed 32x32 and subtraction the top signed multiplication results and add bottom signed multiplication results and add a third register with Q63 saturation. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t0 = op1b s* op2b;

t1 = op1t s* op2t;

Rd = sat.q63(Rd + t0 - t1);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmadrs32(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dkmadrs32(int64_t c, int32x2_t a, int32x2_t b);
```

26.8.21 DKMAXDS32 (Two Cross Signed 32x32 with 64-bit Saturation Add and Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMAXDS32 0011110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMAXDS32 Rd, Rs1, Rs2
```

Purpose:

Do two signed 32x32 and add the top signed multiplication results and subtraction bottom signed multiplication results and add a third register with Q63 saturation. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t01 = op1b s* op2t;
```

(continues on next page)

(continued from previous page)

```
t10 = op1t s* op2b;

Rd = sat.q63(Rd - t01 + t10);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmaxds32(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dkmaxds32(int64_t c, int32x2_t a, int32x2_t b);
```

26.8.22 DKMSDA32 (Two Signed 32x32 with 64-bit Saturation Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMSDA32 0011111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMSDA32 Rd, Rs1, Rs2
```

Purpose:

Do two signed 32x32 and subtraction the top signed multiplication results and subtraction bottom signed multiplication results and add a third register with Q63 saturation. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t0 = op1b s* op2b;

t1 = op1t s* op2t;

Rd = sat.q63(Rd - t0 - t1);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmsda32(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dkmsda32(int64_t c, int32x2_t a, int32x2_t b);
```

26.8.23 DKMSXDA32 (Two Cross Signed 32x32 with 64-bit Saturation Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMSXDA32 0100000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMSXDA32 Rd, Rs1, Rs2
```

Purpose:

Do two cross signed 32x32 and subtraction the top signed multiplication results and subtraction bottom signed multiplication results and add a third register with Q63 saturation. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t01 = op1b s* op2t;

t10 = op1t s* op2b;

Rd = sat.q63(Rd - t01 - t10);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmsxda32(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dkmsxda32(int64_t c, int32x2_t a, int32x2_t b);
```

26.8.24 DSMDS32 (Two Signed 32x32 with 64-bit Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMDS32 0100001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMDS32 Rd, Rs1, Rs2
```

Purpose:

Do two signed 32x32 and add the top signed multiplication results and subtraction bottom signed multiplication. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t0 = op1b s* op2b;

t1 = op1t s* op2t;

Rd = t1 - t0;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmds32(unsigned long long a, unsigned long long b);

int64_t __v_dsmds32 (int32x2_t a, int32x2_t b);
```

26.8.25 DSMDRS32 (Two Signed 32x32 with 64-bit Revered Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMDRS32 0100010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMDRS32 Rd, Rs1, Rs2
```

Purpose:

Do two signed 32x32 and subtraction the top signed multiplication results and add bottom signed multiplication. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t0 = op1b s* op2b;

t1 = op1t s* op2t;

Rd = t0 - t1;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmdrs32(unsigned long long a, unsigned long long b);

int64_t __v_ dsmdrs32 (int32x2_t a, int32x2_t b);
```

26.8.26 DSMXDS32 (Two Cross Signed 32x32 with 64-bit Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMXDS32 0100011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMXDS32 Rd, Rs1, Rs2
```

Purpose:

Do two cross signed 32x32 and add the top signed multiplication results and subtraction bottom signed multiplication. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


t01 = op1b s* op2t;

t10 = op1t s* op2b;
```

(continues on next page)

(continued from previous page)

```
Rd = t10 - t01;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmxds32(unsigned long long a, unsigned long long b);

int64_t __v_dsmxds32(int32x2_t a, int32x2_t b);
```

26.8.27 DSMALDA (Four Signed 16x16 with 64-bit Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMALDA 0100100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMALDA Rd, Rs1, Rs2
```

Purpose:

Do four signed 16x16 and add signed multiplication results and a third register. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


m0 = op1b.H[0] s* op2b.H[0];

m1 = op1b.H[1] s* op2b.H[1];

m2 = op1t.H[0] s* op2t.H[0];

m3 = op1t.H[1] s* op2t.H[1];


Rd = Rd + m0 + m1 + m2 + m3;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmalda(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dsmalda (int64_t c, int16x4_t a, int16x4_t b);
```

26.8.28 DSMALXDA (Four Cross Signed 16x16 with 64-bit Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMALXDA 0100101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMALXDA Rd, Rs1, Rs2
```

Purpose:

Do four cross signed 16x16 and add signed multiplication results and a third register. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


m0 = op1b.H[0] s* op2b.H[1];

m1 = op1b.H[1] s* op2b.H[0];

m2 = op1t.H[0] s* op2t.H[1];

m3 = op1t.H[1] s* op2t.H[0];


Rd = Rd + m0 + m1 + m2 + m3;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmalxda(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dsmalxda (int64_t c, int16x4_t a, int16x4_t b);
```

26.8.29 DSMALDS (Four Signed 16x16 with 64-bit Add and Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMALDS 0100110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMALDS Rd, Rs1, Rs2
```

Purpose:

Do four signed 16x16 and add and subtraction signed multiplication results and a third register. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


m0 = op1b.H[1] s* op2b.H[1];

m1 = op1b.H[0] s* op2b.H[0];

m2 = op1t.H[1] s* op2t.H[1];

m3 = op1t.H[0] s* op2t.H[0];


Rd = Rd + m0 - m1 + m2 - m3;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmalds(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dsmalds (int64_t c, int16x4_t a, int16x4_t b);
```

26.8.30 DSMALDRS (Four Signed 16x16 with 64-bit Add and Revered Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMALDRS 0100111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMALDRS Rd, Rs1, Rs2
```

Purpose:

Do two signed 16x16 and add and revered subtraction signed multiplication results and a third register. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


m0 = op1b.H[0] s* op2b.H[0];

m1 = op1b.H[1] s* op2b.H[1];

m2 = op1t.H[0] s* op2t.H[0];

m3 = op1t.H[1] s* op2t.H[1];


Rd = Rd + m0 - m1 + m2 - m3;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmaldrs(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dsmaldrs (int64_t c, int16x4_t a, int16x4_t b);
```

26.8.31 DSMALXDS (Four Cross Signed 16x16 with 64-bit Add and Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMALXDS 0101000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMALXDS Rd, Rs1, Rs2
```

Purpose:

Do four cross signed 16x16 and add and subtraction signed multiplication results and a third register. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


m0 = op1b.H[1] s* op2b.H[0];

m1 = op1b.H[0] s* op2b.H[1];

m2 = op1t.H[1] s* op2t.H[0];

m3 = op1t.H[0] s* op2t.H[1];


Rd = Rd + m0 - m1 + m2 - m3;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmalxds(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dsmalxds (int64_t c, int16x4_t a, int16x4_t b);
```

26.8.32 DSMSLDA (Four Signed 16x16 with 64-bit Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMSLDA 0101001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMSLDA Rd, Rs1, Rs2
```

Purpose:

Do four signed 16x16 and subtraction signed multiplication results and add a third register. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


m0 = op1b.H[0] s* op2b.H[0];

m1 = op1b.H[1] s* op2b.H[1];

m2 = op1t.H[0] s* op2t.H[0];

m3 = op1t.H[1] s* op2t.H[1];


Rd = Rd - m0 - m1 - m2 - m3;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmslda(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dsmslda (int64_t c, int16x4_t a, int16x4_t b);
```

26.8.33 DSMSLXDA (Four Cross Signed 16x16 with 64-bit Sub)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMSLXDA 0101010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMSLXDA Rd, Rs1, Rs2
```

Purpose:

Do four signed 16x16 and subtraction signed multiplication results and add a third register. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


m0 = op1b.H[0] s* op2b.H[1];

m1 = op1b.H[1] s* op2b.H[0];

m2 = op1t.H[0] s* op2t.H[1];

m3 = op1t.H[1] s* op2t.H[0];


Rd = Rd - m0 - m1 - m2 - m3;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmslxda(long long c, unsigned long long a, unsigned long long b);

int64_t __v_dsmslxda (int64_t c, int16x4_t a, int16x4_t b);
```

26.8.34 DDSMAQA (Eight Signed 8x8 with 64-bit Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DDSMAQA 0101011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DDSMAQA Rd, Rs1, Rs2
```

Purpose:

Do eight signed 8x8 and add signed multiplication results and a third register. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


m0 = op1b.B[0] s* op2b.B[0];

m1 = op1b.B[1] s* op2b.B[1];

m2 = op1b.B[2] s* op2b.B[2];

m3 = op1b.B[3] s* op2b.B[3];

m4 = op1t.B[0] s* op2t.B[0];

m5 = op1t.B[1] s* op2t.B[1];

m6 = op1t.B[2] s* op2t.B[2];

m7 = op1t.B[3] s* op2t.B[3];


s0 = m0 + m1 + m2 + m3;

s1 = m4 + m5 + m6 + m7;

Rd = Rd + s0 + s1;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __ddsmaqa(long long c, unsigned long long a, unsigned long long b);

int64_t __v_ddsmaqa (int64_t c, int8x8_t a, int8x8_t b);
```

26.8.35 DDSMAQA.SU (Eight Signed 8 x Unsigned 8 with 64-bit Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 13	11 - 7	6 - 0
DDSMAQA.SU 0101100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DDSMAQA.SU Rd, Rs1, Rs2
```

Purpose:

Do eight signed 8 x unsigned 8 and add signed multiplication results and a third register. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


m0 = op1b.B[0] su* op2b.B[0];

m1 = op1b.B[1] su* op2b.B[1];

m2 = op1b.B[2] su* op2b.B[2];

m3 = op1b.B[3] su* op2b.B[3];

m4 = op1t.B[0] su* op2t.B[0];

m5 = op1t.B[1] su* op2t.B[1];

m6 = op1t.B[2] su* op2t.B[2];

m7 = op1t.B[3] su* op2t.B[3];


s0 = m0 + m1 + m2 + m3;

s1 = m4 + m5 + m6 + m7;

Rd = Rd + s0 + s1;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __ddsmaqa_su(long long c, unsigned long long a, unsigned long long b);

int64_t __v_ddsmaqa_su (int64_t c, int8x8_t a, int8x8_t b);
```

26.8.36 DDUMAQA (Eight Unsigned 8x8 with 64-bit Unsigned Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DDSMAQA 0101101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DDUMAQA Rd, Rs1, Rs2
```

Purpose:

Do eight unsigned 8x8 and add unsigned multiplication results and a third register. The results are written into Rd.

Description:

Operations:

```
op1t = Rs1.W[x+1]; op2t = Rs2.W[x+1]; // top

op1b = Rs1.W[x]; op2b = Rs2.W[x]; // bottom


m0 = op1b.B[0] u* op2b.B[0];

m1 = op1b.B[1] u* op2b.B[1];

m2 = op1b.B[2] u* op2b.B[2];

m3 = op1b.B[3] u* op2b.B[3];

m4 = op1t.B[0] u* op2t.B[0];

m5 = op1t.B[1] u* op2t.B[1];

m6 = op1t.B[2] u* op2t.B[2];

m7 = op1t.B[3] u* op2t.B[3];


s0 = m0 + m1 + m2 + m3;

s1 = m4 + m5 + m6 + m7;

Rd = Rd + s0 + s1;

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __ddumaqa(long long c, unsigned long long a, unsigned long long b);

int64_t __v_ddumaqa (int64_t c, int8x8_t a, int8x8_t b);
```

26.8.37 DSMA32.u (64-bit SIMD 32-bit Signed Multiply Addition With Rounding and Clip)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMA32.u 1101000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMA32.u  Rd, Rs1, Rs2
```

Purpose:

Do two signed 32x32 and add signed multiplication results with Rounding, then right shift 32-bit and clip q63 to q31. The result is written to Rd.

Description:

For the "DSMA32.u" instruction, multiply the top 32-bit Q31 content of 64-bit chunks in Rs1 with the top 32-bit Q31 content of 64-bit chunks in Rs2. At the same time, multiply the bottom 32-bit Q31 content of 64-bit chunks in Rs1 with the bottom 32-bit Q31 content of 64-bit chunks in Rs2.

At the same time, accumulate the results and perform additional rounding operations, and then move the data to the right by 32-bit, and clip the 64-bit data into 32-bit.The result is written to Rd.

Operations:

```
Rd = (q31_t)((Rs1.W[x] s* Rs2.W[x] + Rs1.W[x + 1] s* Rs2.W[x + 1] + 0x80000000LL) s>> 32);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long __dsma32_u(unsigned long long a, unsigned long long b);

int32_t __v_dsma32_u(int32x2_t a, int32x2_t b);
```

26.8.38 DSMXS32.u (64-bit SIMD 32-bit Signed Multiply Cross Subtraction With Rounding and Clip)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMXS32.u 1101001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMXS32.u  Rd, Rs1, Rs2
```

Purpose:

Do two cross signed 32x32 and sub signed multiplication results with Rounding, then right shift 32-bit and clip q63 to q31. The result is written to Rd.

Description:

For the "DSMXS32.u" instruction, multiply the top 32-bit Q31 content of 64-bit chunks in Rs1 with the bottom 32-bit Q31 content of 64-bit chunks in Rs2. At the same time, multiply the bottom 32-bit Q31 content of 64-bit chunks in Rs1 with the top 32-bit Q31 content of 64-bit chunks in Rs2.

At the same time, subtract the results and perform additional rounding operations, and then move the data to the right by 32-bit, and clip the 64-bit data into 32-bit.The result is written to Rd.

Operations:

```
Rd = (q31_1)((Rs1.W[x + 1] s* Rs2.W[x] - Rs1.W[x] s* Rs2.W[x + 1] + 0x80000000LL) s>> 32);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long __dsmxs32_u(unsigned long long a, unsigned long long b);

int32_t __v_dsmxs32_u(int32x2_t a, int32x2_t b);
```

26.8.39 DSMXA32.u (64-bit SIMD 32-bit Signed Cross Multiply Addition with Rounding and Clip)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMXA32.u 1101010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMXA32.u  Rd, Rs1, Rs2
```

Purpose:

Do two cross signed 32x32 and add signed multiplication results with Rounding, then right shift 32-bit and clip q63 to q31. The result is written to Rd.

Description:

For the "DSMXA32.u" instruction,multiply the top 32-bit Q31 content of 64-bit chunks in Rs1 with the bottom 32-bit Q31 content of 64-bit chunks in Rs2. At the same time, multiply the bottom 32-bit Q31 content of 64-bit chunks in Rs1 with the top 32-bit Q31 content of 64-bit chunks in Rs2.

At the same time, accumulate the results and perform additional rounding operations, and then move the data to the right by 32-bit, and clip the 64-bit data into 32-bit.The result is written to Rd.

Operations:

```
Rd = (q31_t)((Rs1.W[x + 1] s* Rs2.W[x] + Rs1.W[x] s* Rs2.W[x + 1] + 0x80000000LL) s>> 32);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long __dsmxa32_u(unsigned long long a, unsigned long long b);

int32_t __v_dsmxa32_u(int32x2_t a, int32x2_t b);
```

26.8.40 DSMS32.u (64-bit SIMD 32-bit Signed Multiply Subtraction with Rounding and Clip)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMS32.u 1101011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMS32.u  Rd, Rs1, Rs2
```

Purpose:

Do two signed 32x32 and sub signed multiplication results with Rounding, then right shift 32-bit and clip q63 to q31. The result is written to Rd.

Description:

For the "DSMS32.u" instruction, multiply the bottom 32-bit Q31 content of 64-bit chunks in Rs1 with the bottom 32-bit Q31 content of 64-bit chunks in Rs2. At the same time, multiply the top 32-bit Q31 content of 64-bit chunks in Rs1 with the top 32-bit Q31 content of 64-bit chunks in Rs2.

At the same time, subtract the results and perform additional rounding operations, and then move the data to the right by 32-bit, and clip the 64-bit data into 32-bit.The result is written to Rd.

Operations:

```
Rd = (q31_t)((Rs1.W[x] s* Rs2.W[x] - Rs1.W[x + 1] s* Rs2.W[x + 1] + 0x80000000LL) s>> 32);

x=0
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long __dsms32_u(unsigned long long a, unsigned long long b);

int32_t __v_dsms32_u(int32x2_t a, int32x2_t b);
```

26.8.41 DSMADA16 (Signed Multiply Two Halfs and Two Adds 32-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMADA16 1100010	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMADA16 Rd, Rs1, Rs2
```

Purpose:

Do two signed 16-bit multiplications of two 32-bit registers; and then adds the 32-bit results and the 32-bit value of an even/odd pair of registers together.

DSMADA16: rt pair+ top*top + bottom*bottom

Description:

This instruction multiplies the per 16-bit content of the 32-bit elements of Rs1 with the corresponding 16-bit content of the 32-bit elements of Rs2. The result is added to the 32-bit value of an even/odd pair of registers specified by Rd(4,1). The 32-bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 32-bit value of the register-pair are treated as signed integers.

Operations:

```
// DSMADA16

Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]);

Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]);

Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[0]);

Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[1]);

Rd.W = Rd.W + SE32(Mres0[0][31:0]) + SE32(Mres1[0][31:0]) + SE32(Mres0[1][31:0]) +␣
↪SE32(Mres1[1][31:0]);
```

Exceptions: None

Privilege level: All

Note: The result is a 32-bit number of bits, using only one of the pair registers.

Intrinsic functions:

```
long long __dsmada16(long long t, unsigned long long a, unsigned long long b);

int64_t __v_dsmada16(int64_t t, uint16x4_t a, uint16x4_t b);
```

26.8.42 DSMAXDA16 (Signed Crossed Multiply Two Halfs and Two Adds 32-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMAXDA16 1100011	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMAXDA16 Rd, Rs1, Rs2
```

Purpose:

Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then adds the two 32-bit results and the 32-bit value of an even/odd pair of registers together.

DSMAXDA: rt pair+ top*bottom + bottom*top (all 32-bit elements)

Description:

This instruction multiplies the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 with unlimited precision. The result is added to the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit addition result is clipped to 32-bit result. (The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers.)

Operations:

```
// SMALXDA16

Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[1]);

Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]);

Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[1]);

Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[0]);

Rd.W = Rd.W + SE32(Mres0[0][31:0]) + SE32(Mres1[0][31:0]) + SE32(Mres0[1][31:0]) +␣
↪SE32(Mres1[1][31:0]);
```

Exceptions: None

Privilege level: All

Note: The result is a 32-bit number of bits, using only one of the pair registers.

Intrinsic functions:

```
long __dsmaxda16(long long t, unsigned long long a, unsigned long long b);

int32_t __v_dsmaxda16(long long t, uint16x4_t a, uint16x4_t b);
```

26.8.43 DKSMS32.u (Two Signed Multiply Shift-clip and Saturation with Rounding)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKSMS32.u 1100100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKSMS32.u Rd, Rs1, Rs2
```

Purpose:

Computes saturated multiplication of two pairs of q31 type with shifted rounding.

Description: Compute the multiplication of Rs1 and Rs2 of type q31_t, intercept [47:16] for the resulting 64-bit product to get the 32-bit number, then add 1 to it to do rounding, and finally saturate the result after rounding.

Operations:

```
Mres[x][63:0] = Rs1.W[x] s* Rs2.W[x];

Round[x][32:0] = Mres[x][47:15] + 1;

Rd.W[x] = sat.31(Rd.W[x] + Round[x][32:1]);

x=1...0
```

Exceptions: None

Privilege level: All

Note: Although the data type of Rs1, Rs2 and Rd is q31, the actual data range may only be q23 or even q15, just to avoid introducing the concept of 24-bit, so the CPU can treat it as 32-bit data.

Intrinsic functions:

```
unsigned long long __dksms32_u(unsigned long long c, unsigned long long a, unsigned long␣
↪long b);

uint32x2_t __v_dksms32_u(uint32x2_t c, uint32x2_t a, uint32x2_t b);
```

26.8.44 DMADA32 (Two Cross Signed 32x32 with 64-bit Add and Clip to 32-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DMADA32 1100101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DMADA32 Rd, Rs1, Rs2
```

Purpose:

Do two cross signed 32x32 and add the signed multiplication results to q63, then clip the q63 result to q31 , the final results are written into Rd.

Description:

For the "DMADA32" instruction, it multiplies the top 32-bit element in Rs1 with the bottom 32-bit element in Rs2 and then adds the result to the result of multiplying the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2, then clip the q63 result to q31.

Operations:

```
res = (q31_t)((((q63_t) Rd.w[0] << 32) + (q63_t)Rs1.w[0] s*  Rs2.w[1] + (q63_t)Rs1.w[1] s* ␣
↪Rs2.w[0]) s>> 32);

rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long __dmada32(long long c, unsigned long long a, unsigned long long b);

int32_t __v_dmada32(int64_t c, int32x2_t a, int32x2_t b);
```

26.8.45 DSMALBB (Signed Multiply Bottom Halfs & Add 64-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMALBB 1101100	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMALBB Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 16-bit content of the 32-bit elements of a register with the 16-bit content of the corresponding 32-bit elements of another register and add the results with a 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair (RV32) or the register (RV64).

DSMALBB: rt pair + bottom*bottom (all 32-bit elements)

Description:

For the "DSMALBB" instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

The multiplication results are added with the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
Mres[0][31:0] = Rs1.W[0].H[0] * Rs2.W[0].H[0];

Mres[1][31:0] = Rs1.W[1].H[0] * Rs2.W[1].H[0];

Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmalbb(long long t, unsigned long long a, unsigned long long b);

int64_t __v_dsmalbb(int64_t t, int16x4_t a, int16x4_t b);
```

26.8.46 DSMALBT (Signed Multiply Bottom Half & Top Half & Add 64-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMALBT 1101101	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMALBT Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 16-bit content of the 32-bit elements of a register with the 16-bit content of the corresponding 32-bit elements of another register and add the results with a 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair (RV32) or the register (RV64).

DSMALBT rt pair + bottom*top (all 32-bit elements)

Description:

For the "DSMALBT" instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

The multiplication results are added with the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
Mres[0][31:0] = Rs1.W[0].H[0] * Rs2.W[0].H[1];

Mres[1][31:0] = Rs1.W[1].H[0] * Rs2.W[1].H[1];

Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmalbt(long long t, unsigned long long a, unsigned long long b);

int64_t __v_dsmalbt(int64_t t, int16x4_t a, int16x4_t b);
```

26.8.47 DSMALTT (Signed Multiply Top Half & Add 64-bit)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DSMALTT 1101110	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DSMALTT Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 16-bit content of the 32-bit elements of a register with the 16-bit content of the corresponding 32-bit elements of another register and add the results with a 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair (RV32) or the register (RV64).

DSMALTT rt pair + top*top (all 32-bit elements)

Description:

For the "DSMALTT" instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

The multiplication results are added with the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
Mres[0][31:0] = Rs1.W[0].H[1] * Rs2.W[0].H[1];

Mres[1][31:0] = Rs1.W[1].H[1] * Rs2.W[1].H[1];

Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dsmaltt(long long t, unsigned long long a, unsigned long long b);

int64_t __v_dsmaltt(int64_t t, int16x4_t a, int16x4_t b);
```

26.8.48 DKMABB32 (Saturating Signed Multiply Bottom Words & Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMABB32 1101111	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMABB32 Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element in a register with the 32-bit element in another register and add the result to the content of 64-bit data in the third register. The addition result may besaturated and is written to the third register.

DKMABB32: rd + bottom*bottom

Description:

For the "DKMABB32" instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2.

The multiplication result is added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^{63} <= Q63 <= 2^{63}-1), it is saturated to the range and the OV bit is set to 1. The result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[0] * Rs2.W[0]);

if (res > (2^63)-1) {

    res = (2^63)-1;

    OV = 1;

} else if (res < -2^63) {

    res = -2^63;

    OV = 1;

}

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmabb32(long long t, unsigned long long a, unsigned long long b);

int64_t __v_dkmabb32(int64_t t, int32x2_t a, int32x2_t b);
```

26.8.49 DKMABT32 (Saturating Signed Multiply Bottom & Top Words & Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMABT32 1110000	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMABT32 Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element in a register with the 32-bit element in another register and add the result to the content of 64-bit data in the third register. The addition result may be saturated and is written to the third register.

DKMABT32: rd + bottom*top

Description:

For the "DKMABT32" instruction, it multiplies the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2.

The multiplication result is added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^{63} <= Q63 <= 2^{63}-1), it is saturated to the range and the OV bit is set to 1. The result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[0] * Rs2.W[1]);

if (res > (2^63)-1) {

    res = (2^63)-1;

    OV = 1;

} else if (res < -2^63) {

    res = -2^63;

    OV = 1;

}

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmabt32(long long t, unsigned long long a, unsigned long long b);

int64_t __v_dkmabt32(int64_t t, int32x2_t a, int32x2_t b);
```

26.8.50 DKMATT32 (Saturating Signed Multiply Top Words & Add)

Type: SIMD

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
DKMATT32 1110001	Rs2	Rs1	000	Rd	custom-3 1111011

Syntax:

```
DKMATT32 Rd, Rs1, Rs2
```

Purpose:

Multiply the signed 32-bit element in a register with the 32-bit element in another register and add the result to the content of 64-bit data in the third register. The addition result may be saturated and is written to the third register.

DKMATT32: rd + top*top

Description:

For the "DKMATT32" instruction, it multiplies the top 32-bit element in Rs1 with the top 32-bit element in Rs2.

The multiplication result is added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^{63} <= Q63 <= 2^{63}-1), it is saturated to the range and the OV bit is set to 1. The result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[t] * Rs2.W[1]);

if (res > (2^63)-1) {

    res = (2^63)-1;

    OV = 1;

} else if (res < -2^63) {

    res = -2^63;

    OV = 1;

}

Rd = res;
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
long long __dkmatt32(long long t, unsigned long long a, unsigned long long b);

int64_t __v_dkmatt32(int64_t t, int32x2_t a, int32x2_t b);
```

27 Bit Manipulation Introduction

Nuclei processor core can optionally support the Bit Manipulation (B Extension) instructions. This document will not detail it, please refer to RISC-V official document <riscv-spec-20240411.pdf> for the details.

28 Scalar Entropy Source Introduction

Nuclei processor core can optionally support the Scalar Entropy Source (K Extension) instructions. This document will not detail it, please refer to RISC-V official document <riscv-spec-20240411.pdf> for the details.

29 Vector Introduction

Nuclei processor core can optionally support the Vector (V Extension) instructions. This document will not detail it, please refer to <riscv-spec-20240411.pdf> and related Nuclei processor core's Databook for the details.

30 User Extended Instructions Introduction

30.1 Revision History

Rev.	Revision Date	Revised Content
1.0.0	2019/8/20	1.Initial release.
1.1.0	2019/12/20	1.Change Multi-Cycle Response Error to raise an exception (6.2).
1.2.0	2021/6/30	1.To support 64-bit wide NICE operands without needing the DSP configuration.
1.3.0	2022/6/22	1.NICE dose not support custom-3 instructions.
1.4.0	2023/01/16	1.NICE support to access FPU register as operands. 2.Default using the highest 3 bits of *funct7* to do encoding.
1.5.0	2023/09/27	1.User can define the NICE instructions's decoding.
1.6.0	2023/11/30	1.NICE interface data-width is same with LSU data-width.
1.7.0	2024/01/12	1.N900 support Floating/PAIR/MAC NICE. 2.NI900 support VNICE.

30.2 NICE Introduction

Nuclei all Series Cores support configurable NICE (Nuclei Instruction Co-unit Extension) to support extensive customization and specialization. NICE allows customers to create user-defined instructions, enabling the integrations of custom hardware co-units that improve domain-specific performance while reducing power consumption. For example, artificial intelligence, could take the task of parameter training and model matching as an extension of the RISC-V kernel, which can enhance the performance of the entire system.

Co-unit connected by the NICE interface protocol (Hereinafter referred to as NICE-Core) is an independent module outside the Master Nuclei Core, so the NICE-Core can be turned off for reducing power while it is idle.

Nuclei Instruction Co-unit Extension supports the following features:

- Can be turned on/off through *mstatus* register.
- Support one-cycle transfer and multi-cycle transfer.
- Support 64-bit operation for one-cycle transfer when architecture is RV32 (also called NICE_PAIR, please refer to configuration option in CFG_NICE_64BITS)
- Support NICE_MAC/NICE_FPU instructions (Currently 300 and 900 Series support this).
- Support flexible instructions as user can refine the decoding (Currently 300 and 900 Series support this).
- Support Vector NICE instructions (Currently 900 Series support this).
- Support blocking mode and non-blocking mode for multi-cycle transfer.
- Support memory access with ICB protocol.

- Response error and memory access error observable and controllable.

30.3 NICE Instruction Format

NICE is a standard extension which means it should not conflict with any other RISC-V standard extensions. NICE instructions are also part of RISC-V base instruction set.

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD-FP	*Custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*Custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	⩾80b

Fig. 30.1: RISC-V base opcode map, inst[1:0]=11

In the table, major opcodes marked as *custom-0*, *custom-1*, *custom-2*, and *custom-3* are standard extensions and are recommended for custom instruction-set extensions within the base 32-bit instruction format. Principally, customers can use these four custom instruction groups for NICE extensions. *NICE instruction format* (page 283) shows the detail of NICE instruction format.

Note: Custom-2 and Custom-3 instructions will be used as XLCZ instructions if CPU support XLCZ.

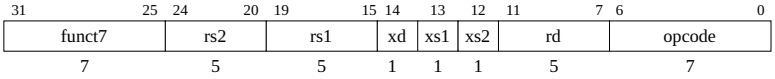


Fig. 30.2: NICE instruction format

For the decoding the NICE instruction's format, it supports two strategies: one is Default Decoding, the other is User Redefined Decoding.

30.3.1 Default NICE Instruction Decoding

In *NICE instruction format* (page 283), once *xd* is set, the NICE instruction will write the result to destination register whose number is encoded in *rd* filed when the instruction retires, otherwise, no write-back operation will go on. If *xs1* or *xs2* is set, the NICE instruction will read the corresponding source register whose number is encoded in *rs1* or *rs2* field, otherwise, no source register will be used.

The 7 bits of *funct7*, can encode instructions for each customer instruction group. Now we use the upper 3 bits to encoding MAC/FPU/Pair type instructions as followed:

Setting bit 31th of NICE instruction (the bit 7 of *funct7*), it means this instruction is MAC type, it will use *rd* as *rs3*, while 0 means it is not MAC type.

Setting bit 30th of NICE instruction (the bit 6 of *funct7*) when NICE has configured with NICE_FPU, it means this instruction is FPU type instruction, all registers encoding target to FPU's register; while 0 means it is Integer type instructions. If the NICE is not configured with NICE_FPU, this bit is open for user.

Setting bit 29th of NICE instruction (the bit 5 of *funct7*) when NICE has configure NICE_64BITS, it means this instruction is Pair type, it will use 64-bit contents of an even/odd pair of registers specified by rs1(4,1), 64-bit contents of an even/odd pair of registers specified by rs2(4,1) and 64-bit contents of an even/odd pair of registers specified by Rd(4,1). *xd*, *xs1,* and *xs2* still control whether they are valid or not.

Other 4 bits of *funct7* can be used as user demands. Besides, NICE could get more instructions when *rs1*, *rs2* or *rd* filed is not used.

Note:

- If the core has been configured DSP Module and DSP Module has been configured with Nuclei extended instructions, please DO NOT use custom-3 opcode space, customer can refer <Nuclei_RISC-V_ISA_Spec.pdf> for detail or contact Nuclei Support.
- If the core has been configured Xlcz extension instructions, please DO NOT use custom-2 & custom-3 opcode space, customer can refer <Nuclei_RISC-V_ISA_Spec.pdf> for detail or contact Nuclei Support.
- FPU type NICE instruction is an option, please refer the related Databook to know it is supported or not or contact Nuclei Support, and FPU NICE instruction dose not support One-Cycle Response.
- Pair type NICE instruction in RV32 core is an option, please refer the related Databook to know it is supported or not or contact Nuclei Support.
- For N300 series core, the three type (MAC/FPU/Pair) can be mixed, but except both of FPU & Pair are set. When Core has single FPU, and NICE support NICE_FPU, it can only do single floating compute in NICE too, when user sets both FPU/Pair bits, then the hardware does not check Pair bit and Pair bit is open to user.
- For N600/N900 series core, it is almost same with N300, but N600/N900 can't support MAC & PAIR integer NICE type as they only has 4 integer GPR read ports.
- For 900 series core with Vector and VNICE feature, the default decoding is not working, it should use user redefined decoding mechanism.

30.3.2 User Redefine NICE Instruction Decoding

If user want to add integer and floating instructions and need all space of *xd*, *xs1*, *xs2* and *funct7*, also wants to encoding the *rs1*, *rs2* and *rd* more flexibly, just like *rs1* is targeting to FPU register and *rs2* is targeting to Integer GPR register and needs pair independently. So we exposing the decoding module to customer , there is a file exu_nice_decode.v which is in design/core can be edited by customer. The interface signals of this module is list as following code:

```
module cpu_exu_nice_decode(
    input  [32-1:0] i_instr,
    output  dec_nice_rs1_en,
    output  dec_nice_rs1_fpu,
    output  dec_nice_rs1_pair,

    output  dec_nice_rs2_en,
    output  dec_nice_rs2_fpu,
    output  dec_nice_rs2_pair,

    output  dec_nice_rd_mac,
    output  dec_nice_rd_en,
    output  dec_nice_rd_fpu,
    output  dec_nice_rd_pair,

    output dec_nice_ilgl,
    output nice_need_fpu
);
```

As the output signals have some combinations we do not support, please refer following tables:

Table 30.2: Combinations Not Support

Instruction Type	Combinations
Single-Cycle	rd_pair 1 rd_fpu 1 rd_mac 1

continues on next page

Table 30.2 – continued from previous page

Instruction Type	Combinations
Single-Cycle	rs1_en 1 rs1_pair 1 rs1_fpu 1
Single-Cycle	rs2_en 1 rs2_pair 1 rs2_fpu 1
Single-Cycle	rd_en 1 rd_fpu 1
Multi-Cycle	rd_pair 1 rd_fpu 1 rd_mac 1
Multi-Cycle	rs1_en 1 rs1_pair 1 rs1_fpu 1
Multi-Cycle	rs2_en 1 rs2_pair 1 rs2_fpu 1
Multi-Cycle	rd_en 1 rd_pair 1

Note:

- If user does not change this file/module, then it is the default encoding.
- When user edit this file/module, please refer the above table carefully or the behavior is undefined.
- If user does not configure NICE_PAIR_MAC in 300 Core RTL's configurations (600 & 900 does not support this) , then rd_mac, rs1_pair, rs2_pair can't output to 1.
- If user does not configure NICE_64BITS in 300/600/900 Core RTL's configurations , then rs1_pair, rs2_pair and rd_pair can't output to 1.
- If user does not configure NICE_FPU in 300/600/900 Core RTL's configurations , then rs1_fpu, rs2_fpu and rd_fpu can't output to 1.
- If user configures NICE-MAC in 600/900 Core RTL's configurations, rs3 is through nice_req_rs1_1; While in 300, and rs3 is through nice_req_rs3.
- If user configures NICE-PAIR in 600/900 Core RTL's configurations, and it can't support 1 cycle pair write back; while in 300, it supports.
- If user configures NICE_FPU & Double FPU in 300/600/900 Core RTL's configurations, it is always PAIR type, rs1_1 & rs1 is floating source1, rs2_1 & rs2 is floating source2, and NICE should write back rs3_1 & rs3 to represent floating rd.

If user want to add vector custom instructions, the interface signals of this module is list as following code:

```
module cpu_exu_nice_decode(
    input  [32-1:0] i_instr,

    output dec_vnice_op1_en,
    output dec_vnice_op1_scalar_reg_fpu,
    output dec_vnice_op1_scalar_reg_int,
    output dec_vnice_op1_vector_reg,

    output dec_vnice_op2_en,

    output dec_vnice_rd_mac,
    output dec_vnice_rd_en,
```

(continues on next page)

(continued from previous page)

```
    output dec_vnice_rd_scalar_reg_fpu,
    output dec_vnice_rd_scalar_reg_int,
    output dec_vnice_rd_vector_reg,

    output dec_vnice_vm,
    output dec_vnice_ilgl,
    output dec_vnice_op,

    output  dec_nice_rs1_en,
    output  dec_nice_rs1_fpu,
    output  dec_nice_rs1_pair,

    output  dec_nice_rs2_en,
    output  dec_nice_rs2_fpu,
    output  dec_nice_rs2_pair,

    output  dec_nice_rd_mac,
    output  dec_nice_rd_en,
    output  dec_nice_rd_fpu,
    output  dec_nice_rd_pair,

    output dec_nice_ilgl,
    output nice_need_fpu
);
```

Table 30.3: VNICE_NICE_Decode_signals

Name	Direction	Width	Description
i_instr	Input	32	32 bits instruction code.
dec_vnice_op1_en	Output	1	Indicate this VNICE instruction need operand 1 (op1) or not.
dec_vnice_op1_scalar_reg_fpu	Output	1	The VNICE instruction's op1 is from FPU.
dec_vnice_op1_scalar_reg_int	Output	1	The VNICE instruction's op1 is from Integer GPR.
dec_vnice_op1_vector_reg	Output	1	The VNICE instruction's op1 is from Vector.
dec_vnice_op2_en	Output	1	Indicate this VNICE instruction need operand 2 (op2) or not.
dec_vnice_op2_scalar_reg_fpu	Output	1	The VNICE instruction's op2 is from FPU.
dec_vnice_op2_scalar_reg_int	Output	1	The VNICE instruction's op2 is from Integer GPR.
dec_vnice_op2_vector_reg	Output	1	The VNICE instruction's op2 is from Vector.
dec_vnice_rd_en	Output	1	Indicate this VNICE instruction need rd or not.
dec_vnice_rd_scalar_reg_fpu	Output	1	The VNICE instruction's rd is from FPU.
dec_vnice_rd_scalar_reg_int	Output	1	The VNICE instruction's rd is from Integer GPR.
dec_vnice_rd_vector_reg	Output	1	The VNICE instruction's rd is from Vector.
dec_vnice_ilgl	Output	1	Indicate this is an illegal VNICE instruction or not.
dec_vnice_op	Output	1	Indicate this is a valid VNICE instruction or not.
dec_vnice_vm	Output	1	Indicate this is a valid mask VNICE instruction and need v0 as mask.

30.4 NICE and VNICE Interface Descriptions

This chapter describes the NICE and VNICE interface signals. It contains the following sections:

- *NICE global signals*
- *NICE request channel signals*
- *NICE one-cycle response channel signals*
- *NICE multi-cycle response channel signals*
- *NICE memory request channel signals*
- *NICE memory response channel signals*
- *VNICE signals*

30.4.1 NICE Global Signals

Table 30.4: NICE_global_signals

Name	Direction	Width	Description
nice_clk	Output	1	Clock for NICE-Core
nice_rst_n	Output	1	Reset for NICE-Core

30.4.2 NICE Request Channel Signals

Table 30.5: NICE_request_channel_signals

Name	Direction	Width	Description
nice_req_valid	Output	1	This signal indicates Master Core sends a nice request. It should keep HIGH until nice_req_ready is set.
nice_req_ready	Input	1	This signal indicates NICE-Core can receive a nice request.
nice_req_instr	Output	32	The entire NICE instruction encoding from Master Core. It should keep stable until nice_req_ready is set.
nice_req_rs1	Output	32/64	The value of source register 1. It should keep stable until nice_req_ready is set.
nice_req_rs2	Output	32/64	The value of source register 2. It should keep stable until nice_req_ready is set.
nice_req_rs3	Output	32/64	The value of source register 3 for MAC Type. It should keep stable until nice_req_ready is set.
nice_req_rs1_1	Output	32	The value of odd register in pair registers rs1(4,1), the high 32-bit in 64-bit data. Note: This signal exists only when CFG_NICE_64BITS is configured
nice_req_rs2_1	Output	32	The value of odd register in pair registers rs2(4,1), the high 32-bit in 64-bit data. Note: This signal exists only when CFG_NICE_64BITS is configured
nice_req_rs3_1	Output	32	The value of odd register in pair registers rs3(4,1) for MAC Type, the high 32-bit in 64-bit data. Note: This signal exists only when CFG_NICE_64BITS is configured.
nice_req_mmode	Output	1	This signal indicates the privilege mode of Master Core: 1: machine mode 0: non-machine mode It should keep stable until nice_req_ready is set.

continues on next page

Table 30.5 – continued from previous page

Name	Direction	Width	Description
nice_req_smode	Output	1	This signal indicates the privilege mode of Master Core: 1: Supervisor mode 0: one-supervisor mode It should keep stable until nice_req_ready is set.

30.4.3 NICE One-Cycle Response Channel Signals

Table 30.6: NICE_one-cycle_response_channel_signals

Name	Direction	Width	Description
nice_rsp_1cyc_type	Input	1	This signal indicates current NICE instruction is a one-cycle instruction and Master Core can get the result in one cycle.
nice_rsp_1cyc_dat	Input	32/64	The result from NICE-Core in one-cycle response
nice_rsp_1cyc_dat_1	Input	32	The value of odd register in pair registers rd(4,1), the high 32-bit in 64-bit data. Note: This signal exists only when CFG_NICE_64BITS is configured
nice_rsp_1cyc_err	Input	1	This signal indicates current one-cycle response has an error and Master Core will take an illegal instruction exception when detecting this signal HIGH. Only support in 200 & 300.

30.4.4 NICE Multi-Cycle Response Channel Signals

Table 30.7: NICE_multi-cycle_response_channel_signals

Name	Direction	Width	Description
nice_rsp_multicyc_valid	Input	1	This signal indicates NICE-Core sends a multi-cycle response. It should keep HIGH until nice_rsp_multicyc_ready is set.
nice_rsp_multicyc_ready	Output	1	This signal indicates Master Core can receive multi-cycle response.
nice_rsp_multicyc_dat	Input	32/64	The result from NICE-Core in multi-cycle response. It should keep stable until nice_rsp_multicyc_ready is set.
nice_rsp_multicyc_err	Input	1	This signal indicates current multi-cycle has an error and Master Core won't write the result to register file.

30.4.5 NICE Memory Request Channel Signals

Table 30.8: NICE_memory_request_channel_signals

Name	Direction	Width	Description
nice_icb_cmd_valid	Input	1	This signal indicates NICE-Core sends a memory access request to Master Core. It should keep HIGH until nice_icb_cmd_ready is set. Memory access request should be sent during a multi-cycle transfer.
nice_icb_cmd_ready	Output	1	This signal indicates Master Core can receive memory access request.
nice_icb_cmd_addr	Input	32/64	The address of memory access request. It should keep stable until nice_icb_cmd_ready is set.

continues on next page

Table 30.8 – continued from previous page

Name	Direction	Width	Description
nice_icb_cmd_read	Input	1	Write or Read of memory access request: 0:Write 1:Read It should keep stable until nice_icb_cmd_ready is set.
nice_icb_cmd_wdata	Input	IM	The write data of memory write request. It should keep stable until nice_icb_cmd_ready is set, data-width is same of Core's LSU.
nice_icb_cmd_size	Input	2	The size of memory access request: 2'b00: byte 2'b01: half-word 2'b10: word 2'b11: reserved It should keep stable until nice_icb_cmd_ready is set. Note: NICE does not support misaligned access.
nice_icb_cmd_mmode	Input	1	The privilege mode is M-mode of memory access request. It should keep stable until nice_icb_cmd_ready is set.
nice_icb_cmd_smode	Input	1	The privilege mode is S-mode of memory access request. It should keep stable until nice_icb_cmd_ready is set.
nice_mem_holdup	Input	1	This signal helps NICE-Core occupy LSU pipe of Master Core for stalling next load and store instruction. This signal should be set one cycle after NICE-Core receives multi-cycle NICE instruction which includes memory operation and cleared after all memory accesses are done.

30.4.6 NICE Memory Response Channel Signals

Table 30.9: NICE_memory_response_channel_signals

Name	Direction	Width	Description
nice_icb_rsp_valid	Output	1	This signal indicates Master Core sends a memory access response to NICE-Core. It should keep HIGH until nice_icb_rsp_ready is set.
nice_icb_rsp_ready	Input	1	This signal indicates NICE-Core can receive memory access response.
nice_icb_rsp_rdata	Output	IM	The read data of memory access. It should keep stable until nice_icb_rsp_ready is set, data-width is same of Core's LSU.
nice_icb_rsp_err	Output	1	This signal indicates an error is detected during memory access of Master Core.

30.4.7 VNICE Channel Signals

Table 30.10: VNICE_channel_signals

Name	Direction	Width	Description
vnice_req_valid	Output	1	This signal indicates Master Core sends a vnice request. It should keep HIGH until vnice_req_ready is set.
vnice_req_ready	Input	1	This signal indicates VNICE-Core can receive a vnice request.
vnice_req_instr	Output	32	The entire VNICE instruction encoding from Master Core. It should keep stable until vnice_req_ready is set.

continues on next page

Table 30.10 – continued from previous page

Name	Direction	Width	Description
vnice_beat	Output	2	Indicate the vnice transaction's begin or end. When lmul > 1, 01 means first lmul transaction; 00 means middle ; 10 means last. When lmul <= 1, 11 means only 1 transactions.
vnice_element_active	Output	vlen/8	Indicate the elements of vector rs1/rs2/rs3 element is active or not. One bit means one element, so the valid bit num of this signal is vlen/sew, the higher bits beyond the valid bits are 0.
vnice_csr_vlmul	Output	3	Indicate the lmul of Vector vlmul csr.
vnice_csr_vsew	Output	3	Indicate the sew of Vector vsew csr.
vnice_rounding_mode	Output	2	Indicate the integer rounding mode of Vector vxrm csr.
vnice_fpu_rounding_mode	Output	2	Indicate the floating rounding mode of FPU frm csr.
vnice_vs1	Output	vlen	The value of source vector register 1.
vnice_vs2	Output	vlen	The value of source vector register 2.
vnice_vd	Output	vlen	The value of source vector register 3 if it is vector MAC type.
vnice_wbck_valid	Input	1	The vnice_core's valid signal.
vnice_wbck_ready	Output	1	The ready output rsp signal to vnice_core.
vnice_wbck_wdata	Input	vlen	The vnice_core's compute result to write back.
vnice_wbck_vxsat	Input	1	Indicate the vnice_core's fix-point computing has saturation flag or not.
vnice_wbck_fflag	Input	1	Indicate the vnice_core's floating computing has exception flag or not.

30.5 NICE and VNICE Transfer

NICE instructions can be sent to NICE-Core (including VNICE-Core)) only when mstatus.xs is NOT Zero, otherwise, an illegal instruction exception will be raised.

Before instruction sent to NICE-Core through the NICE/VNICE interface, it is decoded by the Master Core and marked as a NICE/VNICE instruction, at the same time rs1 and rs2 registers are read for the NICE/VNICE interface if needed.

While NICE/VNICE instruction has a dependency on previous unfinished instruction including common instruction or another NICE/VNICE instruction, the pipeline would be stalled until the dependency is eliminate With this mechanism, NICE/VNICE instruction behaves just like a common instruction from the Master Core side.

NICE request channel confirms a transfer by *nice_req_valid* and *nice_req_ready* handshaking. *nice_req_valid* and other request information should keep stable until *nice_req_ready* signal is HIGH.

VNICE request channel confirms a transfer by *vnice_valid* and *vnice_ready* handshaking. *vnice_valid* and other request information should keep stable until *vnice_ready* signal is HIGH.

The NICE response might return through the one-cycle channel or multi-cycle channel, which depends on the implementation of the NICE-Core.

The VNICE response might return through the one-cycle channel or multi-cycle channel, it uses *vnice_wbck_valid* and *vnice_wbck_ready* handshaking.

30.5.1 One-Cycle Response for NICE

If the NICE-Core can execute the instruction in one cycle, it sends the response to Master Core through one-cycle response channel, with or without data. In this way, *nice_rsp_1cyc_type* and *nice_rsp_1cyc_rdat* should keep for one cycle.

One-Cycle NICE Response with Data (page 291) shows a one-cycle response with data.

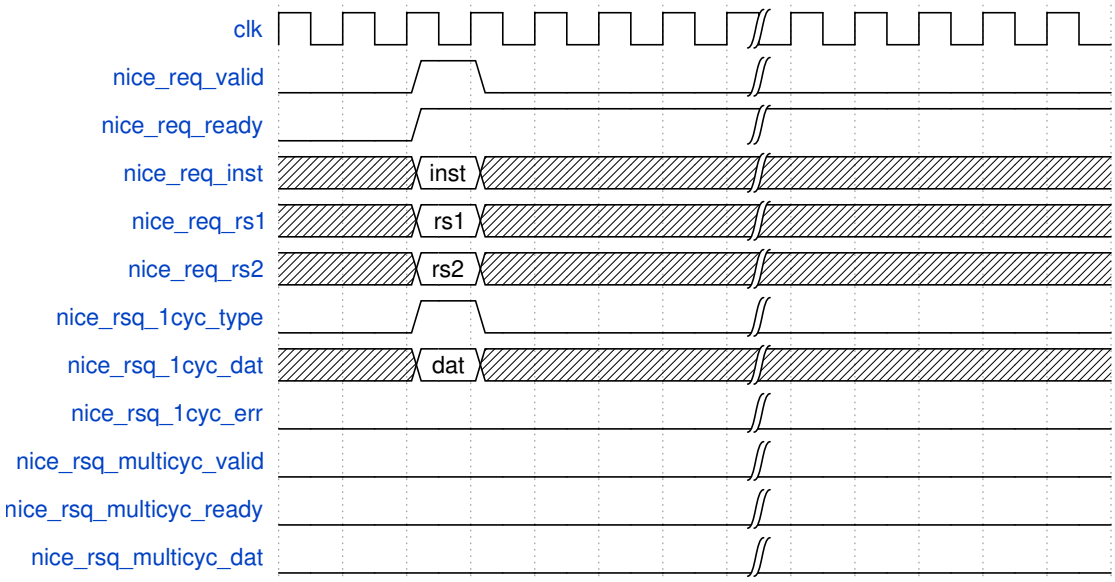


Fig. 30.3: One-Cycle NICE Response with Data

30.5.2 Multi-Cycle Response for NICE

If NICE-Core needs more cycles to get the result (for example, huge computation or memory access. Please refer to *NICE and VNICE Memory Access* (page 293) for memory access operation), it sends the response to Master Core through the multi-cycle response channel.

There are two operation modes for NICE multi-cycle instruction: **blocking mode** and **non-blocking mode**. Blocking mode will clear *nice_req_ready* for stalling new NICE request until the current NICE transfer is done. Non-blocking mode, however, can receive a new NICE request no matter current NICE transaction is finished or not.

30.5.2.1 Multi-Cycle Blocking Mode

NICE multi-cycle blocking mode transfer (page 292) shows a multi-cycle blocking mode transfer with rs1, rs2, and rd valid. In this case, after the *nice_req_valid* and *nice_req_ready* handshake successfully, NICE-Core keeps *nice_req_ready* LOW until the data is ready to be written back.

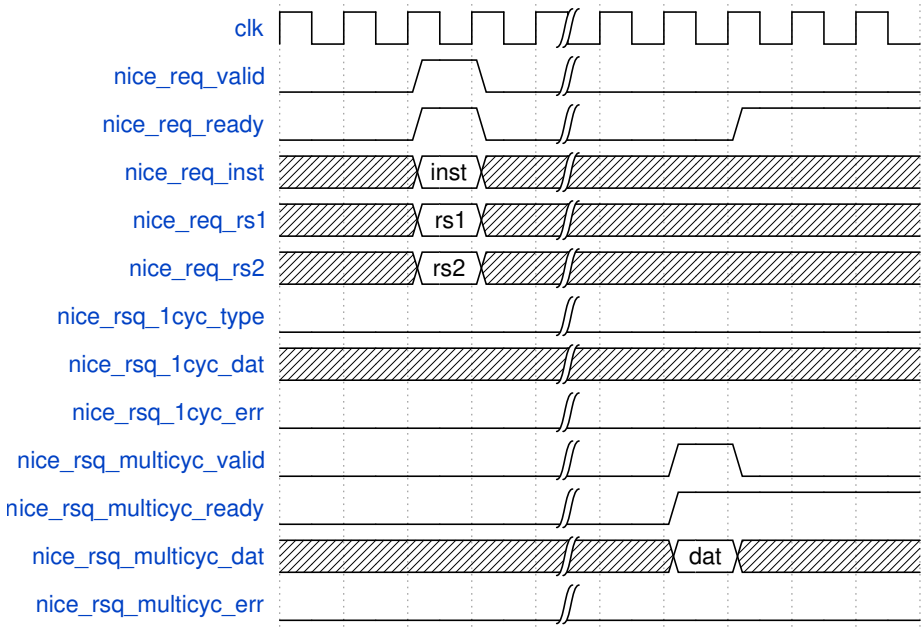


Fig. 30.4: NICE multi-cycle blocking mode transfer

30.5.2.2 Multi-Cycle Non-Blocking Mode

NICE multi-cycle non-blocking mode transfer without delay (page 292) shows four multi-cycle non-blocking mode transfers. In this case, NICE-Core can receive a new transfer while the previous transfer is still being processed. Each transfer needs four cycles to be done and sent the response, and because Master Core supports at most six outstanding transfer which is defined by RTL implementation, Master Core can send nice request contiguously without delay.

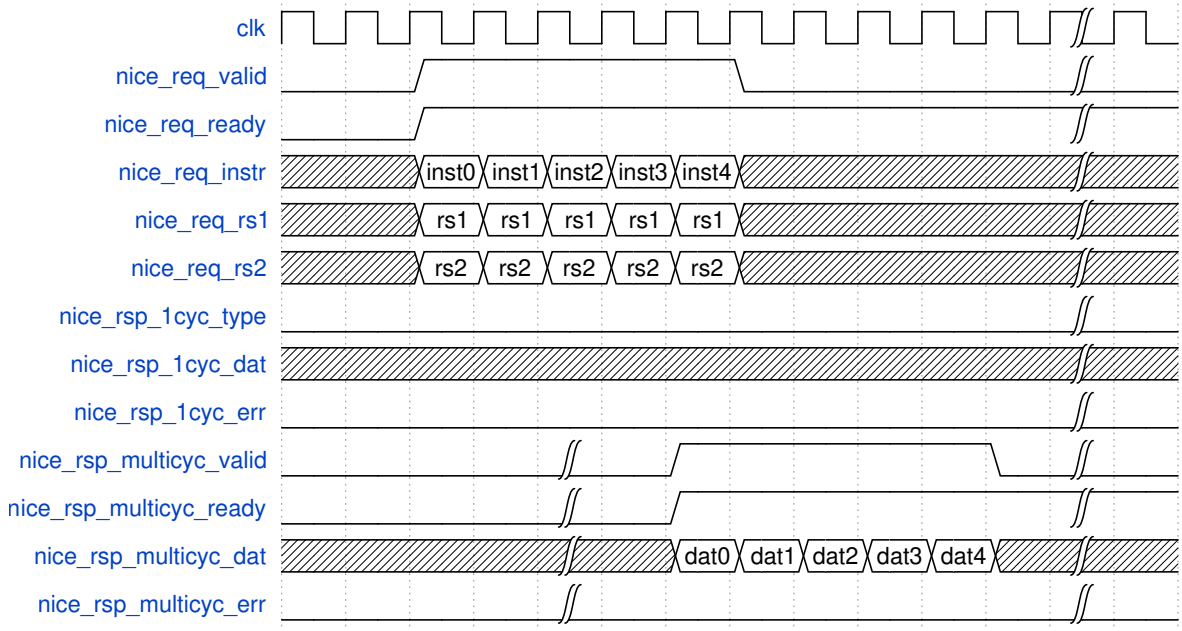


Fig. 30.5: NICE multi-cycle non-blocking mode transfer without delay

NICE multi-cycle non-blocking mode transfer with delay (page 293) shows four multi-cycle non-blocking transfer within eight cycles. In this case, NICE-Core can receive new transfer while the previous transfer is still being processed. Each transfer needs eight cycles to be done and sent response, but because Master Core supports at most four outstanding transfers, *nice_req_valid* must keep LOW while four NICE transfers are all being processed, until Master Core gets a response.

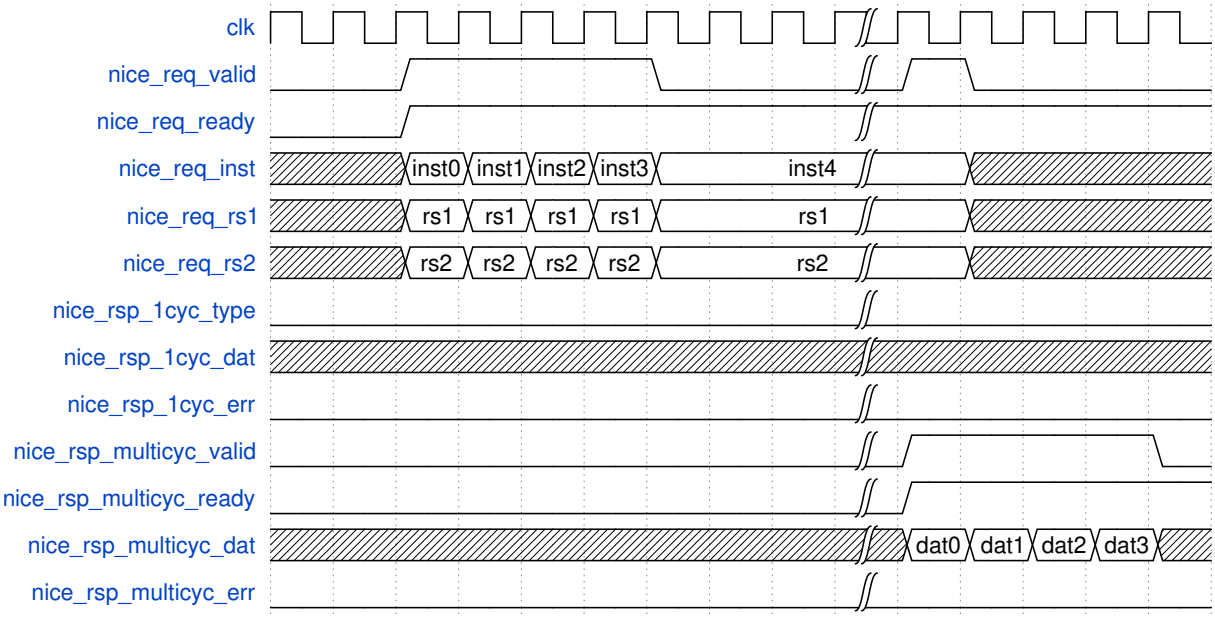


Fig. 30.6: NICE multi-cycle non-blocking mode transfer with delay

30.5.3 Response for VNICE

The VNICE response mechanism is the same of multi-cycle response of NICE.

30.6 NICE and VNICE Memory Access

Generally, NICE-Core can access memory via the NICE interface, and the operation should be included in a multi-cycle transfer. While a multi-cycle transfer is going to access memory,

the *nice_mem_holdup* should raise one cycle after *nice_req_valid* and *nice_req_ready* handshaking, and keep HIGH until NICE-Core finishes all nice memory accesses. This mechanism blocks the following load and store instruction, which can avoid some deadlock scenarios. With the help of *nice_mem_holdup*, NICE-Core can kick off one or several memory accesses at any time before the multi-cycle transfer is finished.

NICE accesses memory by ICB protocol. The ICB protocol contains command channel and response channel.

In the command channel, NICE-Core sends ICB request including *nice_icb_cmd_valid*, *nice_icb_cmd_addr*, *nice_icb_cmd_size* and *nice_icb_cmd_read*, then these signals are waiting for *nice_icb_cmd_ready* from Master Core. Once valid-ready handshakes successfully, Master Core processes the memory access operation with its LSU pipe.

In the response channel, Master Core sends *nice_icb_rsp_valid*, and *nice_icb_rsp_rdata* if it is a read operation, to NICE-Core and waits for *nice_icb_rsp_ready*.

nice_req_mmode indicates the current privilege mode in Master Core when a nice request is sent, and *nice_icb_cmd_mmode* should be the same mode with it when NICE-Core sends memory access request.

Note: NICE doesn't support misaligned memory access.

30.6.1 Single Memory Access Operation in Multi-Cycle Transfer

30.6.1.1 Single Memory Read Operation in Multi-Cycle Transfer

Single memory read operation in multi-cycle transfer (page 294) shows a single memory read operation in multi-cycle transfer.

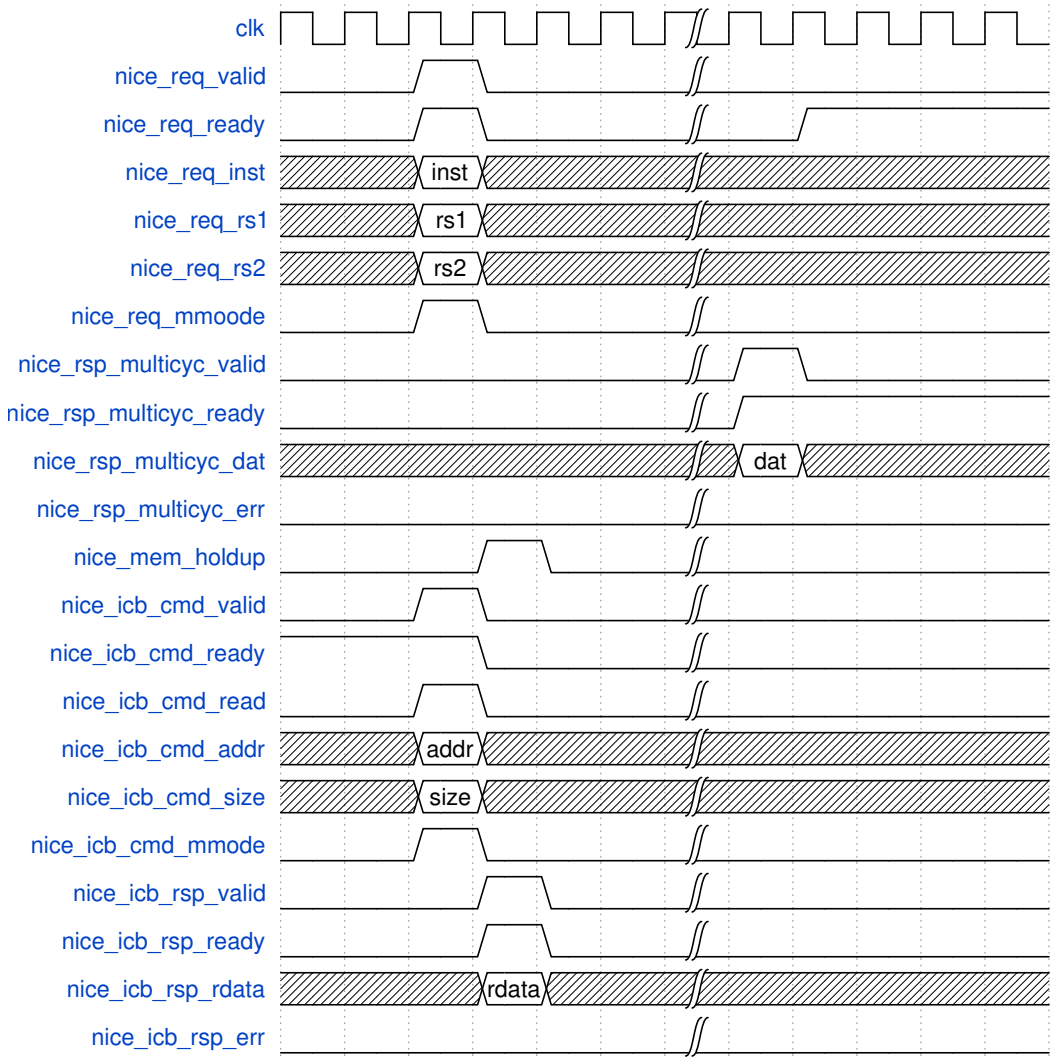


Fig. 30.7: Single memory read operation in multi-cycle transfer

30.6.1.2 Single Memory Write Operation in Multi-Cycle Transfer

Single memory write operation in multi-cycle transfer (page 295) shows single memory write operation in multi-cycle transfer.

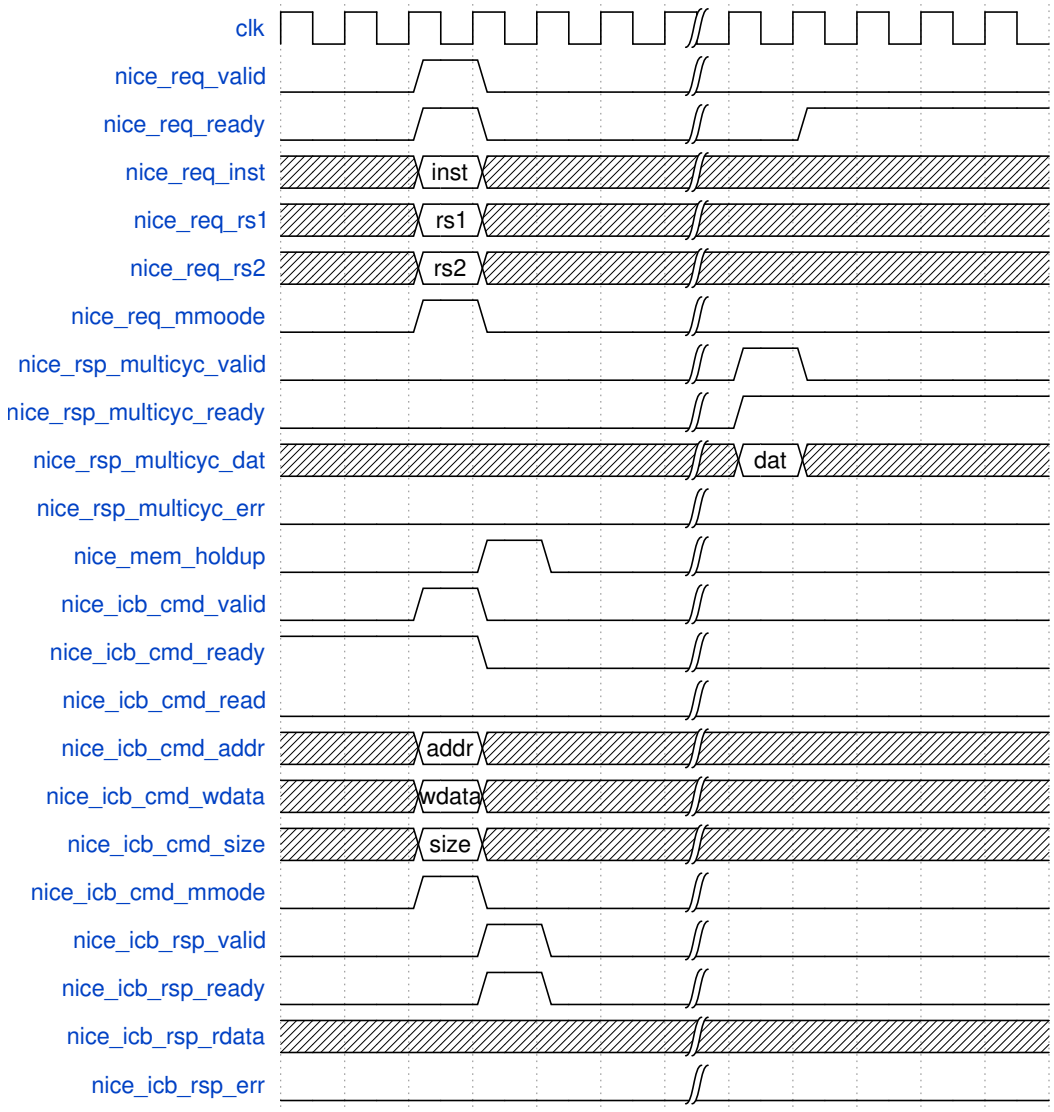


Fig. 30.8: Single memory write operation in multi-cycle transfer

30.6.2 Multiple Memory Access Operation in Multi-Cycle Transfer

Several memory accesses including read and write operation (page 296) shows several memory accesses including read and write operations in a multi-cycle transfer. The Maximum num of ICB outstanding transfer depends on the implementation of NICE-Core.

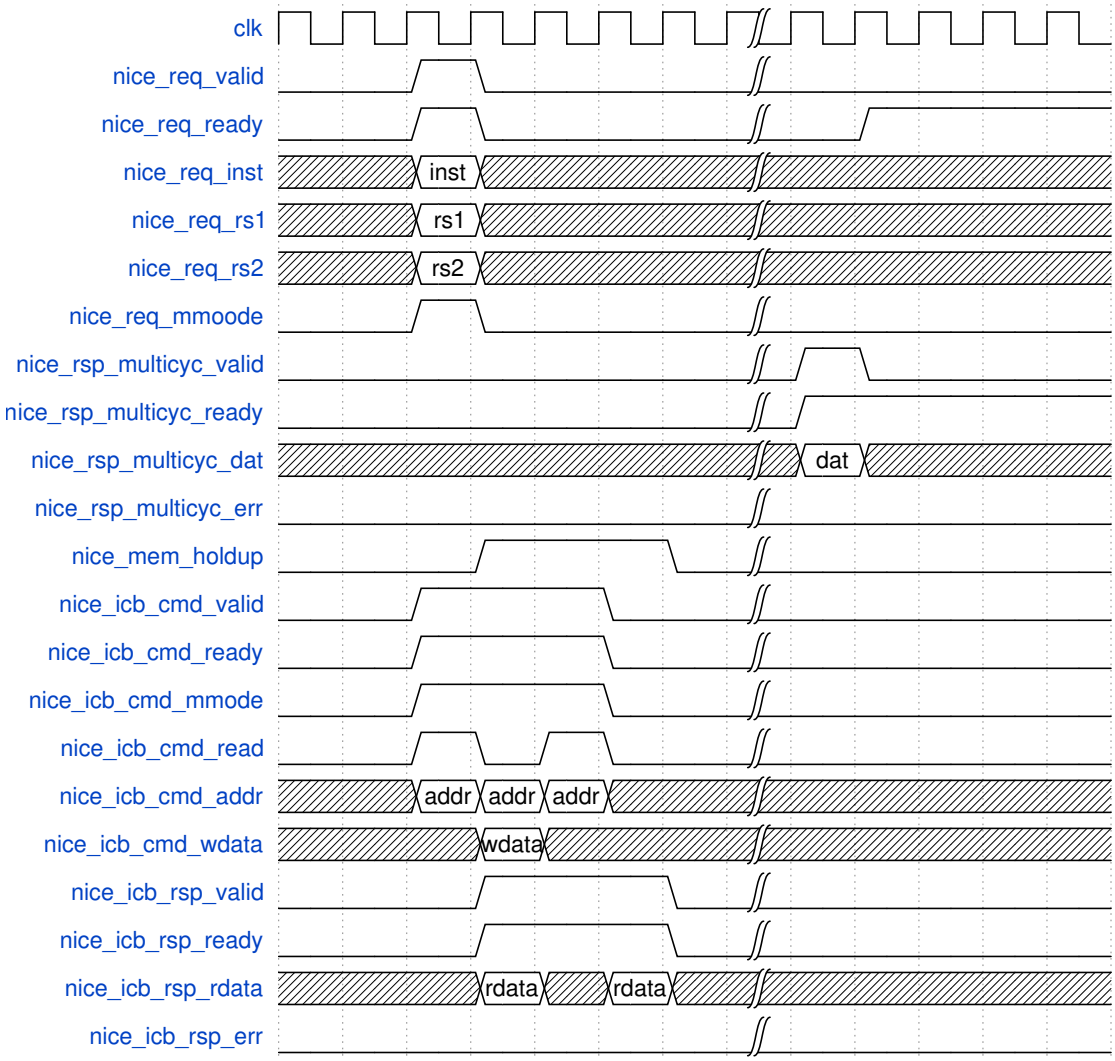


Fig. 30.9: Several memory accesses including read and write operation

30.6.3 VNICE Memory Access

The VNICE-Core can access memory via the NICE interface too, the mechanism is same with NICE-Core.

30.7 NICE and VNICE Response Error

NICE-Core can send an error response to Master Core when it detects any error. There are two error types: one-cycle response error and multi-cycle response error.

30.7.1 One-Cycle Response Error

For one-cycle response error, it probably should be an illegal instruction found by NICE-Core. *nice_req_1cyc_err* signal will keep one cycle with *nice_req_1cyc_type*. When the Master Core receives the one-cycle response, an illegal instruction exception will be raised.

One-cycle response error (page 297) shows one-cycle response error.

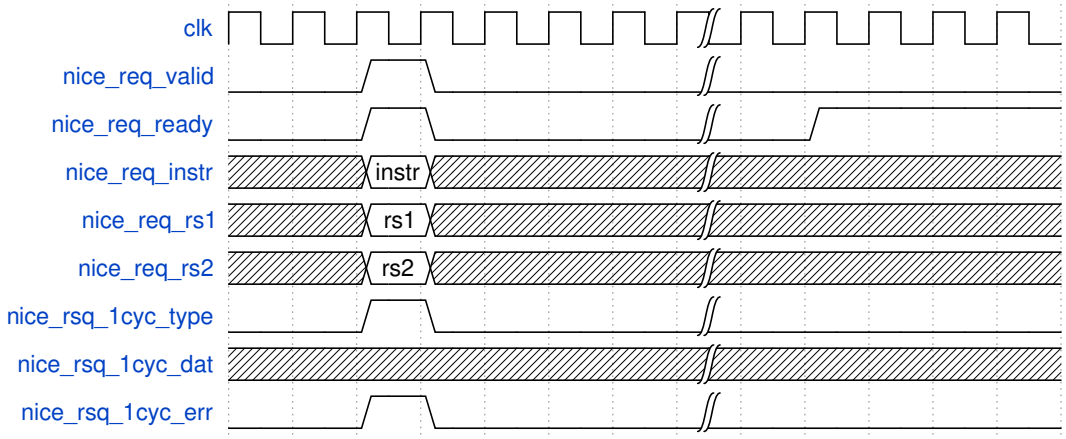


Fig. 30.10: One-cycle response error

30.7.2 Multi-Cycle Response Error

For multi-cycle response error, it could be a memory access error or multi-cycle execution error. *Multi-cycle response error* (page 297) shows a memory access error scenario. Master Core sends *nice_icb_rsp_err* to NICE-Core for each corresponding memory access response, then NICE-Core sends one cycle *nice_rsp_multicyc_err* to Master Core at multi-cycle response phase.

While the Master Core receives *nice_rsp_multicyc_err*, the result won't be written back to register file. Additionally, a load access fault exception will be raised(Exception Code =5) and *mdcause* will be set to 3.

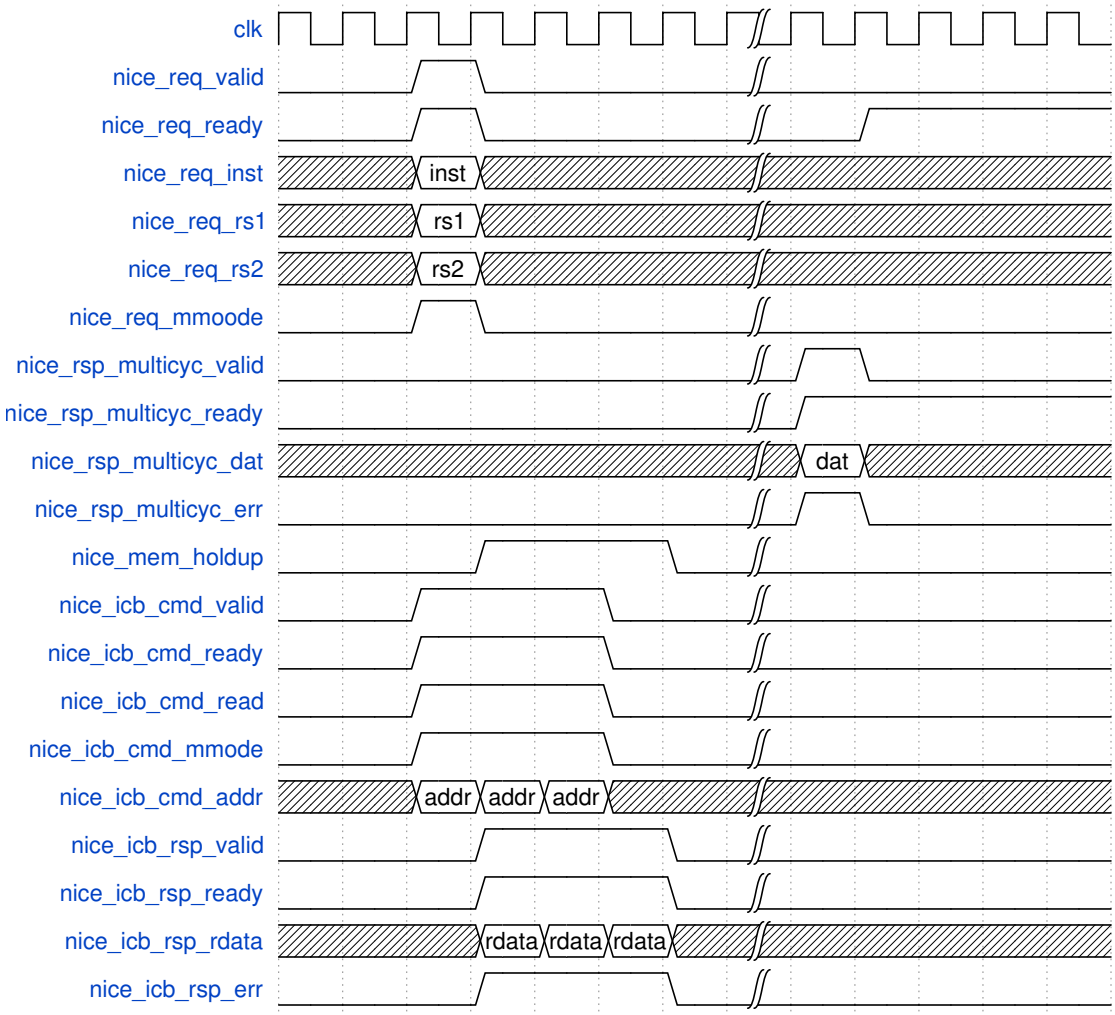


Fig. 30.11: Multi-cycle response error

Note: Currently 900 Series Core can't not support NICE One-Cycle Response Error, if there is error happened, please use Multi_Cycle Response Error instead.

30.7.3 VNICE Response Error

For VNICE, it uses user-defined decoding, so if there is illegal instruction error, it is detected and reported in decoding stage. If the VNICE-Core has computing error, it can use *vnice_wbck_vstat* and *vnice_wbck_fflag* to report, or it can consider to send interrupt to master core.

30.8 NICE and VNICE Demo Introduction

This chapter shows a NICE demo to explain the usage of NICE. The NICE demo includes hardware and software parts. For hardware, the NICE-Core is included in Nuclei RTL package and the main function is to sum a 3x3 matrix for each row and column. For software, please refer to *NICE Software Environment* (page 299).

The demo demonstrates a few instructions for mainly introducing the function of the NICE-Core and how to use these extended NICE instructions in software as well as the compiler.

30.8.1 Instruction In NICE Demo

This NICE Demo implements the following 3 instructions for NICE-Core.

Table 30.11: Instructions for NICE-Core

Instruction	Description	Encoding
CLW	Load 12-byte data from memory to row buffer.	opcode:0x5b, custom-2 xd:0, no write-back register xs1:1, rs1 is vaild for load address xs2:0, rs2 is invalid funct7:1
CSW	Store 12-byte data from row buffer to memory.	opcode:0x5b, custom-2 xd:0, no write-back register xs1:1, rs1 is valid for store address xs2:0 funct7:2
CACC	Sums a row of the matrix, and columns are accumulated automatically.	opcode:0x5b, custom-2 xd:1, rd is valid for write-back register xs1:1, rs1 is valid for the first address of a row xs2:0, rs2 is invalid funct7:6

In this NICE-Core, there is a 12-byte row buffer for saving the accumulated results of three columns. Before the operation of summing the matrix, the row buffer should be cleared with CLW instruction.

CACC instruction loads and accumulates all elements of a row one by one from memory and the result will be written back to register file directly. In addition, the columns are accumulated automatically for each CACC instruction and the results are saved in the row buffer in the NICE core. *The behavior of CACC instruction* (page 299) shows the behavior of CACC instruction

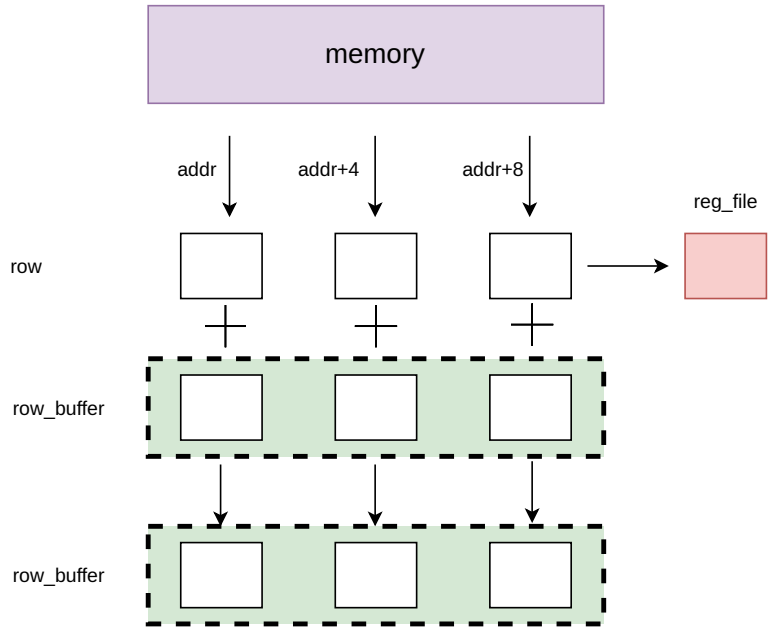


Fig. 30.12: The behavior of CACC instruction

After all summing operations are done, the row buffer could be written back to memory by CSW instruction.

30.8.2 NICE Software Environment

30.8.2.1 SDK Environment

The software environment used in this demo follows the Nuclei public SDK environment(https://github.com/Nuclei-Software/nuclei-sdk). Users could download the public software environment which contains some basic demo tests. After that, users can build the software environment based on the existing hello_world project, or create a new one according to the hello_world directory structure. This demo takes the first one, and names the project demo_nice.

30.8.2.2 Inline Assembly For User-defined Instruction

In this section, the CACC instruction will be an example to illustrate the usage of inline assembly. From *Instruction In NICE Demo* (page 298), we know that CACC is an R-type instruction and *R-type instruction format* (page 299) shows its format.

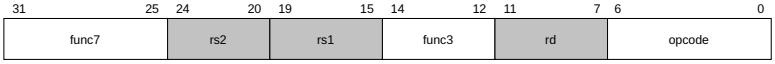


Fig. 30.13: R-type instruction format

All user-defined instructions in the assembler are implemented by the pseudo instruction *.insn.* Following is the usage of pseudo instruction *.insn* for R-type.

```
.insn r opcode, func3, func7 , rd, rs1, rs2
```

For CACC instruction, the inline assembly function is as below.

```
inline int custom_rowsum(int addr)

{

 int rowsum;

```

(continues on next page)

(continued from previous page)

```

 asm volatile (

   ".insn r 0x5b, 6, 6, %0, %1, x0"

   :"=r"(rowsum)

   :"r"(addr)

 );


 return rowsum;

}
```

custom _rowsum is the C function containing the CACC instruction. The function has an input and an output variable, respectively, corresponding to the starting address of a row of the matrix in the memory and the sum of the row elements.

The assembly part is contained in the asm volatile block. The *%0* indicates the output register whose value is reflected in the *rowsum* variable and register number is automatically assigned by the compiler. *%1* indicates the input register whose value is reflected in the *addr* variable and the register number is automatically assigned by the compiler.

'.insn' and *'r'* indicates this is a pseudo and R-type instruction. *'0x5b'* is the value of the opcode field, which means it is a NICE instruction belonging to custom-2. The first *'6'* is the value of func3 field, which means the destination and source 1 register are valid. The second *'6'* is the value of func7 field, indicating it is a CACC instruction.

This inline assembly function will sum all elements in a row, and the result will be written back to register file directly. In addition, the columns are accumulated automatically for each *custom_rowsum* function and the results are saved in the buffer in the NICE core. The data of the buffer can be written back to memory with CSW instruction. Before the operation of summing the matrix, the buffer should be cleared with CLW instruction.

The following are the other two inline assembly functions.

CLW:

```
inline void custom_lbuf(int addr)

{

 int zero = 0;


 asm volatile (

   ".insn r 0x5b, 2, 1, x0, %1, x0"

   :"=r"(zero)

   :"r"(addr)

 );


}
```

CSW:

```
// custom sbuf

inline void custom_sbuf(int addr)

```

(continues on next page)

(continued from previous page)

```
{

 int zero = 0;


 asm volatile (

   ".insn r 0x5b, 2, 2, x0, %1, x0"

   :"=r"(zero)

   :"r"(addr)

 );

}
```

30.8.2.3 Call Inline Assembly Function

The sum algorithm for row and column of the matrix is very simple. The conventional algorithm might be implemented by using two layers of for loops as shown below. *i* and *j* represent the number of rows and columns. *row_sum* and *col_sum* represent the sum of rows and columns.

```
// normal test case without NICE accelerator.

void normal_case(unsigned int array[ROW_LEN][COL_LEN], unsigned int col_sum[COL_LEN],␣
↪unsigned int row_sum[ROW_LEN])
{
   int i = 0, j = 0;

   unsigned int tmp = 0;
   for (i = 0; i < ROW_LEN; i++) {
       tmp = 0;
       for (j = 0; j < COL_LEN; j++) {
           col_sum[j] += array[i][j];
           tmp += array[i][j];
       }
       row_sum[i] = tmp;
   }
 }
```

The NICE algorithm compresses some operations and reduces some unnecessary loading and writing back operations, hence it has a faster execution speed. The following code shows the NICE algorithm. *i* represents the number of rows. *row_sum* and *col_sum* represent the sum of rows and columns. All the above inline assembly functions can be called directly in the C function.

```
// teat case using NICE accelerator.
void nice_case(unsigned int array[ROW_LEN][COL_LEN], unsigned int col_sum[COL_LEN],␣
↪unsigned int row_sum[ROW_LEN])
{
   int i;
   unsigned long init_buf[COL_LEN] = {0};

   custom_lbuf(init_buf);
   for (i = 0; i < ROW_LEN; i++) {
       row_sum[i] = custom_rowsum((unsigned long*)array[i]);
   }
```

(continues on next page)

(continued from previous page)

```
    custom_sbuf((unsigned long*)col_sum);
}
```

In this demo, for comparing the performance, the conventional algorithm and the NICE algorithm are both tested.

30.8.2.4 Main Function And Makefile

The main function in demo_nice.c file mainly initializes the test environment, including:

- Matrix initialization
- Enable NICE-Core
- Enable performance evaluation function
- Call two sum functions, and report the result of performance comparison.

The makefile script in the project has some changes, including adding insn.c file, which contains all inline assembly function, and adding *-fgnu89-line* in CFLAGS to support inline assembly function.

There are only several files in the whole project: Makefile, demo_nice.c, insn.c and insn.h. Users can compile the project to get the executable file and make it run in the processor with the NICE-Core.

30.8.2.5 Result Analysis

The result of NICE demo (page 303) shows the result of NICE demo test. The test contains both the conventional algorithm and the NICE optimization algorithm to calculate the 3x3 matrix. The optimization options are disabled and the debug information is enabled during the compilation process. As can be seen from the figure, the NICE optimization code functions correctly, and comparing to the conventional result, it has a significant reduction in the number of instructions and the operating cycle.

```
************************************************
** begin to sum the array using ordinary add sum
the element of array is :
        10      20      30
        20      30      40
        30      40      50

the sum of each row is :
                60      90      120
the sum of each col is :
                60      90      120


************************************************
** begin to sum the array using nice add sum
the element of array is :
        10      20      30
        20      30      40
        30      40      50

the sum of each row is :
                60      90      120
the sum of each col is :
                60      90      120
************************************************
** performance list
         normal:
              instret: 21119, cycle: 29624
         nice  :
              instret: 20710, cycle: 29091
************************************************
```

Fig. 30.14: The result of NICE demo

Performance at different optimization level (page 303) shows the performance at different optimization levels of the compiler toolchain.

The O0+Debug indicates that the Debug information output is enabled and the compiler optimization option is disabled, and the remaining four items respectively correspond to the different optimization levels of the compiler tool with debug information disabled.

00	O0+Debug		O0		O1		O2		O3	
01	Instruction number	Cycle number	Instruction number	Cycle number	Instruction number	Cycle number	Instruction number	Cycle number	Instruction number	Cycle number
Convention	21119	29624	654	859	411	532	391	511	391	512
NICE	20710	29091	247	354	90	133	86	128	86	128

Fig. 30.15: Performance at different optimization level

As can be seen from the result, the NICE core does improve the performance of the RISC-V core in this application, and it is foreseeable that the larger the matrix, the better the performance.

30.8.3 VNICE Demo Introduction

The VNICE demo to explain the usage of VNICE. The VNICE demo includes hardware and software parts. For hardware, the VNICE-Core is included in Nuclei RTL package and the main function is to use its VNP port to load data from external memory to Vector's registers, then do a complex multiply and store the results in Vector registers through its VNP port to external memory.

For the software demo, please contact Nuclei Support.

31 Nuclei Additional Xlcz Instruction for Codesize

31.1 Revision History

Rev.	Revision Date	Revised Section	Revised Content
1.1.0	2023/7/12	N/A	1.First version as the full English

31.2 Nuclei Additional Xlcz Instruction for Codesize

31.2.1 Code master table

Table 31.2: The summary is as follows

	31~12				**11-7**	**6-0**	**Type**	
xl.lb	0000	imm[7:0]	rs1(!rd)	001	rd	1111011	cust3	Auto-matic
xl.lbu	0001	imm[7:0]	rs1(!rd)		rd	1111011	cust3	
xl.lh	0010	imm[8:1]	rs1(!rd)		rd	1111011	cust3	
xl.lhu	0011	imm[8:1]	rs1(!rd)		rd	1111011	cust3	
xl.lw	0100	imm[9:2]	rs1(!rd)		rd	1111011	cust3	
xl.sb	0101	imm[7:5] rs2	rs1		imm[4:0]	1111011	cust3	
xl.sh	0110	imm[8:6] rs2	rs1		imm[5:1]	1111011	cust3	
xl.sw	0111	imm[9:7] rs2	rs1		imm[6:2]	1111011	cust3	
xl.lwu (rv64)	1000	imm[9:2]	rs1(!rd)		rd	1111011	cust3	
xl.ld (rv64)	1001	imm[10:3]	rs1(!rd)		rd	1111011	cust3	
xl.sd(rv64)	1010	imm[10:8] rs2	rs1		imm[7:3]	1111011	cust3	
xl.lgp.b	00	imm[15:0]		00	rd	1011011	cust2	Assembly only
xl.lgp.bu	01	imm[15:0]		00	rd	1011011	cust2	
xl.lgp.h	00	imm[15:1]		001	rd	1011011	cust2	
xl.lgp.hu	00	imm[15:1]		101	rd	1011011	cust2	
xl.lgp.w	00	imm[16:2]		010	rd	1011011	cust2	
xl.lgp.wu (rv64)	00	imm[16:2]		110	rd	1011011	cust2	
xl.lgp.d (rv64)	00	imm[17:3]		011	rd	1011011	cust2	
xl.sgp.b	11	imm[15:11] rs2	imm[10:5]	00	imm[4:0]	1011011	cust2	
xl.sgp.h	10	imm[15:11] rs2	imm[10:6]	001	imm[5:1]	1011011	cust2	
xl.sgp.w	10	imm[16:12] rs2	imm[11:7]	010	imm[6:2]	1011011	cust2	
xl.sgp.d (rv64)	10	imm[17:13] rs2	imm[12:8]	011	imm[7:3]	1011011	cust2	

continues on next page

Table 31.2 – continued from previous page

xl.addrchk	0	imm[10:5]	rs2	rs1	011	imm[4:1|11]	1111011	cust3	Api
xl.bezm	1	imm[10:5]	rs2	rs1	011	imm[4:1|11]	1111011	cust3	
xl.nzmsk	1110000		00000	rs1	001	rd	1111011	cust3	
xl.ffnz	1110001		00000	rs1	001	rd	1111011	cust3	
xl.beqi	offset[12|10:5]		imm[4:0]	rs1	010	offset[4:1|11]	1111011	cust3	Automatic
xl.bnei	offset[12|10:5]		imm[4:0]	rs1	100	offset[4:1|11]	1111011	cust3	
xl.muli	110	imm [8:5]	imm[4:0]	rs1	001	rd	1111011	uust3	
xl.addibne	1	scale[1:0]	uimm[9:1]	rs1	111	rd	1111011	cust3	
xl.slet	1110011		rs2	rs1	001	rd	1111011	cust3	Api
xl.sletu	1110100		rs2	rs1	001	rd	1111011	cust3	
xl.extract	00	uimm3[4:0]	uimm2[4:0]	rs1	101	rd	1111011	cust3	
xl.extractr	01	00000	rs2	rs1		rd	1111011	cust3	
xl.extractu	10	uimm3[4:0]	uimm2[4:0]	rs1		rd	1111011	cust3	
xl.extractur	11	00000	rs2	rs1		rd	1111011	cust3	
xl.insert	00	uimm3[4:0]	uimm2[4:0]	rs1	110	rd	1111011	cust3	
xl.bset	10	uimm3[4:0]	uimm2[4:0]	rs1		rd	1111011	cust3	
xl.bsetr	11	00000	rs2	rs1		rd	1111011	cust3	
xl.bclr	00	uimm3[4:0]	uimm2[4:0]	rs1	111	rd	1111011	cust3	
xl.bclrr	01	00000	rs2	rs1		rd	1111011	cust3	
xl.clb	1110101		00000	rs1	001	rd	1111011	cust3	
xl.fl1	1110110		00000	rs1		rd	1111011	cust3	
xl.ff1	1110111		00000	rs1		rd	1111011	cust3	
xl.fl0	1111000		00000	rs1		rd	1111011	cust3	
xl.ff0	1111001		00000	rs1		rd	1111011	cust3	
xl.bitrev	10110	uimm3[1:0]	uimm2[4:0]	rs1		rd	1111011	cust3	
xl.flh	0101		imm[8:1]	rs1	101	rd(freg)	1111011	cust3	assembly only
xl.flw	0110		imm[9:2]	rs1		rd(freg)	1111011	cust3	
xl.fld	0111		imm[10:3]	rs1		rd(freg)	1111011	cust3	
xl.fsh	1101	imm[8:6]	rs2(freg)	rs1		imm[5;1]	1111011	cust3	
xl.fsw	1110	imm[9:7]	rs2(freg)	rs1		imm[6:2]	1111011	cust3	
xl.fsd	1111	imm[10:8]	rs2(freg)	rs1		imm[7:3]	1111011	cust3	

Note: Xlcz extension is currently supported by 300 Series Core v4.1.0 and later version.

31.2.2 xl.lb

Format:

31 - 28	27 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0000	imm[7:0]	rs1(!=rd)	001	rd	1111011	cust3

Syntar:

```
xl.lb rd,imm(rs1)
```

Description:

This instruction implements loading single-byte signed data from the storage space at the address specified by rs1, while the base is updated to (rs1+imm). The destination address is specified by rs1.

Operations:

```
rd = Sext (Mem8(rst1)) ;

rs1 += imm[7:0] ;
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.3 xl.lbu

Format:

31 - 28	27 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0001	imm[7:0]	rs1(!=rd)	001	rd	1111011	cust3

Syntar:

```
xl.lbu rd,imm(rs1)
```

Description:

This instruction implements loading single-byte unsigned data from a storage space at a specified address,At the same time, the base is updated to (rs1+imm).The destination address is specified by rs1.

Operations:

```
rd = Zext(Mem8(rst1));

rs1 += imm[7:0];
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.4 xl.lh

Format:

31 - 28	27 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0010	imm[8:1]	rs1(!=rd)	001	rd	1111011	cust3

Syntax:

```
xl.lh rd, imm(rs1)
```

Description:

This instruction implements loading 2-bytes of signed data from the storage space at the specified address.At the same time, the base is updated to (rs1+imm).The destination address is specified by rs1.

Operations:

```
rd = Sext(Mem16(rs1));

rs1 +=imm[8:0];
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.5 xl.lhu

Format:

31 - 28	27 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0011	imm[8:1]	rs1(!=rd)	001	rd	1111011	cust3

Syntax:

```
xl.lhu rd ,imm(rs1)
```

Description:

This instruction implements loading 2-bytes of unsigned data from the storage space at the specified address.At the same time, the base is updated to (rs1+imm).The destination address is specified by rs1.

Operations:

```
rd = Zext(Mem8(rs1));

rs1 +=imm[8:0];
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.6 xl.lw

Format:

31 - 28	27 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0100	imm[9:2]	rs1(!=rd)	001	rd	1111011	cust3

Syntax:

```
xl.lw rd,imm(rs1)
```

Description:

This instruction implements loading 4-bytes of signed data from the storage space at the specified addre.At the same time, the base is updated to (rs1+imm).The destination address is specified by rs1.

Operations:

```
rd = Sext(Mem32(rs1));

rs1 +=imm[9:0];
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.7 xl.sb

Format:

31 - 28	27 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0101	imm[7:5]	rs2	rs1	001	imm[4:0]	1111011	cust3

Syntax:

```
xl.sb rs2,imm(rs1)
```

Description:

This instruction stores single-byte data in rs2 to a specified address space. At the same time,the base is updated to (rs1+imm).

Operations:

```
Mem8(rs1)= rs2;

rs1 +=imm[7:0];
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.8 xl.sh

Format:

31 - 28	27 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0110	imm[8:6]	rs2	rs1	001	imm[5:1]	1111011	cust3

Syntax:

```
xl.sh rs2,imm(rs1)
```

Description:

This instruction stores 2-byte data in rs2 to a specified address space. At the same time,the base is updated to (rs1+imm) and save to rs1.

Operations:

```
Mem16(rs1)= rs2;

rs1 +=imm[8:0];
```

Exceptions:

Unaligned exception, illegal instruction exception

;

31.2.9 xl.sw

Format:

31 - 28	27 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0111	imm[9:7]	rs2	rs1	001	imm[6:2]	1111011	cust3

Syntax:

```
xl.sw rs2,imm(rs1)
```

Description:

The instruction stores 4-bytes of data in rs2 into the storage space at the specified address.At the same time,the base is updated to (rs1+imm) and save to rs1.

Operations:

```
Mem32(rs1)= rs2;

rs1 +=imm[9:0];
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.10 xl.lwu(rv64)

Format:

31 - 28	27 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1000	imm[9:2]	rs1(!=rd)	001	rd	1111011	cust3

Syntax:

```
xl.lwu rd,imm(rs1)
```

Description:

This instruction implements loading 4-bytes of unsigned data from the storage space at the specified addre.At the same time ,the base is updated to (rs1+imm) and saved to rs1.

Operations:

```
rd = Zext(Mem32(rs1));

rs1 +=imm[9:0];
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.11 xl.ld(rv64)

Format:

31 - 28	27 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1001	imm[10:3]	rs1(!=rd)	001	rd	1111011	cust3

Syntax:

```
xl.ld rd,imm(rs1)
```

Description:

The instruction implements loading 8-bytes of signed data from the storage space at the specified address.At the same, time the base is updated to (rs1+imm) and saved to rs1.

Operations:

```
rd = Sext(Mem64(rs1));

rs1 +=imm[10:0];
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.12 xl.sd(rv64)

Format:

31 - 28	27 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1010	imm[10:8]	rs2	rs1	001	imm[7:3]	1111011	cust3

Syntax:

```
xl.sd rs2,imm(rs1)
```

Description:

The instruction stores 8-bytes of data in rs2 into the storage space at the specified address. At the same time,The base is updated to (rs1+imm) and save to rs1.

Operations:

```
Mem64(rs1)= rs2;

rs1 +=imm[10:0];
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.13 xl.lgp.b

Format:

31 - 30	29 - 14	13 - 12	11 - 7	6 - 0	Type
00	imm[15:0]	00	rd	1011011	cust2

Syntax:

```
xl.lgp.b rd,imm
```

Description:

The instruction implements loading 1-byte of signed data from the storage space at the specified address.The destination address is determined by GP+imm[15:0].

Operations:

```
rd = Sext(Mem8(gp+imm));
```

Exceptions:

illegal instruction exception

31.2.14 xl.lgp.bu

Format:

31 - 30	29 - 14	13 - 12	11 - 7	6 - 0	Type
01	imm[15:0]	00	rd	1011011	cust2

Syntax:

```
xl.lgp.bu rd,imm
```

Description:

The instruction implements loading 1-byte of unsigned data from the storage space at the specified address.The destination address is determined by GP+imm[15:0].

Operations:

```
rd = Zext(Mem8(gp+imm));
```

Exceptions:

illegal instruction exception

31.2.15 xl.lgp.h

Format:

31 - 30	29 - 15	14 - 12	11 - 7	6 - 0	Type
00	imm[15:1]	001	rd	1011011	cust2

Syntax:

```
xl.lgp.h rd,imm
```

Description:

The instruction implements loading 2-byte of signed data from the storage space at the specified address.The destination

address is determined by GP+imm<<1.

Operations:

```
rd = Sext(Mem16(gp+imm<<1));
```

Exceptions:

illegal instruction exceptio

31.2.16 xl.lgp.hu

Format:

31 - 30	29 - 15	14 - 12	11 - 7	6 - 0	Type
00	imm[15:1]	101	rd	1011011	cust2

Syntax:

```
xl.lgp.hu rd,imm
```

Description:

The instruction implements loading 2-byte of unsigned data from the storage space at the specified address.The destination address is determined by GP+imm<<1.

Operations:

```
rd = Zext(Mem16(gp+imm<<1));
```

Exceptions:

illegal instruction exception

31.2.17 xl.lgp.w

Format:

31 - 30	29 - 15	14 - 12	11 - 7	6 - 0	Type
00	imm[16:2]	010	rd	1011011	cust2

Syntax:

```
xl.lgp.w rd,imm
```

Description:

The instruction implements loading dobule 4-byte of signed data from the storage space at the specified address.The destination address is determined by GP+imm<<2.

Operations:

```
rd = Sext(Mem32(gp+imm<<2));
```

Exceptions:

illegal instruction exception

31.2.18 xl.lgp.wu(rv64)

Format:

31 - 30	29 - 15	14 - 12	11 - 7	6 - 0	Type
00	imm[16:2]	110	rd	1011011	cust2

Syntax:

```
xl.lgp.wu rd,imm
```

Description:

The instruction implements loading 4-byte of unsigned data from the storage space at the specified address.The destination address is determined by GP+imm<<2.

Operations:

```
rd = Zext(Mem32(gp+imm<<2));
```

Exceptions:

illegal instruction exception

31.2.19 xl.lgp.d(rv64)

Format:

31 - 30	29 - 15	14 - 12	11 - 7	6 - 0	Type
00	imm[17:3]	011	rd	1011011	cust2

Syntax:

```
xl.lgp.d rd,imm
```

Description:

The instruction implements loading 8-byte of signed data from the storage space at the specified address.The destination address is determined by GP+imm[16:3]<<3.

Operations:

```
rd = Mem64(gp+imm<<3);
```

Exceptions:

illegal instruction exception

31.2.20 xl.sgp.b

Format:

31 - 30	29 - 25	24 - 20	19 - 14	13 - 12	11 - 7	6 - 0	Type
11	imm[15:11]	rs2	imm[10:5]	00	imm[4:0]	1011011	cust2

Syntax:

```
xl.sgp.b rs2,imm
```

Description:

Saves the lowest byte in rs2 to the storage space at the specified address.The destination address is determined by GP+imm.

Operations:

```
Mem8(gp+imm) = rs2;
```

Exceptions:

illegal instruction exception

31.2.21 xl.sgp.h

Format:

31 - 30	29 - 25	24 - 20	19 - 14	13 - 12	11 - 7	6 - 0	Type
10	imm[15:11]	rs2	imm[10:6]	001	imm[5:1]	1011011	cust2

Syntax:

```
xl.sgp.h rs2,imm
```

Description:

Saves the lowest 2-byte in rs2 to the storage space at the specified address.The destination address is determined by GP+imm<<1.

Operations:

```
Mem16(gp+imm<<1) = rs2;
```

Exceptions:

illegal instruction exception

31.2.22 xl.sgp.w

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
10	imm[16:12]	rs2	imm[11:7]	010	imm[6:2]	1011011	cust2

Syntax:

```
xl.sgp.w rs2,imm
```

Description:

Saves the lowest byte in rs2 to the storage space at the specified address.The destination address is determined by GP+imm<<2.

Operations:

```
Mem32(gp+imm<<2) = rs2;
```

Exceptions:

illegal instruction exception

31.2.23 xl.sgp.d(rv64)

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
10	imm[17:13]	rs2	imm[12:8]	011	imm[7:3]	1011011	cust2

Syntax:

```
xl.sgp.d rs2,imm
```

Description:

Saves the lowest 8-byte in rs2 to the storage space at the specified address.The destination address is determined by GP+imm<<3.

Operations:

```
Mem64(gp+imm<<3) = rs2;
```

Exceptions:

illegal instruction exception

31.2.24 xl.addrchk

Format:

31	30 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0	imm[10:5]	rs2	rs1	011	imm[4:1|11]	1111011	cust3

Syntax:

```
xl.addrchk rs1,rs2 imm
```

Description:

Check that the values of rs1 and rs2 are aligned.For rv32, this is to check whether the lower 2-bit is 0.For rv64, it checks whether the lower 3-bits are 0.If either of rs1 or rs2 does not meet the alignment conditions, it jumps to the specified address.This command is used to check string entry addresses.

Operations:

```
for rv32 MASK=0x3
for rv64 MASK=0x7

cond = (rs1|rs2)&MASK;
if cond == 1 then branch to PC+imm<<1
else branch to PC+4
```

Exceptions:

illegal instruction exception

Intrinsic

```
TODO
```

31.2.25 xl.bezm

Format:

31	30 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1	imm[10:5]	rs2	rs1	011	imm[4:1|11]	1111011	cust3

Syntax:

```
xl.bezm rs1,rs2 imm
```

Description:

Check that rs1 contains 0-bytes and that the values of rs1 and rs2 are equal.If rs1 contains 8'h00, or rs1 is not equal to rs2, skip to the specified address.Also save the difference to a fixed register (tentative x31).(This instruction is used to check the equality of 4-byte or 8-byte strings)

Operations:

```
rsx_msk = rsx&0x7F7F7F7F; //rv32, rsx=[rs1,rs2]
rsx_msk = rsx&0x7F7F7F7F7F7F7F7F; //rv64, rsx=[rs1,rs2]

cond = (rs1_msk_b[x] == 8'h00)|(rs1_msk != rs2_msk)
if cond == 1 then branch to PC+imm<<1
else branch to PC+4

x31_byte0 = (rs1_msk_byte0 - rs2_msk_byte0);
x31_byte1 = (rs1_msk_byte1 - rs2_msk_byte1);
x31_byte2 = (rs1_msk_byte2 - rs2_msk_byte2);
x31_byte3 = (rs1_msk_byte3 - rs2_msk_byte3);
for rv64, also need:
x31_byte4 = (rs1_msk_byte4 - rs2_msk_byte4);
x31_byte5 = (rs1_msk_byte5 - rs2_msk_byte5);
x31_byte6 = (rs1_msk_byte6 - rs2_msk_byte6);
x31_byte7 = (rs1_msk_byte7 - rs2_msk_byte7);
```

Exceptions:

illegal instruction exception

Intrinsic

```
TODO
```

31.2.26 xl.nzmsk

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1110000	00000	rs1	001	rd	1111011	cust3

Syntax:

```
xl.nzmsk rd,rs1
```

Description:

The position of the first 0 byte of rs1 is checked from the low byte and the corresponding mask value is generated accordingly. If not, write back to full FF. Assuming that the first 0 byte of rs1 is in the X byte position, 0 to X bytes are written to full FF, and the remaining bytes are written to 0. This instruction is used to determine the end of a string and generate the corresponding mask value accordingly.

Operations:

```
rd[0] = 8'hff;
rd_b[x] = (rs1_b[x-1] == 8'h00? 8'h00 : 8'hff) & rd_b[x-1]; x>=1

rs1=0x12345678,then rd=0xffffffff;
rs1=0x12345600,then rd=0x000000ff;
rs1=0x12340078,then rd=0x0000ffff;
rs1=0x12340000,then rd=0x000000ff;
rs1=0x12005678,then rd=0x00ffffff;
```

Exceptions:

illegal instruction exception

Intrinsic

```
unsigned int __xl_nzmsk_ (unsinged int a); //rv32
unsigned long long __xl_nzmsk_ (unsinged long long a); //rv64
```

31.2.27 xl.ffnz

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1110001	00000	rs1	001	rd	1111011	cust3

Syntax:

```
xl.ffnz rd,rs1
```

Description:

Check the result in bytes starting from the lower byte of rs1. Returns the value of the first non-zero byte.This instruction is used to return the result of a string comparison, the difference.

Operations:

```
if (rs1_b[0] != 8'h00)
 rd = rs1_b[0]
else if  (rs1_b[1] != 8'h00)
 rd = rs1_b[1]

else  if  (rs1_b[2] != 8'h00)
 rd = rs1_b[2]
else  if  (rs1_b[3] != 8'h00)
 rd = rs1_b[3]

 `ifdef RV64
else  if  (rs1_b[4] != 8'h00)
 rd = rs1_b[4]
else  if  (rs1_b[5] != 8'h00)
 rd = rs1_b[5]

 else  if  (rs1_b[6] != 8'h00)
 rd = rs1_b[6]
else  if  (rs1_b[7] != 8'h00)
 rd = rs1_b[7]
`endif
else
 rd = 8'h00;
```

Exceptions:

illegal instruction exception

Intrinsic

```
unsigned char __xl_ffnz_ (unsinged int a); //rv32
unsigned char __xl_ffnz_ (unsinged long long a); //rv64
```

31.2.28 xl.beqi

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
offset[12|10:5]	imm[4:0]	rs1	010	offset[4:1|11]	1111011	cust3

Syntax:

```
xl.beq rs1,cimm,ofst
```

Description:

Compare rs1 and cimm immediate numbers. If they are equal, jump to PC+ofset for execution.

Operations:

```
if (rs1 == sext(cimm)) PC=PC+ofst
else PC=PC+4
```

Exceptions:

illegal instruction exception

31.2.29 xl.bnei

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
offset[12|10:5]	imm[4:0]	rs1	100	offset[4:1|11]	1111011	cust3

Syntax:

```
xl.bnei rs1,cimm,ofst
```

Description:

Compare rs1 and cimm immediate numbers. If they are not equal, jump to PC+ofset for execution.

Operations:

```
if (rs1 != sext(cimm)) PC=PC+ofst
else PC=PC+4
```

Exceptions:

illegal instruction exception

31.2.30 xl.muli

Format:

31 - 29	28 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
110	imm[8:5]	imm[4:0]	rs1	001	rd	1111011	cust3

Syntax:

```
xl.muli rd, rs1, imm
```

Description:

Multiply rs1 with imm after symbol bit extension,and save to rd.

Operations:

```
rd = rs1*sext(imm)
```

Exceptions:

illegal instruction exception

31.2.31 xl.addibne

Format:

31	30 - 29	28 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1	scale[1:0]	uimm[9:1]	rs1	111	rd	1111011	cust3

Syntax:

```
xl.addibne rd, rs1, scale, uimm
```

Description:

Compare rd with rs1+scale, jump to pc-uimm if they are not equal, and save the results of rd+scale to rd.

Operations:

```
if rs1 != rd then branch to PC-uimm
else branch to PC+4/2

ofst= (scale[1:0] == 2'b00) ? 1 :
     (scale[1:0] == 2'b01) ? 2 :
     (scale[1:0] == 2'b10) ? 4 : 8;

rd=rd+ofst.
```

Exceptions:

illegal instruction exception

31.2.32 xl.slet

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1110011	rs2	rs1	001	rd	1111011	cust3

Syntax:

```
xl.slet rd,rs1,rs2
```

Description:

If rs1 is less than or equal to rs2, return 1, otherwise return 0, signed comparison.

Operations:

```
rd = (rs1 <= rs2) ? 1 : 0;
```

Exceptions:

illegal instruction exception

Intrinsic

```
unsigned int __xl_slet_ (singed int a, signed int b); //rv32
unsigned long long __xl_slet_ (singed long long a, signed long long b); //rv64
```

31.2.33 xl.sletu

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1110100	rs2	rs1	011	rd	1111011	cust3

Syntax:

```
xl.sletu rd,rs1,rs2
```

Description:

If rs1 is less than or equal to rs2, return 1, otherwise return 0, signed comparison.

Operations:

```
rd = (rs1 <= rs2) ? 1 : 0;
```

Exceptions:

illegal instruction exception

Intrinsic

```
unsigned int __xl_sletu_ (unsinged int a, unsigned int b); //rv32
unsigned long long __xl_sletu_ (unsinged long long a, unsigned long long b); //rv64
```

31.2.34 xl.extract

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
00	uimm3[4:0]	uimm2[4:0]	rs1	101	rd	1111011	cust3

Syntax:

```
xl.extract rd, rs1, uimm3, uimm2
```

Description:

Intercept the data between [uimm3 + uimm2-1:uimm2] from rs1, then do signed bit extension, and save it to rd.

Operations:

```
t=min(uimm3+uimm2-1, 31); //top
b= uimm2;//bottom
rd = Sext(rs1[t:b])
```

Exceptions:

illegal instruction exception

Intrinsic

```
signed int __xl_extract_ (unsinged int a, unsigned char imm1,  unsigned char imm2); //rv32
signed long long  __xl_extract_ (unsinged long long a,  unsigned char imm1,  unsigned char␣
↪imm2); //rv64
```

31.2.35 xl.extractr

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
01	00000	rs2	rs1	101	rd	1111011	cust3

Syntax:

```
xl.extractr rd, rs1, rs2
```

Description:

Intercept the data between[rs2[9:5]+rs2[4:0]-1 : rs2[4:0]] from rs1, then do signed bit extension, and save it to rd.

Operations:

```
t = min(rs2[9:5]+rs2[4:0]-1, 31);
b = rs2[4:0];
rd = Sext(rs1[t:b])
```

Exceptions:

illegal instruction exception

Intrinsic

```
signed int __xl_extractr_ (unsinged int a, unsigned short b); //rv32
signed long long  __xl_extractr_ (unsinged long long a,  unsigned short b); //rv64
```

31.2.36 xl.extractu

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
10	uimm3[4:0]	uimm2[4:0]	rs1	101	rd	1111011	cust3

Syntax:

```
xl.extractu rd, rs1, uimm3, uimm2
```

Description:

Intercept the data between[uimm3 + uimm2-1:uimm2] from rs1, then do the unsigned bit extension, and save it to rd.

Operations:

```
t=min(uimm3+uimm2-1, 31);
b= uimm2;
rd = Zext(rs1[t:b])
```

Exceptions:

illegal instruction exception

Intrinsic

```
unsigned int __xl_extractu_ (unsinged int a, unsigned char imm1,  unsigned char imm2); //
↪rv32
unsigned long long  __xl_extractu_ (unsinged long long a,  unsigned char imm1,  unsigned␣
↪char imm2); //rv64
```

31.2.37 xl.extractur

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
11	00000	rs2	rs1	101	rd	1111011	cust3

Syntax:

```
xl.extractur rd, rs1, rs2
```

Description:

Intercept the data between[rs2[9:5] + rs2[4:0]-1 : rs2[4:0]] from rs1, then do the unsigned bit extension, and save it to rd.

Operations:

```

```

Exceptions:

illegal instruction exception

Intrinsic

```
unsigned int __xl_extractur_ (unsinged int a, unsigned short b); //rv32
unsigned long long  __xl_extractur_ (unsinged long long a,  unsigned short b); //rv64
```

31.2.38 xl.insert

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
00	uimm3[4:0]	uimm2[4:0]	rs1	110	rd	1111011	cust3

Syntax:

```
xl.insert rd, rs1, uimm3, uimm2
```

Description:

Change the data in the [uimm3+uimm2:uimm2] field of rd to rs1[ls3:0].The data of other bit fields in rd remains unchanged.

Operations:

```
td=min(uimm3+uimm2, 31);
bd=uimm2;
ts=uimm3-(max(uimm3+uimm2,32)-32)
rd[td: bd] = rs1[ts:0])
```

Exceptions:

illegal instruction exception

Intrinsic

```
signed int __xl_insert_ (unsinged int a, unsigned char imm1,  unsigned char imm2); //rv32
signed long long  __xl_insert_ (unsinged long long a,  unsigned char imm1,  unsigned char␣
↪imm2); //rv64
```

31.2.39 xl.bset

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
10	uimm3[4:0]	uimm2[4:0]	rs1	110	rd	1111011	cust3

Syntax:

```
xl.bset rd, rs1, uimm3, uimm2
```

Description:

The data in the [uimm3+uimm2:uimm2] bit field in rs1 is modified to complete 1, and the rest is kept unchanged and saved to rd.

Operations:

```
mask = ((1 << (uimm3+1)) - 1) << uimm2;//uimm3+uimm2 <= 31 or 63;
rd = rs1 | mask;
```

Exceptions:

illegal instruction exception

Intrinsic

```
signed int __xl_bset_ (unsinged int a, unsigned char imm1,  unsigned char imm2); //rv32
signed long long  __xl_bset_ (unsinged long long a,  unsigned char imm1,  unsigned char␣
↪imm2); //rv64
```

31.2.40 xl.bsetr

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
11	00000	rs2	rs1	110	rd	1111011	cust3

Syntax:

```
xl.bsetr rd, rs1, rs2
```

Description:

The data in the [rs2[9:5] + rs2[4:0] : rs2[4:0] bit field in rs1 is modified to complete 1, and the rest is kept unchanged and saved to rd.

Operations:

```
mask = ((1 << (rs2[9:5]+1)) -1) << rs2[4:0];//rs2[9:5]+rs2[4:0] <= 31 or 63;
rd = rs1 | mask;
```

Exceptions:

illegal instruction exception

Intrinsic

```
signed int __xl_bsetr_ (unsinged int a, unsigned short b); //rv32
signed long long  __xl_bsetr_ (unsinged long long a,  unsigned short b); //rv64
```

31.2.41 xl.bclr

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
00	uimm3[4:0]	uimm2[4:0]	rs1	111	rd	1111011	cust3

Syntax:

```
xl.bclr rd, rs1, uimm3, uimm2
```

Description:

The data in the [uimm3+uimm2:uimm2] bit field in rs1 is modified to complete 0, and the rest is kept unchanged and saved to rd.

Operations:

```
mask = ~(((1 << (uimm3+1))-1) << uimm2); //uimm3+uimm2 <= 31 or 63;
rd = rs1 & mask;
```

Exceptions:

illegal instruction exception

Intrinsic

```
signed int __xl_clr_ (unsinged int a, unsigned char imm1,  unsigned char imm2); //rv32
signed long long  __xl_clr_ (unsinged long long a,  unsigned char imm1,  unsigned char␣
↪imm2); //rv64
```

31.2.42 xl.bclrr

Format:

31 - 30	29 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
01	00000	rs2	rs1	111	rd	1111011	cust3

Syntax:

```
xl.bsetr rd, rs1, rs2
```

Description:

The data in the [rs2[9:5] + rs2[4:0] : rs2[4:0] bit field in rs1 is modified to complete 0, and the rest is kept unchanged and saved to rd.

Operations:

```
mask = ~(((1 << (rs2[9:5]+1))-1) << rs2[4:0]); //rs2[9:5]+rs2[4:0] <= 31 or 63;
rd = rs1 & mask;
```

Exceptions:

illegal instruction exception

Intrinsic

```
signed int __xl_bclrr_ (unsinged int a, unsigned short b); //rv32
signed long long  __xl_bclrr_ (unsinged long long a,  unsigned short b); //rv64
```

31.2.43 xl.clb

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1110101	00000	rs1	001	rd	1111011	cust3

Syntax:

```
xl.clb rd, rs1
```

Description:

Count the leading bit (0 or 1) in rs1, that is, the number of consecutive bits since MSB.If rs1 is 0, 0 is returned.

Operations:

```
uint8 __clb__ (uint rs1)
{
  uint8 header = rs1&0x80000000;
  uint tmp=rs1;
  uint8 cnt=0;
  if (rs1 == 0) return 0;
  for (uint8 i=0;i<32; i++)
  {
    if ((tmp&0x80000000) && (header)) cnt++;
    else if (~(tmp&0x80000000) && ~(header)) cnt ++;
    else break;

    tmp = tmp<<1;
  }
```

(continues on next page)

(continued from previous page)

```
 return cnt;
}
```

Exceptions:

illegal instruction exception

31.2.44 xl.fl1

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1110110	00000	rs1	001	rd	1111011	cust3

Syntax:

```
xl.fl1 rd, rs1
```

Description:

Starting with MSB, find out where the last bit1 in rs1 is located.If rs1 is 0, then rd is equal to 32 or 64.

Operations:

```
uint8 __fl1__ (uint rs1)
{
 cnt=0;
 while(!(rs1&0x1)) {rs1=rs1>>1;cnt++;}
 return cnt;
}

rs1=0x80001000, then rd=12
```

Exceptions:

illegal instruction exception

31.2.45 xl.ff1

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1110111	00000	rs1	001	rd	1111011	cust3

Syntax:

```
xl.ff1 rd, rs1
```

Description:

Starting with MSB, find the location of the first bit1 in rst1.If rs1 is 0, then rd is equal to 32 or 64.

Operations:

```
uint8 __ff1__ (uint rs1)
{
 XLEN=32;//for rv64, XLEN=64;
 if (rs1==0) return XLEN;
 cnt=XLEN-1;
 while(!(rs1&0x8000_0000)) {rs1=rs1<<1;cnt--;}
```

(continues on next page)

(continued from previous page)

```
 return cnt;
}

rs1=0x80001000, then rd=31
```

Exceptions:

illegal instruction exception

31.2.46 xl.fl0

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1111000	00000	rs1	001	rd	1111011	cust3

Syntax:

```
xl.fl0 rd, rs1
```

Description:

Starting with MSB,find the location of the last bit0 in rst1.If rs1 is 0, then rd is equal to 32 or 64.

Operations:

```
uint8 __fl0__ (uint rs1)
{
 cnt=0;
 while((rs1&0x1)) {rs1=rs1>>1;cnt++;}
 return cnt;
}

rs1=0x80001000, then rd=0
```

Exceptions:

illegal instruction exception

31.2.47 xl.ff0

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1111001	00000	rs1	001	rd	1111011	cust3

Syntax:

```
xl.ff0 rd, rs1
```

Description:

Starting with MSB, find out where the last bit1 in rs1 is located.If rs1 is 0, then rd is equal to 32 or 64.

Operations:

```
 uint8 __ff0__ (uint rs1)
{
 XLEN=32;//for rv64, XLEN=64;
 if (rs1==0) return XLEN;
```

(continues on next page)

(continued from previous page)

```
 cnt=XLEN-1;
 while((rs1&0x8000_0000)) {rs1=rs1<<1;cnt--;}
 return cnt;
}

rs1=0x80001000, then rd=30
```

Exceptions:

illegal instruction exception

31.2.48 xl.bitrev

Format:

31 - 27	26 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
10110	uimm3[1:0]	uimm2[4:0]	rs1	001	rd	1111011	cust3

Syntax:

```
xl.bitrev rd, rs1, uimm3, uimm2

uimm3 == 0:Flip in groups of 1bit
uimm3 == 1:Flip in groups of 2bit
uimm3 == 2:Flip in groups of 3bit
uimm3 == 3:undefined

uimm2 Represents the number of bits shifted to the left of rs1 before flipping

xl.bitrev rd, rs1, 0, 0  revert bits in group of 1bit, with no shift.

IN   : 0x12345678 0001_0010 0011_0100 0101_0110 0111_1000
SHIFT: 0x12345678 0001_0010 0011_0100 0101_0110 0111_1000 //no shift
OUT  : 0x1e6a2c48 0001_1110 0110_1010 0010_1100 0100_1000


xl.bitrev rd, rs1, 1, 2  revert bits in group of 2bits, after left shift 2.

in:    0x12345678 0001_0010 0011_0100 0101_0110 0111_1000
shift: 0x48d159e0 0100_1000 1101_0001 0101_1001 1110_0000 //left shift 2
out:   0x0b654721 0000_1011 0110_0101 0100_0111 0010_0001


xl.bitrev rd, rs1, 2, 4  revert bits in group of 3bits, after left shift 4.

in:    0x12345678 0001_0010 0011_0100 0101_0110 0111_1000
shift: 0x23456780 0010_0011 0100_0101 0110_0111 1000_0000 //left shift 4
out:   0x08765432 0000_0100 1111_0010 1010_1001 1000_0001
```

Description:

Performs a bit flip operation on rs1 according to the specified pattern and group. uimm3 specifies the group information, and uimm2 specifies the number of left shifts before the reversal.

Operations:

```
uint __bitrev__ (uint rs1, uint8 grp_bits, uint8 lsft_bit);
```

Exceptions:

illegal instruction exception

31.2.49 xl.flh

Format:

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0101	imm[8:1]	rs1	101	rd(freg)	1111011	cust3

Syntax:

```
xl.flh rd, imm(rs1)
```

Description:

The instruction reads 2 bytes of data from the memory address specified by rs1, saves it to the floating-point register frd, and updates the base to (rs1+imm) and saves it to rs1. Semi-precision efficiency.

Operations:

```
frd = Mem16(rs1)
rs1 += imm
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.50 xl.flw

Format:

31 - 28	27 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0110	imm[9:2]	rs1	101	rd(reg)	1111011	cust3

Syntax:

```
xl.flw rd,imm(rs1)
```

Description:

The instruction reads 4 bytes of data from the memory address specified by rs1, saves it to the floating-point register frd, and updates the base to (rs1+imm) and saves it to rs1. Single precision efficiency.

Operations:

```
frd = Mem32(rs1)

rs1 +=imm
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.51 xl.fld

Format:

31 - 28	27 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
0111	imm[10:3]	rs1)	101	rd(freg)	1111011	cust3

Syntax:

```
xl.fld rd,imm(rs1)
```

Description:

The instruction reads 8 bytes of data from the memory address specified by rs1, saves it to the floating-point register frd, and updates the base to (rs1+imm) and saves it to rs1. Double precision efficiency.

Operations:

```
frd =Mem64(rs1)

rs1 +=imm
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.52 xl.fsh

Format:

31 - 28	27 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1101	imm[8:6]	rs2(freg)	rs1	101	imm[5:1]	1111011	cust3

Syntax:

```
xl.fsh rs2,imm(rs1)
```

Description:

The instruction stores 2 bytes of data in rs2 into the storage space at the specified address, while the base is updated to (rs1+imm) and saved to rs1.

Operations:

```
Mem16(rs1)= rs2

rs1 +=imm
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.53 xl.fsw

Format:

31 - 28	27 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1110	imm[9:7]	rs2(freg)	rs1	101	imm[6:2]	1111011	cust3

Syntax:

```
xl.fsw rs2,imm(rs1)
```

Description:

The instruction stores 4 bytes of data in rs2 into the storage space at the specified address, while the base is updated to (rs1+imm) and saved to rs1.

Operations:

```
Mem32(rs1)= rs2

rs1 +=imm
```

Exceptions:

Unaligned exception, illegal instruction exception

31.2.54 xl.fsd

Format:

31 - 28	27 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Type
1111	imm[10:8]	rs2(freg)	rs1	101	imm[7:3]	1111011	cust3

Syntax:

```
xl.fsd rs2,imm(rs1)
```

Description:

The instruction stores 8 bytes of data in rs2 into the storage space at the specified address, while the base is updated to (rs1+imm) and saved to rs1.

Operations:

```
Mem64(rs1)= rs2

rs1 +=imm
```

Exceptions:

Unaligned exception, illegal instruction exception

32 SMP and Cluster Cache

32.1 SMP and Cluster Cache Overview

Nuclei processor core can optionally support Cluster Cache (CC) and Symmetric Multi-Processor (SMP), in a Nuclei MP core design (like UX900 MP core), it default integrates the Cluster Cache (CC) and SMP related module called Snoop Control Unit (SCU). Following figure shows a Nuclei MP Cluster diagram:

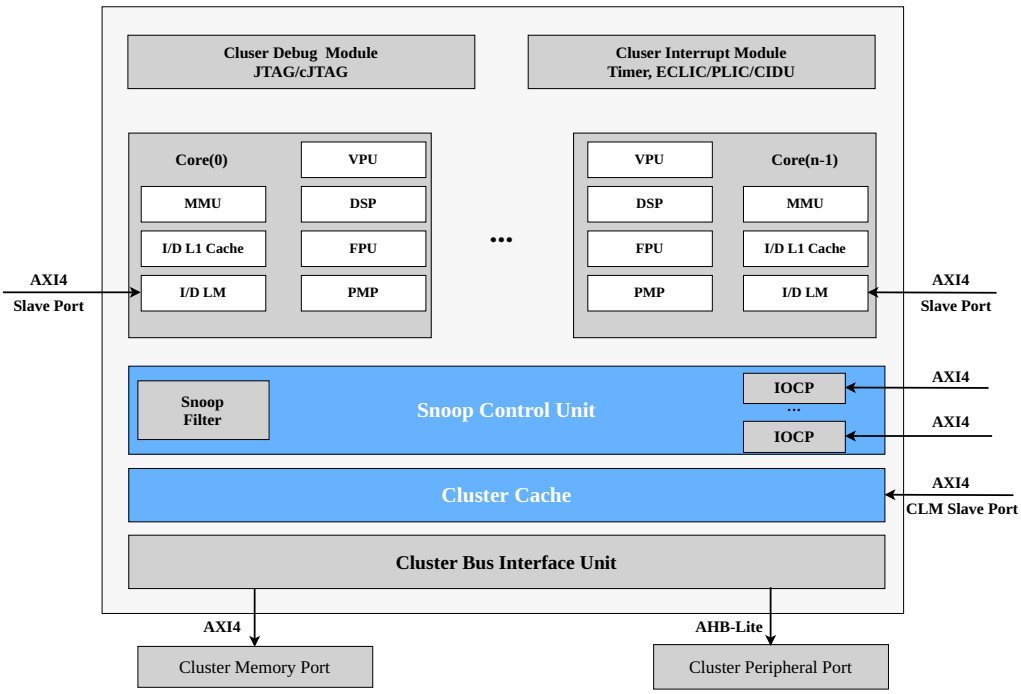


Fig. 32.1: Multi-Core Cluster Diagram

In the Multi-Core cluster design, all cores are symmetric and each core gets its private L1 instruction and data cache, I/D-LM and PMP/MMU/FPU/DSP/VPU module.

All cores share the same Debug Module, Interrupt Module, Cluster Bus Interface Unit and Cluster Cache, Cluster Cache is used to provide fast access to copies of external memory (usually DDR in the SOC) for masters in the cluster.

The I/O Coherency Port (IOCP) is slave port for the external masters which want to access the shareable and cacheable data of cores. And the SCU acts as a broadcast with filtering coherency manager for all the cores in the cluster and all external masters which hooks to IOCP.

The Cluster Cache can be configured to Cluster Local Memory (CLM) by user's software, all cores share the CLM and can access by the same address range and the same latency. The CLM Slave Port is slave port for external masters which want to access the CLM.

This chapter describes details of CC, SMP, SCU and related software visible registers, it also involves some basic Cache Control and Maintenance (CCM) operations.

32.2 Client Description

SCU support CORE and IOCP data coherency.

For simplicity, SCU refers to the components upstream that need to support data consistency as 'client'.

So:

- One CORE is a client.
- All IOCP is a client.

For example, there are four cores and two IOCP in cluster. So the client number is 5.

- Client0: CORE0
- Client1: CORE1
- Client2: CORE2
- Client3: CORE3
- Client4: IOCP0 and IOCP1

32.3 SMP and Cluster Cache Registers and Description

32.3.1 SMP and Cluster Cache Registers

Table 32.1: SMP and Cluster Cache Registers list

Offset	RW	Name	Description
0x0	MR	SMP_VER	Machine Mode SMP Version Register
0x4	MR	SMP_CFG	Machine Mode SMP Configuration Register
0x8	MR	CC_CFG	Machine Mode CC Config Register
0xC	MRW	SMP_ENB	Machine Mode SMP Enable Register
0x10	MRW	CC_CTRL	Machine Mode CC Control Register
0x14	MRW	CC_mCMD	Machine Mode CC Command and Status
0x18	MRW	CC_ERR_INJ	CC ECC Error Injection Control Register
0x1C	MRW	CC_RECV_CNT	CC ECC Recoverable Error Count
0x20	MRW	CC_FATAL_CNT	CC ECC Fatal Error Count
0x24	MRW	CC_RECV_THV	CC ECC Recoverable Error Threshold Value
0x28	MRW	CC_FATAL_THV	CC ECC Fatal Error Threshold Value
0x2C	MRW	CC_BUS_ERR_ADDR	CC Maintain Operate Bus Error Physical Address, it occupies 8 Bytes. CC_CTRL register can decide it can be accessed or not in S Mode.
0x40	MRW	CLIENT0_ERR_STATUS	Client0 of CC Error Status. CC_CTRL register can decide it can be accessed or not in S Mode.
…	…	…	…
0xBC	MRW	CLINT31_ERR_STATUS	Client31 of CC Error Status. CC_CTRL register can decide it can be accessed or not in S Mode.
0xC0	SRW	CC_sCMD	Supervisor Mode CC Command and Status. CC_CTRL register can decide it can be accessed or not in S Mode.

continues on next page

Table 32.1 – continued from previous page

Offset	RW	Name	Description
0xC4	URW	CC_uCMD	User Mode CC Command and Status. CC_CTRL register can decide it can be accessed or not in S/U Mode.
0xC8	MR	SNOOP_PENDING	Indicate the Core is being snooped or not in the SCU
0xCC	MR	TRANS_PENDING	Indicate the Core ‘s transaction finish or not in the SCU
0xD0	MRW	CLM_ADDR_BASE	Cluster Local Memory base address, this register occupies 8 Bytes
0xD8	MRW	CLM_WAY_EN	Cluster Cache way enable register for Cluster Local Memory
0xDC	MRW	CC_INVALID_ALL	Cluster Cache invalid all register. CC_CTRL register can decide it can be accessed or not in S Mode.
0xE0	MRW	STM_CTRL	Stream read/write control reigster.
0xE4	MRW	STM_CFG	Stream read/write configuration reigster.
0xE8	MRW	STM_TIMEOUT	Stream read/write timeout reigster.
0xEC	MRW	DFF_PROT	Hardware Register protect Enable register.
0xF0	MRW	ECC_ERR_MSK	Mask L2M ECC Error to ecc_cc_error_masked or safety_error output.
0x100	MRW	NS_RG0	Non-Sharable Memory Region 0, this register occupies 8 Bytes
…	…	…	…
0x178	MRW	NS_RG15	Non-Sharable Memory Region 15, this register occupies 8 Bytes
0x180	MRW	SMP_PMON_SEL0	Performance Monitor Event Selector 0. CC_CTRL register can decide it can be accessed or not in S Mode.
…	…	…	…
0x1BC	MRW	SMP_PMON_SEL15	Performance Monitor Event Selector 15. CC_CTRL register can decide it can be accessed or not in S/U Mode.
0x1C0	MRW	SMP_PMON_CNT0	Performance Monitor Event Counter 0, this register occupies 8 Bytes. CC_CTRL register can decide it can be accessed or not in S/U Mode.
…	…	…	…
0x23C	MRW	SMP_PMON_CNT15	Performance Monitor Event Counter 15, this register occupies 8 Bytes. CC_CTRL register can decide it can be accessed or not in S/U Mode.
0x280	MRW	CLIENT0_ERR_ADDR	The register of address of client0 which causes error, this register occupies 8 Bytes. CC_CTRL register can decide it can be accessed or not in S Mode.
…	…	…	…
0x378	MRW	CLIENT31_ERR_ADDR	The register of address of client31 which causes error, this register occupies 8 Bytes. CC_CTRL register can decide it can be accessed or not in S Mode.
0x380	MRW	CLIENT0_WAY_MASK	Cluster Cache way mask control register for client0.
0x384	MRW	CLIENT1_WAY_MASK	Cluster Cache way mask control register for client1.
…	…	…	…
0x3FC	MRW	CLIENT31_WAY_MASK	Cluster Cache way mask control register for client31
0x900	MRW	IOCP_PPI_REGION_EN	IOCP PPI region Enable register.
0x904	MRW	IOCP_CPPI_REGION_EN	IOCP CPPI region Enable register.
0x908	MRW	IOCP_DEV_REGION_L_BASE	IOCP Device region low base address register.
0x90C	MRW	IOCP_DEV_REGION_L_MASK	IOCP Device region low address mask register.

continues on next page

Table 32.1 – continued from previous page

Offset	RW	Name	Description
0x910	MRW	IOCP_DEV_REGION_H_BASE	IOCP Device region high base address register.
0x914	MRW	IOCP_DEV_REGION_H_MASK	IOCP Device region high address mask register.
0xA04	MRW	IOCP_NOC_REGION0_L_BASE	IOCP Non-Cacheable region0 low base address register.
0xA08	MRW	IOCP_DEV_REGION0_L_MASK	IOCP Non-Cacheable region0 low address mask register.
0xA0C	MRW	IOCP_NOC_REGION1_L_BASE	IOCP Non-Cacheable region0 low base address register.
0xA10	MRW	IOCP_NOC_REGION1_L_MASK	IOCP Non-Cacheable region0 low address mask register.
…	…	…	…
0xB04	MRW	IOCP_NOC_REGION0_H_BASE	IOCP Non-Cacheable region0 high base address register.
0xB08	MRW	IOCP_NOC_REGION0_H_MASK	IOCP Non-Cacheable region0 high address mask register.
0xB0C	MRW	IOCP_NOC_REGION0_H_BASE	IOCP Non-Cacheable region1 high base address register.
0xB10	MRW	IOCP_NOC_REGION0_H_MASK	IOCP Non-Cacheable region1 high address mask register.
…	…	…	…
0xC00	MRW	IOCP_DEV_MACRO_REGION_EN	IOCP device region entry enable register.
0xC04	MRW	IOCP_NOC_MACRO_REGION_EN	IOCP non-cacheable region entry enable register.
0xC08	MRW	IOCP_CACH_MACRO_REGION_EN	IOCP cacheable region entry enable register.

Note:

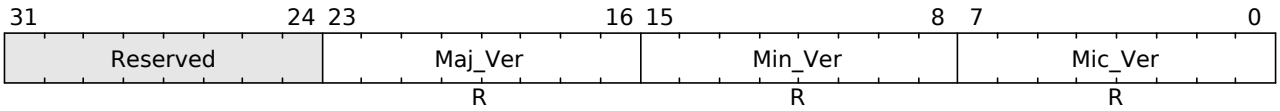
1. Snoop Control Unit and Cluster Cache module are connected with the Cores with internal bus and the accessing privilege info can be used directly by hardware to control the accessing permission described in this chapter.
2. In this chapter, the client means the each core in the cluster and all IOCP masters. When there are m cores and n IOCP masters. the core 0 is client 0 ,the core m is client (m-1), all IOCP masters share the client m.

32.3.2 SMP_VER

This register is used to show the microarchitecture implementation version of SMP related module.

Table 32.2: SMP_VER Register

Field Name	Bits	Reset Value	Description
Mic_Ver	7:0	X	Micro Version Number
Min_Ver	15:8	X	Minor Version Number
Maj_Ver	23:16	X	Major Version Number
Reserved	31:24	0	Reserved 0

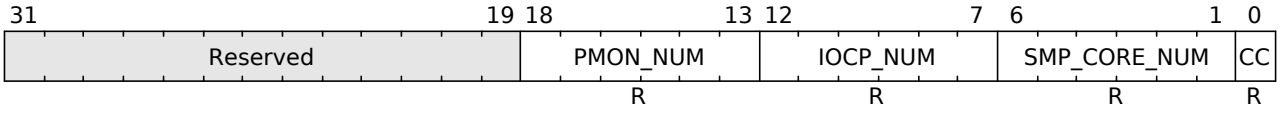


32.3.3 SMP_CFG

This register is used to show the hardware configuration of SMP related design.

Table 32.3: SMP_CFG Register

Field Name	Bits	Reset Value	Description
CC_PRESENT	0	X	Cluster Cache present or not. 0: not present; 1: Present
SMP_CORE_NUM	6:1	X	Indicate core number in the cluster, it is fixed value (core_num -1) when the RTL is generated.
IOCP_NUM	12:7	X	The IO Coherency port number in the cluster, it is fixed value when the RTL is generated.
PMON_NUM	18:13	X	The performance monitor number, it is fixed value when the RTL is generated.
Reserved	31:19	0	Reserved 0

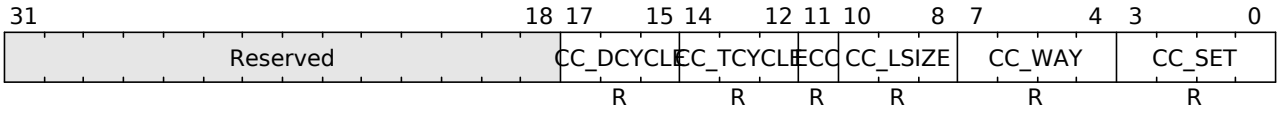


32.3.4 CC_CFG

This register is used to show the hardware configuration of Cluster Cache.

Table 32.4: CC_CFG Register

Field Name	Bits	Reset Value	Description
CC_SET	3:0	X	Cluster Cache Set Number = 2^(CC_SET)
CC_WAY	7:4	X	Cluster Cache Way Number = (CC_WAY + 1)
CC_LSIZE	10:8	X	Cluster Cache Line Size = 2^(CC_LSIZE + 2)
CC_ECC	11	X	Indicate the Cluster Cache supports ECC or not
CC_TCYCLE	14:12	X	Indicate the L2 Tag sram access cycle - 1. 0: 1-cycle 1: 2-cycle
CC_DCYCLE	17:15	X	Indicate the L2 Data sram access cycle - 1. 0: 1-cycle 1: 2-cycle
Reserved	31:18	0	Reserved 0



32.3.5 SMP_ENB

This register is used to control which cores or all IOCP in the cluster can be cache coherency with each other handled by hardware, or software needs to maintain the cache coherency when the corresponding bits not enabled. 1 means enable while 0 means disable. The number of effective bits in this register is number of cores in the cluster plus 1 if IOCP is configured (All IOCP share one bit), and the upper bit stands for all IOCP.

If the bit is 1, the coherency of (core i)'s shareable and cacheable access will be handled by SCU module, and the procedure is transparent for software or users. We recommend users to enable each bit when use Nuclei SMP design.

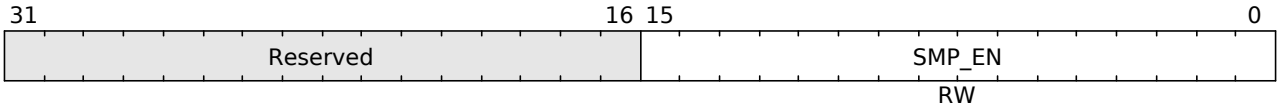
If the bit i is 0, no hardware is responsible for cache coherence between this core with other cores and external masters, so the memory space used by this core should be planned as non-sharable or non-cacheable region, if core does access the shareable and cacheable memory of other cores, then the data coherency is software's responsibility through Nuclei CCM (Cache Control and Management).

Note: If user wants to let some of cores in the cluster to share cacheable data, the program sequence is as below:

- Firstly enable Cluster Cache.
- Secondly enable Cluster Cache SMP for the cores in the cluster.
- Thirdly enable the related cores' L1 I/D Cache.

Table 32.5: SMP_ENB Register

Field Name	Bits	Reset Value	Description
SMP Enable	16:0	0	SMP Enable Bit for clients in the Cluster. Each bit control one clients. The low bit is CORE clients. high bit is IOCP clients.
Reserved	31:17	0	Reserved 0



32.3.6 CC_CTRL

This register is used to control Cluster Cache specific functions and get some status.

It can only be accessed in Machine Mode.

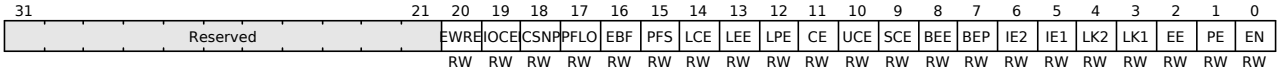
Table 32.6: CC_CTRL register

Field Name	Bits	Reset Value	Description
CC_EN (EN)	0	0	CC Enable Bit. 0: Disable, 1: Enable
CC_ECC_EN (PE)	1	1	CC ECC Enable Bit. 0: Disable, 1: Enable
ECC_EXCP_EN (EE)	2	0	CC ECC Exception Enable Bit. 0: Disable, 1: Enable
LOCK_ECC_CFG (LK1)	3	0	Lock the CC ECC Configuration Bit (CC_ECC_EN & LOCK_ECC_CFG). 0: Disable, 1: Enable. If this bit is set to 1, previous two bits and this bit can't be changed by software anymore, only reset can change to reset value 0.
LOCK_ECC_ERR_INJ (LK2)	4	0	Lock CC ECC Error Injection Register. 0: Disable, 1: Enable. If this bit is 1, then this bit and register CC_ERR_INJ can't be changed by software anymore, only reset can change to reset value 0.
RECV_ERR_IRQ_EN (IE1)	5	0	Enable the interrupt when recoverable error count exceeds the threshold. 0: Disable, 1: Enable
FATAL_ERR_IRQ_EN (IE2)	6	0	Enable the interrupt when fatal error count exceeds the threshold. 0: Disable, 1: Enable
BUS_ERR_PEND (BEP)	7	0	Indicate if there is Bus Error Pending of all type including CC maintain operation, copy-back of each client, refill bus error, shadow tag ecc error, snoop error, L2 ECC error, bus error of write early respond outstanding. 0: No Pending, 1: Pending.

continues on next page

Table 32.6 – continued from previous page

Field Name	Bits	Reset Value	Description
BUS_ERR_IRQ_EN (BEE)	8	0	Enable the Bus Error interrupt of CC maintain operation. 0: Disable, 1: Enable
SUP_CMD_EN (SCE)	9	0	Enable S Mode can operate register CC_sCMD and SMP_PMON_SEL. 0: Disable, 1: Enable
USE_CMD_EN (UCE)	10	0	Enable U Mode can operate register CC_uCMD and SMP_PMON_SEL. 0: Disable, 1: Enable
ECC_CHK_EN (CE)	11	0	CC ECC Check Enable Bit. 0: Disable, 1: Enable
CLM_ECC_EN (LPE)	12	1	CLM ECC Enable Bit. 0: Disable, 1: Enable
CLM_EXCP_EN (LEE)	13	0	CLM ECC Exception Enable Bit. 0: Disable, 1: Enable
CLM_ECC_CHK_EN (LCE)	14	0	CLM ECC Check Enable Bit. 0: Disable, 1: Enable
PF_SH_CL_EN (PFS)	15	0	Prefetch shared cachelines Enable. 0: Disable, 1: Enable
PF_L2_EARLY_EN (EBF)	16	1	Prefetch L2 to biu early Enable. 0: Disable, 1: Enable
PF_BIU_OUTS_EN (PFL0)	17	0	Prefetch limit BIU outstanding. 0: Disable, 1: Enable
I_SNOOP_D_EN (ICSNP)	18	0	Snoop to dcache for icache refill reads Enable. 0: Disable, 1: Enable
IOCC_ERR(IOCE)	19	0	IOCC Has error. 0: No error, 1: Error
EARLY_WR_ERR(EWRE)	20	0	Early write response has error 0: No error, 1: Error
Reserved	31:21	0	Reserved 0



32.3.7 CC_mCMD

Machine Mode CC Maintain Command and Status Register.

This register is used to set specific maintain command for Cluster Cache and check the command result in Machine Mode. When valid command is specified, the field of Complete is clear to 0 in next cycle and the command is executed, and filed of Complete will set to 1 until the command is finished. If the WB command cause Bus Error during executing, then WB is aborted and the Complete filed is set to 1 and update the filed of Result_Code and CC_BUS_ERR_ADDR register to the error address. If multiple CC maintain commands have caused Bus Error, CC_BUS_ERR_ADDR register records the latest one.

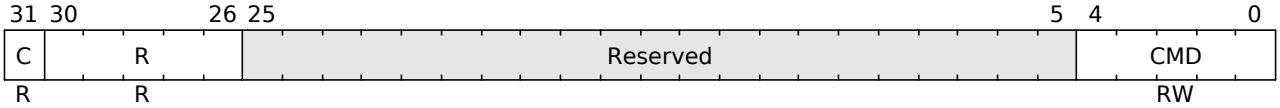
Table 32.7: CC_mCMD Register

Field Name	Bits	Reset Value	Description
CMD	4:0	0	Cluster Cache Maintain Command Code
Reserved	22:5	0	Reserved 0
Recoverable ECC Err IRQ Status Clear (RESC)	23	0	Read as Recoverable ECC Error IRQ Pending, write 1 will clear CC_RECV_CNT.
Fatal ECC Err IRQ Status Clear (FESC)	24	0	Read as Fatal ECC Error IRQ Pending, write 1 will clear CC_FATAL_CNT.
Bus Err Status Clear (BESC)	25	0	Read as BUS_ERR_PEND, write 1 will clear BUS_ERR_PEND.
Result_Code (R)	30:26	0	Result of the CMD

continues on next page

Table 32.7 – continued from previous page

Field Name	Bits	Reset Value	Description
Complete (C)	31	0	Indicate the CMD complete or not. 0: Not Complete, 1: Complete



CMD Types list in following table:

Table 32.8: CMD Types

Operation	Codes	Description
WB_ALL	5'b00_111	Flush all the valid and dirty cachelines, Lock bit is not affected.
WBINVAL_ALL	5'b00_110	Unlock and Flush and invalid all the valid and dirty cachelines.

Result Code information list in following table:

Table 32.9: Result Code

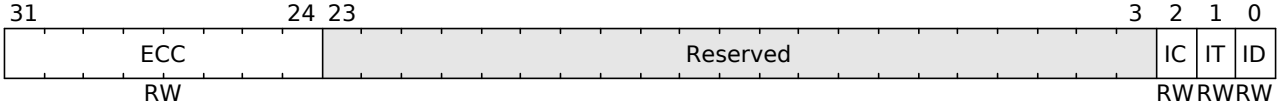
Code	Fail Info
0	Command Succeed
1	Exceed the Upper entry Num of Lockable way (N-Way Cluster Cache, Lockable is N-1)
3	Refill has Bus Error
4	ECC Error
5	Copy-Back has Bus Error
6 - 31	Invalid Command

32.3.8 CC_ERR_INJ

ECC Error Injection Control Register

Table 32.10: CC_ERR_INJ Register

Field Name	Bits	Reset Value	Description
INJ_DATA (ID)	0	0	Inject error to CC Data Ram
INJ_TAG (IT)	1	0	Inject error to CC Tag Ram
INJ_CLM (IC)	2	0	Inject error to CLM Ram
Reserved	23:3	0	Reserved
INT_ECC_CODE (ECC)	31:24	0	The content which will be injected



User can inject the ECC Code to Cluster Cache Data Ram or Tag Ram when lock the Cluster Cache Line, the steps are:

- Use the L1 D-Cache WBINVAL command to flush the test address's cacheline from L1 D-Cache and Cluster Cache, for the detail of WBINVAL command, please refer chapter CCM Mechanism.
- Set the CC_ERR_INJ register.
- Use the Cluster Cache LOCK command to load the test address memory content into Cluster Cache's cacheline (the ECC Error Code will be injected to Cluster Cache Data ram and Tag RAM).
- Check the ccm_data CSR to know if the LOCK command succeeds or not.

- Read the test address (the L1 D-Cache will read the content of Cluster Cache, the Cluster Cache ECC Error will happen at this point).
- Check the ECC Error happen or not by ECC Exception or other methods.

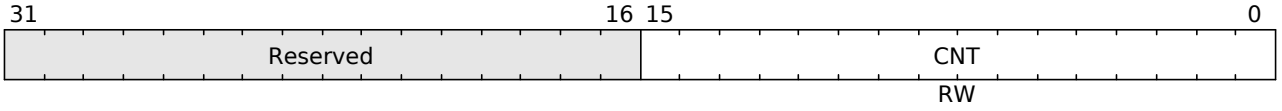
32.3.9 CC_RECV_CNT

ECC Recoverable Error Count Register

This register is used to count the Cluster Cache ECC recoverable error events, it only begins to count when the ECC function of Cluster Cache enables, and it is a saturated counter.

Table 32.11: ECC_RECV_CNT Register

Field Name	Bits	Reset Value	Description
CNT	15:0	0	Count the recoverable error, it is saturated
Reserved	31:16	0	Reserved 0



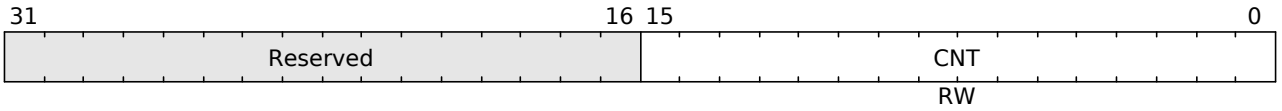
32.3.10 CC_FATAL_CNT

ECC Fatal Error Count Register.

This register is used to count the Cluster Cache ECC fatal error events, it only begins to count when the ECC function of Cluster Cache enables, and it is a saturated counter.

Table 32.12: ECC_FATAL_CNT Register

Field Name	Bits	Reset Value	Description
CNT	15:0	0	Count the fatal error, it is saturated
Reserved	31:16	0	Reserved 0



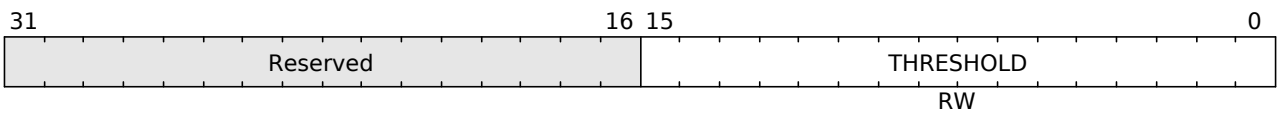
32.3.11 CC_RECV_THV

ECC Recoverable Error Threshold Register.

This register is used to set the threshold value of Cluster Cache ECC recoverable errors.

Table 32.13: ECC_RECV_THV Register

Field Name	Bits	Reset Value	Description
THRESHOLD	15:0	0	The threshold value of ECC recoverable error, if the ECC_RECV_CNT value is equal or greater than the ECC_RECV_THV value, it will trigger the related interrupt.
Reserved	31:16	0	Reserved 0



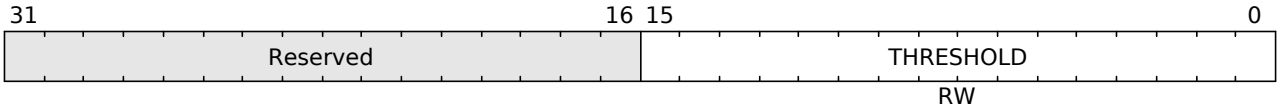
32.3.12 CC_FATAL_THV

ECC Fatal Error Threshold Register

This register is used to set the threshold value of Cluster Cache ECC fatal error.

Table 32.14: ECC_FATAL_THV Register

Field Name	Bits	Reset Value	Description
THRESHOLD	15:0	0	The threshold value of ECC fatal error, if the ECC_FATAL_CNT value is equal or greater than the ECC_FATAL_THV value, it will trigger the related interrupt.
Reserved	31:16	0	Reserved 0



32.3.13 CC_BUS_ERR_ADDR

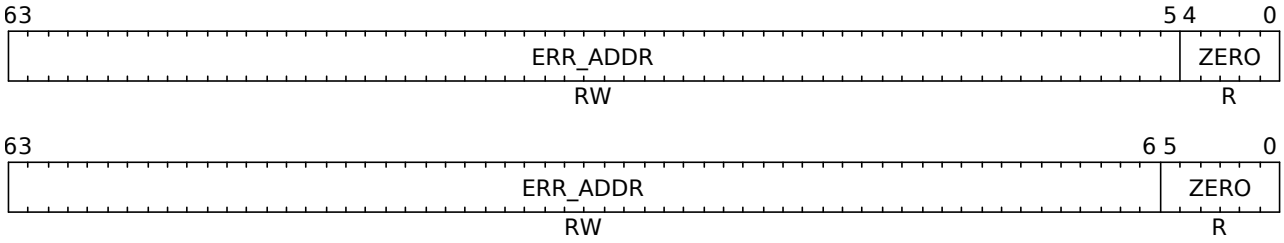
CC Bus Error Address Register

This register is used to record the Physical Address when user sets the address for CC Management Operation and then it causes the Bus Error.

Note: When multi bus errors happens, it only logs the first. And it only logs the cache line's begin address, the offset inside the cache line is ignored.

Table 32.15: CC_Bus_ERR_ADDR Register

Field Name	Bits	Reset Value	Description
CONST_0	5/4:0	Constant 0.	When cache line is 32B, the bits of this field is 4:0. When cache line is 64B, it is 5:0.
ERR_ADDR	PA_SIZE:6/5	0	The error address when Bus Error happens.



32.3.14 CLIENT(n)_ERR_STATUS

Client Error Status Register for client(n).

This register is used to record the detail error type for the client.

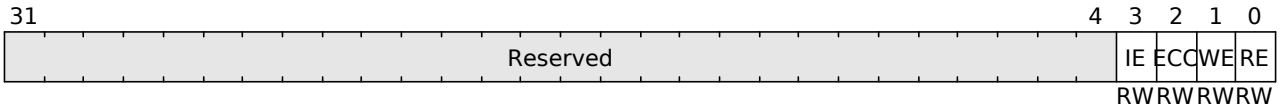
Table 32.16: CLIENT_ERR_STATUS Register

Field Name	Bits	Reset Value	Description
READ_BUS_ERR (RE)	0	0	The error type is read bus error.
WRITE_BUS_ERR (WE)	1	0	The error type is write bus error.

continues on next page

Table 32.16 – continued from previous page

Field Name	Bits	Reset Value	Description
CC_SCU_ECC_ERR (ECC)	2	0	The error type is cluster cache tag/data , SCU shadow tag or snoop L1 access ECC error.
IOCP_BUS_ERR (IE)	3	0	The error type is IOCP Bus response error. It only applies to clients hooked to IOCP Ports.
Reserved	31:16	0	Reserved 0



Note: When an error happens for the client, the register's corresponding field is set to 1, and the field can be only written to 0 to clear. Only when the register is 0, it can log a new error and the the corresponding address register will log the address.

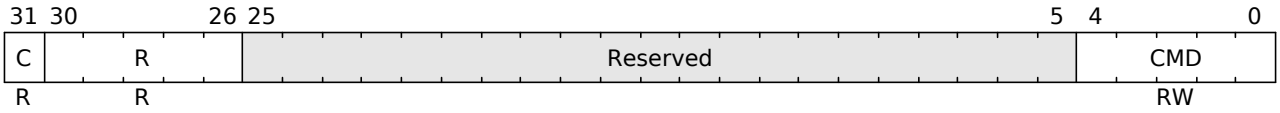
32.3.15 CC_sCMD

Supervisor Mode CC Maintain Command and Status Register.

This register is used to set specific maintain command for Cluster Cache and check the command result in Supervisor Mode.

Table 32.17: CC_sCMD Register

Field Name	Bits	Reset Value	Description
CMD	4:0	0	Cluster Cache Maintain Command Code
Reserved	25:5	0	Reserved 0
RESULT_CODE	30:26	0	Result of the CMD
Complete	31	0	Indicate the CMD complete or not. 0: Not Complete, 1: Complete



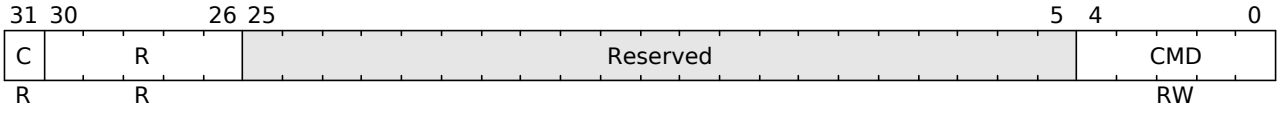
32.3.16 CC_uCMD

User Mode CC Maintain Command and Status Register.

This register is used to set specific maintain command for Cluster Cache and check the command result in User Mode.

Table 32.18: CC_uCMD Register

Field Name	Bits	Reset Value	Description
CMD	4:0	0	Cluster Cache Maintain Command Code
Reserved	25:5	0	Reserved 0
RESULT_CODE	30:26	0	Result of the CMD
Complete	31	0	Indicate the CMD complete or not. 0: Not Complete, 1: Complete



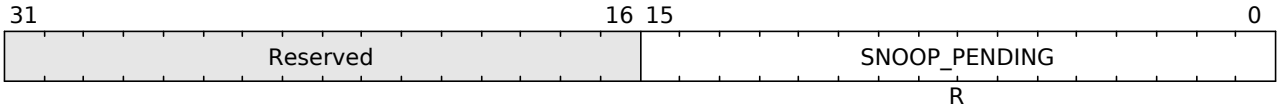
32.3.17 SNOOP_PENDING

Core Snoop Pending Register.

This register is used to check if a core's private L1 D-Cache is snooped by other clients or not.

Table 32.19: SNOOP_PENDING Register

Field Name	Bits	Reset Value	Description
SNOOP_PENDING	15:0	0	Snoop pending bit for each client in the cluster. User can query this register's related bit to know the snooping is finished or not when user want to disable the specific core's smp_en bit.
Reserved	31:16	0	Reserved 0



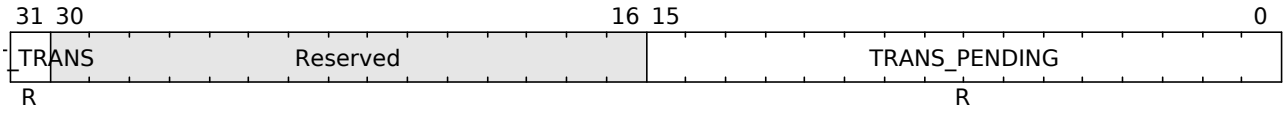
32.3.18 TRANS_PENDING

Core TRANS Pending Register.

This register is used to check if a core still has load/store transactions related SCU module or not.

Table 32.20: TRANS_PENDING Register

Field Name	Bits	Reset Value	Description
TRANS_PENDING	15:0	0	Transaction pending bit for each core in the cluster. User can query this register's related bit to know the core finishes all transaction related SCU or not when user want to use Cluster Cache WB_ALL or WBINVAL_ALL command.
Reserved	30:16	0	Reserved 0
EXT_TRANS	31	0	External Memory Bus Transaction pending bit.



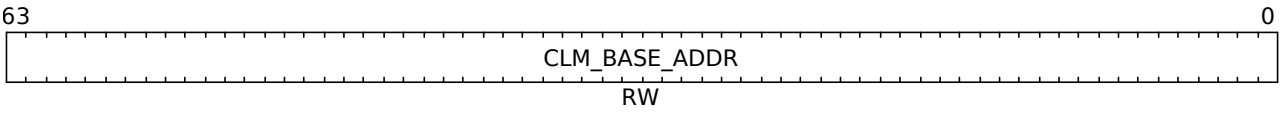
32.3.19 CLM_ADDR_BASE

Cluster LM Address Base Register.

This register is to set the base address of CLM. It is aligned to Cluster Cache size.

Table 32.21: CLM_ADDR_BASE Register

Field Name	Bits	Reset Value	Description
ADDR	PA_SIZE:0	RTL Input	The base address of CLM. Default value is from RTL input value.



Note: the PMA of CLM follows the Device/Non-Cacheable/Cacheable Region partition. If the CLM is in cacheable

region and user want to access the CLM with cache , it should enable the Cluster Cache first and enable the corresponding L1 ICache/DCache.

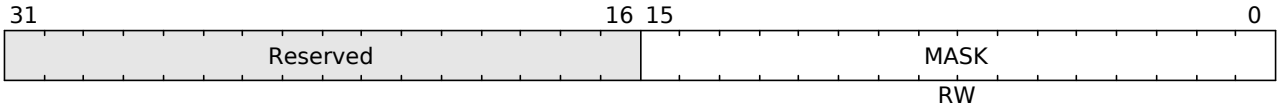
32.3.20 CLM_WAY_EN

Cluster LM Way Enable Register.

This register is to set the data ram of Cluster Cache to be used as Cluster Local Memory (CLM). When bit n is 1, it means the way n of Cluster Cache is used as Cluster Local Memory. If user has set m ways to CLM, then the CLM address is [Cluster_LM_ADDR_BASE ~ (Cluster_LM_ADDR_BASE + Way_Size*m)] , Way_Size is CC_Size/Num_Way.

Table 32.22: CLM_WAY_EN Register

Field Name	Bits	Reset Value	Description
ENA	15:0	RTL Input	This way is used as CLM or not. Default value is from RTL input value.
Reserved	31:16	0	Reserved 0.



The recommended procedure for using the CLM (Cluster Local Memory) is as follows:

- Use the register CLM_Base_Addr to set the CLM Base address or the default value is (PA - Cluster Cache size).
- Use the register CLM_WAY_EN to set the size of CLM (we recommend that user should enable continuous ways, so the CLM is one region), or the default value is 0x7fff.
- Use the register CC_CTRL to enable the Cluster Cache.
- Then user can use CLM normally with its base address and size.
- If user wants to change the CLM to Cache, please set the register CLM_WAY_EN to cut the CLM or disable CLM. Then more SRAM can be used as Cluster Cache.
- If user wants to to change the Cache to CLM, please use the register CLM_CTRL to disable the Cluster Cache firstly, and reset the CLM_WAY_EN (also CLM_Base_Addr if want to) to get more size for CLM , finally enable the Cluster Cache.

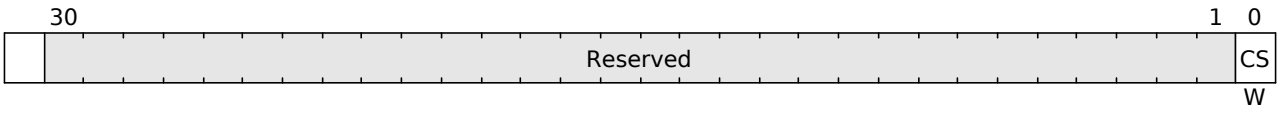
32.3.21 CC_INVALID_ALL

Cluster Cache Invalid All Register.

This register is used as invalid all Cluster Cache, and if some ways are set as CLM, setting of this register does not affect the CLM.

Table 32.23: CLUSTER_CACHE_INVAL_ALL Register

Field Name	Bits	Reset Value	Description
CS	0	0	Write 1 will invalid all Cluster Cache, and when the operation is done, the hardware will clear this bit to 0 automatically.
Reserved	31:1	0	Reserved 0.

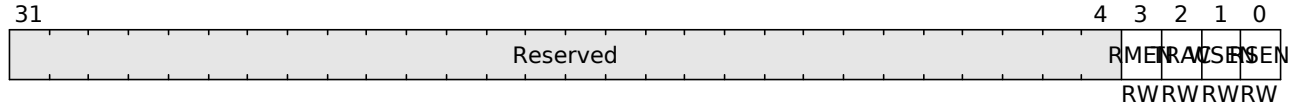


32.3.22 STM_CTRL

Stream read/write control register.

Table 32.24: STM_CTRL Register

Field Name	Bits	Reset Value	Description
RD_STM_EN(RSEN)	0	0	Read stream Enable.
WR_STM_EN(WSEN)	1	0	Write stream Enable.
TRANS_ALLOC(TRAC)	2	0	Translate alloc attribute to non-alloc attribute Enable.
RD_MERGE_EN(RMEN)	3	0	Non-cachebale attribute write merge Enable.
Reserve	31:4	0	Reserved 0.

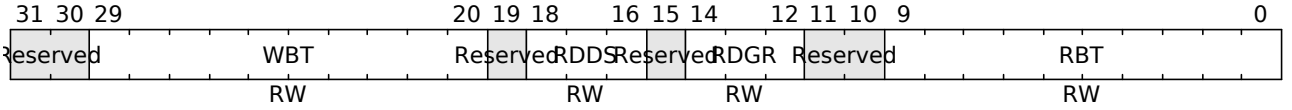


32.3.23 STM_CFG

Stream read/write configuration register.

Table 32.25: STM_CFG Register

Field Name	Bits	Reset Value	Description
RD_BYTE_THRE(RBT)	9:0	0	The prefetch number for Read stream.
Reserved	11:10	4	Reserved 0.
RD_DEGREE(RDGR)	14:12	4	The delta between prefetch address and current bus address.
Reserved	15	0	Reserved 0.
RD_DISTANCE(RDDS)	18:16	4	The threshold bytes number matching writestream training successfully.
Reserved	19	0	Reserved 0.
WR_BYTE_THRE(WBT)	29:20	0x20	The linebuff timeout free time when no same cacheline transaction.
Reserve	31:30	0	Reserved 0.

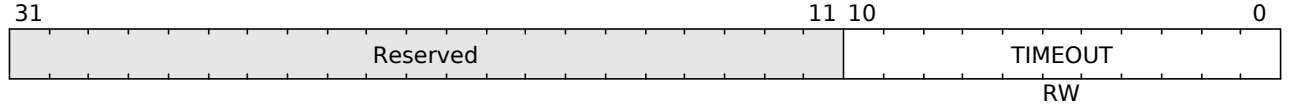


32.3.24 STM_TIMEOUT

Stream timeout register.

Table 32.26: STM_TIMEOUT Register

Field Name	Bits	Reset Value	Description
TIMEOUT	10:0	0	write streaming wait clk num
Reserved	31:11	4	Reserved 0.

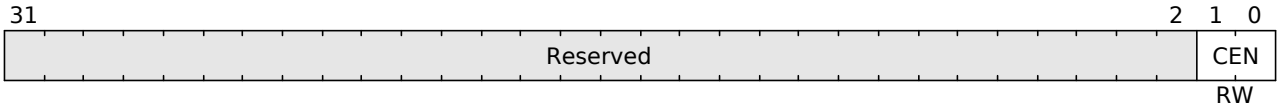


32.3.25 DFF_PROT

Hardware register protect enable register.

Table 32.27: DFF_PROT Register

Field Name	Bits	Reset Value	Description
chk_en(CEN)	1:0	1	Register protect check enable. 2'b01: Disable 2'b10: Enable Other: illegal value.
Reserved	31:2	0	Reserved 0.

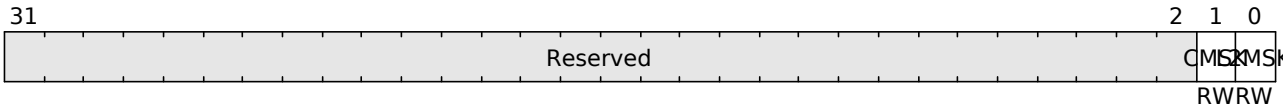


32.3.26 ECC_ERR_MSK

Mask L2M ECC Error to ecc_cc_error_masked or safety_error output.

Table 32.28: ECC_ERR_MSK Register

Field Name	Bits	Reset Value	Description
CC_L2_ERR_MSK(L2MSK)	0	1	Mask L2 double bit error output. 0: Not mask 1: Mask
CC_CORE_ERR_MASK(CMSK)	1	1	Mask Core double bit error output. 0: Not mask 1: Mask
Reserved	31:2	0	Reserved 0.



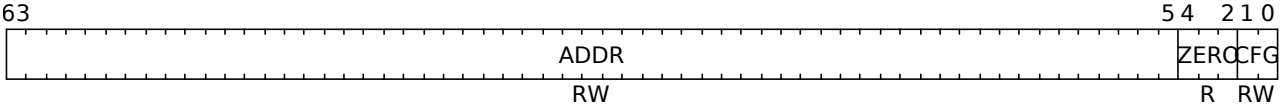
32.3.27 NS_RG(n)

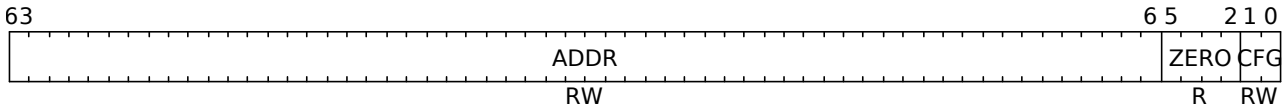
Non-Sharable Region Register

This register is to set a non-shareable region. It always gets 16 Non-Shareable Region registers; each register is 8 bytes. The ADDR filed is naturally aligned power-of-2 (NAPOT) just like RISC-V PMP address coding, please find details of NAPOT in <RISC-V Privileged Spec>.

Table 32.29: NS_RGX Register

Field Name	Bits	Reset Value	Description
CFG	1:0	0	0: Disable this region; 2:NACL; 3:NAPOT.
CONSTANT_ZERO	5/4:2	0	If cacheline is 32B, the width of Constant 0 is 3 (4:2); if cacheline is 64B, the width is 4 (5:2). Also the granularity of the region equals to cache-line size.
ADDR	PA_SIZE:6/5	0	Coding of address and space of the region





Addr	CFG	Match type and size
yyyy…yyyy	NACL	CL-byte NAPOT range (CL is cacheline)
yyyy…yyy0	NAPOT	CLx2-byte NAPOT range
yyyy…yy01	NAPOT	CLx4-byte NAPOT range
yyyy…y011	NAPOT	CLx8-byte NAPOT range
…	…	…
yy01…1111	NAPOT	2^(PA_SIZE-2)-byte NAPOT range
y011…1111	NAPOT	2^(PA_SIZE-1)-byte NAPOT range
0111…1111	NAPOT	2^(PA_SIZE)-byte NAPOT range
1111…1111	NAPOT	2^(PA_SIZE+1)-byte NAPOT range

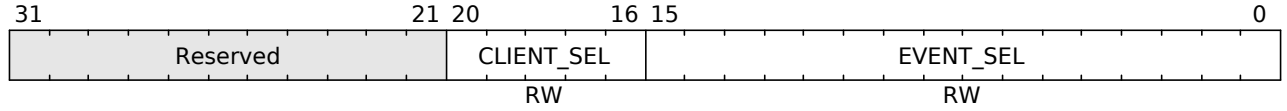
32.3.28 SMP_PMON_SEL(n)

Performance Monitor Event Selector Register

This register is to select Cluster Cache micro-architecture event to monitor the performance.

Table 32.31: Performance Monitor Event Selector Register

Field Name	Bits	Reset Value	Description
EVENT_SEL	15:0	0	Select the event for this Performance Monitor Counter
CLIENT_SEL	20:16	0	Specify the core in the cluster or external master number hooked to I/O Coherency Port
Reserved	31:21	0	Reserved 0



The EVENT_SEL coding is:

Table 32.32: Cluster Cache Performance Monitor Events

Value	Event Name
0	Disable monitor
1	Data Read Count
2	Data Write Count
3	Instruction Read Count
4	Data Read CC Hit Count
5	CC Replace Count for Data Write
6	CC Replace Count for Data Read
7	Data Read CC Miss Count
8	Instruction Read CC Hit Count
9	Instruction Read CC Miss Count
10	CC Replace Count for Instruction Read

The CLIENT_SEL value is from 0 and the core number or external master number is fixed for a Nuclei delivered Core IP design. For example, there is a four cores cluster with 4 I/O coherency ports and user hook 4 master devices to each coherency port, then 0 means core 0 in the cluster, 3 means core 3 in the cluster, 4 means master hooked to I/O coherency port 0, 7 means master hooked to port 3.

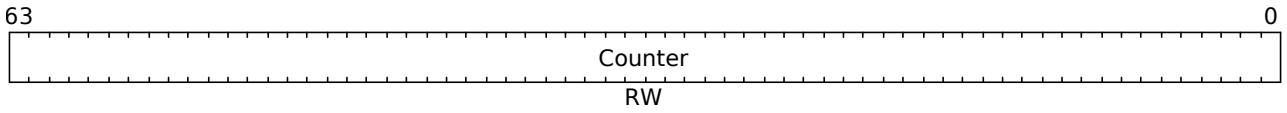
32.3.29 SMP_PMON_CNT(n)

Performance Monitor Event Counter Register.

This register is to count the specific event as the corresponding selector register selects.

Table 32.33: Performance Monitor Event Counter Register

Field Name	Bits	Reset Value	Description
CNT	63:0	0	If the event selected happens once, the value adds 1. If it overflows, hardware will set the Overflow bit in corresponding Performance Event Selector Register and let this value begin to count from 0.



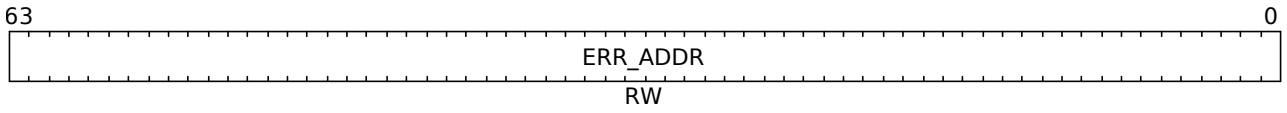
32.3.30 CLIENT(n)_ERR_ADDR

Client Error Address Register.

This register is used to record the detail physical address for the client when error happens.

Table 32.34: CLIENT_ERR_ADDR Register

Field Name	Bits	Reset Value	Description
ERR_ADDR	PA_SIZE:0	0	The error address when Error happens for the client. It only can be updated when corresponding Error Status register is 0.



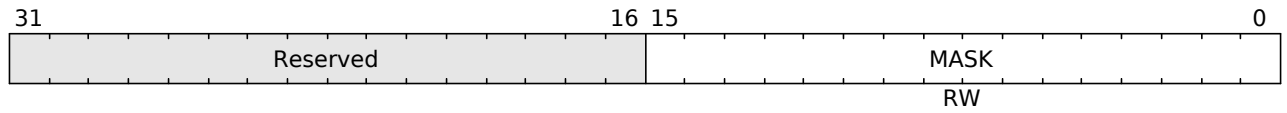
32.3.31 CLIENT(n)_WAY_MASK

Client Way Mask Register.

This register is to set the Cluster Cache ways which can be used by the client when it writes allocate to Cluster Cache. If the bit n is 1, then this way can't be used by this client.

Table 32.35: CLIENT_WAY_MASK Register

Field Name	Bits	Reset Value	Description
MASK	15:0	0	Mask this way for the client.
Reserved	31:16	0	Reserved 0.



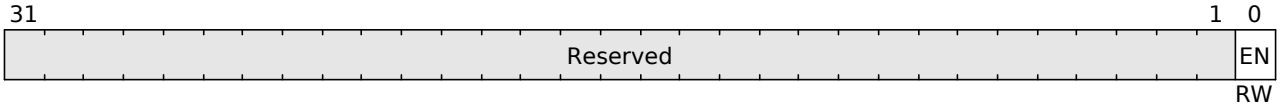
32.3.32 IOCP_PPI_REGION_EN

PPI region enable register.

This register is to enable ppi region as device attribute for IOCP master.

Table 32.36: IOCP_PPI_REGION Register

Field Name	Bits	Reset Value	Description
EN	0	0	Enable
Reserved	31:1	0	Reserved 0.



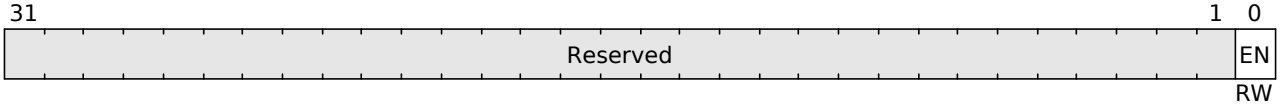
32.3.33 IOCP_CPPI_REGION_EN

CPPI region enable register.

This register is to enable cppi region as device attribute for IOCP master.

Table 32.37: IOCP_CPPI_REGION Register

Field Name	Bits	Reset Value	Description
EN	0	0	Enable
Reserved	31:1	0	Reserved 0.

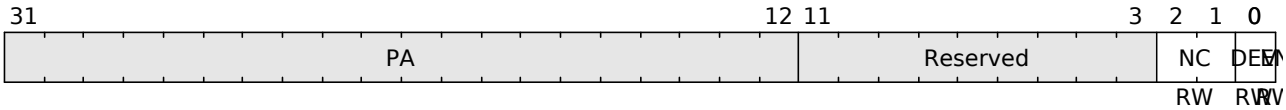


32.3.34 IOCP_DEV_REGION_L_BASE

This register is to define the low 32bit device attribute memory region address for IOCP master.

Table 32.38: IOCP_DEV_REGION_L_BASE Register

Field Name	Bits	Reset Value	Description
EN	0	0	This register enable.
DEV	1	0	If 1, this region attribute is device.
NC	2	0	If 1, this region attribute is non-cacheable.
Reserved	11:3	0	Reserved 0.
PA	31:12	0	The base address.

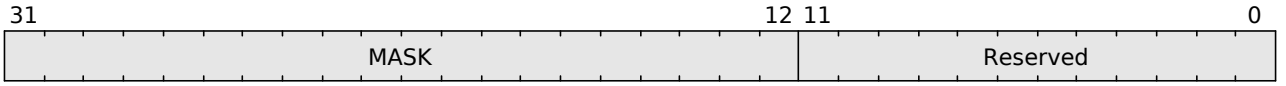


32.3.35 IOCP_DEV_REGION_L_MASK

This register is to define the low 32 bit device attribute memory region address mask for IOCP master.

Table 32.39: IOCP_DEV_REGION_L_MASK Register

Field Name	Bits	Reset Value	Description
Reserved	11:0	0	Reserved 0.
MASK	31:12	0	The address mask.



32.3.36 IOCP_DEV_REGION_H_BASE

This register is to define the high 32 bit device attribute memory region base address for IOCP master.

If PA_SIZE is less than 32, this register read 0 , write ignore.

32.3.37 IOCP_DEV_REGION_H_MASK

This register is to define the high 32 bit device attribute memory region address mask for IOCP master.

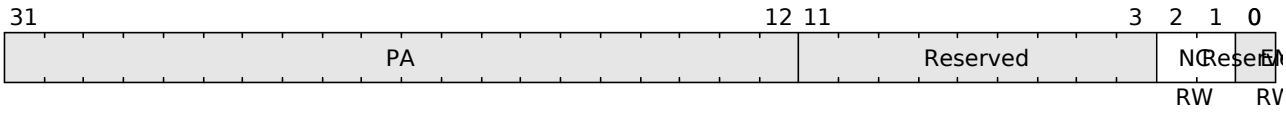
If PA_SIZE is less than 32, this register read 0 , write ignore.

32.3.38 IOCP_NOC_REGION(n)_L_BASE

This register is to define the low 32bit non-cacheable attribute memory region address for IOCP master.

Table 32.40: IOCP_NOC_REGION_L_BASE Register

Field Name	Bits	Reset Value	Description
EN	0	0	This register enable.
Reserved	1	0	Reserved 0.
NC	2	0	If 1, this region attribute is non-cacheable.
Reserved	11:3	0	Reserved 0.
PA	31:12	0	The base address.

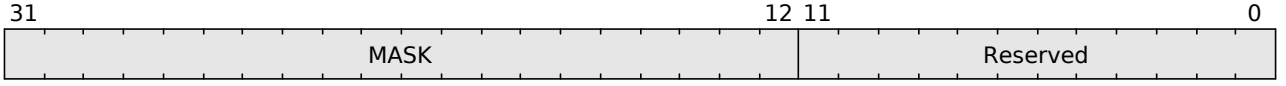


32.3.39 IOCP_NOC_REGION(n)_L_MASK

This register is to define the low 32 bit non-cacheable attribute memory region address mask for IOCP master.

Table 32.41: IOCP_NOC_REGION_L_MASK Register

Field Name	Bits	Reset Value	Description
Reserved	11:0	0	Reserved 0.
MASK	31:12	0	The address mask.



32.3.40 IOCP_NOC_REGION(n)_H_BASE

This register is to define the high 32 bit non-cacheable attribute memory region base address for IOCP master.

If PA_SIZE is less than 32, this register read 0 , write ignore.

32.3.41 IOCP_NOC_REGION(n)_H_MASK

This register is to define the high 32 bit non-cacheable attribute memory region address mask for IOCP master.

If PA_SIZE is less than 32, this register read 0 , write ignore.

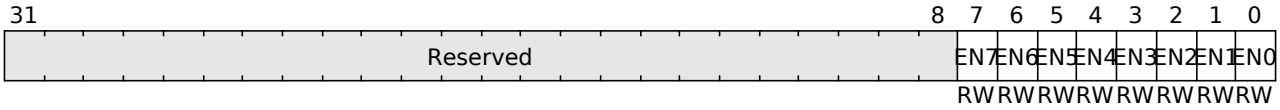
32.3.42 IOCP_DEV_MACRO_REGION_EN

Hardware CFG_DEVICE_REGION entry enable register.

This register is to enable device region entry as device attribute for IOCP master.

Table 32.42: IOCP_DEV_MACRO_REGION_EN Register

Field Name	Bits	Reset Value	Description
ENTRY0_EN(EN0)	0	1	Entry 0 enable.
ENTRY1_EN(EN1)	1	1	Entry 1 enable.
ENTRY2_EN(EN2)	2	1	Entry 2 enable.
ENTRY3_EN(EN3)	3	1	Entry 3 enable.
ENTRY4_EN(EN4)	4	1	Entry 4 enable.
ENTRY5_EN(EN5)	5	1	Entry 5 enable.
ENTRY5_EN(EN6)	6	1	Entry 6 enable.
ENTRY7_EN(EN7)	7	1	Entry 7 enable.
Reserved	31:8	0	Reserved 0.



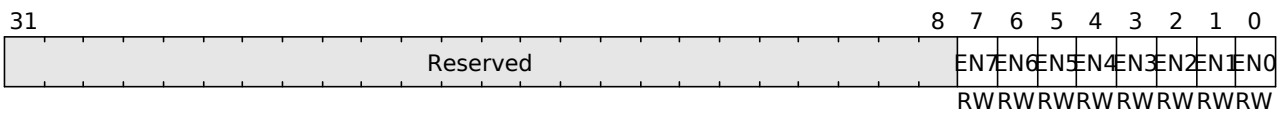
32.3.43 IOCP_NOC_MACRO_REGION_EN

Hardware CFG_NC_REGION entry enable register.

This register is to enable non-cacheable region entry as non-cacheable attribute for IOCP master.

Table 32.43: IOCP_NOC_MACRO_REGION_EN Register

Field Name	Bits	Reset Value	Description
ENTRY0_EN(EN0)	0	1	Entry 0 enable.
ENTRY1_EN(EN1)	1	1	Entry 1 enable.
ENTRY2_EN(EN2)	2	1	Entry 2 enable.
ENTRY3_EN(EN3)	3	1	Entry 3 enable.
ENTRY4_EN(EN4)	4	1	Entry 4 enable.
ENTRY5_EN(EN5)	5	1	Entry 5 enable.
ENTRY5_EN(EN6)	6	1	Entry 6 enable.
ENTRY7_EN(EN7)	7	1	Entry 7 enable.
Reserved	31:8	0	Reserved 0.



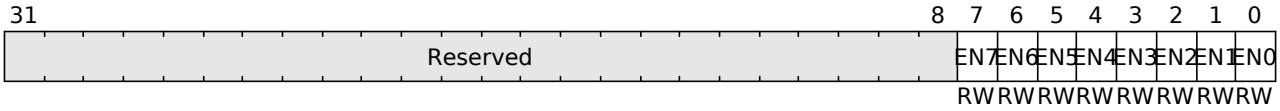
32.3.44 IOCP_CACH_MACRO_REGION_EN

Hardware CFG_CACHEABLE_REGION entry enable register.

This register is to enable cacheable region entry as cacheable attribute for IOCP master.

Table 32.44: IOCP_CACH_MACRO_REGION_EN Register

Field Name	Bits	Reset Value	Description
ENTRY0_EN(EN0)	0	1	Entry 0 enable.
ENTRY1_EN(EN1)	1	1	Entry 1 enable.
ENTRY2_EN(EN2)	2	1	Entry 2 enable.
ENTRY3_EN(EN3)	3	1	Entry 3 enable.
ENTRY4_EN(EN4)	4	1	Entry 4 enable.
ENTRY5_EN(EN5)	5	1	Entry 5 enable.
ENTRY5_EN(EN6)	6	1	Entry 6 enable.
ENTRY7_EN(EN7)	7	1	Entry 7 enable.
Reserved	31:8	0	Reserved 0.



32.4 SMP and Cluster Cache Error Handling

Previous section introduces that Cluster Cache may report Bus Error or ECC Error, to easy customer handle these errors, the SMP and CC module gathers all the Bus Error and ECC Errors events and report the events as an **Internal Interrupt** to all cores (please refer to *Interrupt Type* (page 24) and *Internal Interrupt* (page 25) to review the concept) , and the **Interrupt ID is fixed to 17**.

33

Riscv Smepmp Extension

PMP Enhancements for memory access and execution prevention on Machine mode Extension.

Version: v1.0, 12/2021

33.1 Introduction

Being able to access the memory of a process running at a high privileged execution mode, such as the Supervisor or Machine mode, from a lower privileged mode such as the User mode, introduces an obvious attack vector since it allows for an attacker to perform privilege escalation, and tamper with the code and/or data of that process. A less obvious attack vector exists when the reverse happens, in which case an attacker instead of tampering with code and/or data that belong to a high-privileged process, can tamper with the memory of an unprivileged / less-privileged process and trick the high-privileged process to use or execute it.

To prevent this attack vector, two mechanisms known as Supervisor Memory Access Prevention (SMAP) and Supervisor Memory Execution Prevention (SMEP) were introduced in recent systems. The first one prevents the OS from accessing the memory of an unprivileged process unless a specific code path is followed, and the second one prevents the OS from executing the memory of an unprivileged process at all times. RISC-V already includes support for SMAP, through the sstatus.SUM bit, and for SMEP by always denying execution of virtual memory pages marked with the U bit, with Supervisor mode (OS) privileges, as mandated on the Privilege Spec.

Terms:

- PMP Entry: A pair of pmpcfg[i] / pmpaddr[i] registers.
- PMP Rule: The contents of a pmpcfg register and its associated pmpaddr register(s), that encode a valid protected physical memory region, where pmpcfg[i].A != OFF, and if pmpcfg[i].A == TOR, pmpaddr[i-1] < pmpaddr[i].
- Ignored: Any permissions set by a matching PMP rule are ignored, and all accesses to the requested address range are allowed.
- Enforced: Only access types configured in the PMP rule matching the requested address range are allowed; failures will cause an access-fault exception.
- Denied: Any permissions set by a matching PMP rule are ignored, and no accesses to the requested address range are allowed.; failures will cause an access-fault exception.
- Locked: A PMP rule/entry where the pmpcfg.L bit is set.
- PMP reset: A reset process where all PMP settings of the hart, including locked rules/settings, are re-initialized to a set of safe defaults, before releasing the hart (back) to the firmware / OS / application.

33.1.1 Threat model

However, there are no such mechanisms available on Machine mode in the current (v1.11) Privileged Spec. It is not possible for a PMP rule to be enforced only on non-Machine modes and denied on Machine mode, to only allow access to a memory region by less-privileged modes. it is only possible to have a locked rule that will be enforced on all modes, or a rule that will be enforced on non-Machine modes and be ignored by Machine mode. So for any physical memory region which is not protected with a Locked rule, Machine mode has unlimited access, including the ability to execute it.

Without being able to protect less-privileged modes from Machine mode, it is not possible to prevent the mentioned attack vector. This becomes even more important for RISC-V than on other architectures, since implementations are allowed where a hart only has Machine and User modes available, so the whole OS will run on Machine mode instead of the non-existent Supervisor mode. In such implementations the attack surface is greatly increased, and the same kind of attacks performed on Supervisor mode and mitigated through SMAP/SMEP, can be performed on Machine mode without any available mitigations. Even on implementations with Supervisor mode present attacks are still possible against the Firmware and/or the Secure Monitor running on Machine mode.

33.2 Proposal

1. Machine Security Configuration (mseccfg) is a new RW Machine mode CSR, used for configuring various security mechanisms present on the hart, and only accessible to Machine mode. It is 64 bits wide, and is at address 0x747 on RV64 and 0x747 (low 32bits), 0x757 (high 32bits) on RV32. All mseccfg fields defined on this proposal are WARL, and the remaining bits are reserved for future standard use and should always read zero. The reset value of mseccfg is implementation-specific, otherwise if backwards compatibility is a requirement it should reset to zero on hard reset.
2. On mseccfg we introduce a field on bit 2 called Rule Locking Bypass (mseccfg.RLB) with the following functionality:
   a. When mseccfg.RLB is 1 locked PMP rules may be removed/modified and locked PMP entries may be edited.
   b. When mseccfg.RLB is 0 and pmpcfg.L is 1 in any rule or entry (including disabled entries), then mseccfg.RLB remains 0 and any further modifications to mseccfg.RLB are ignored until a PMP reset.

   Note: that this feature is intended to be used as a debug mechanism, or as a temporary workaround during the boot process for simplifying software, and optimizing the allocation of memory and PMP rules. Using this functionality under normal operation, after the boot process is completed, should be avoided since it weakens the protection of M-mode-only rules. Vendors who don't need this functionality may hardwire this field to 0.

   ---

3. On mseccfg we introduce a field in bit 1 called Machine Mode Whitelist Policy (mseccfg.MMWP). This is a sticky bit, meaning that once set it cannot be unset until a PMP reset. When set it changes the default PMP policy for M-mode when accessing memory regions that don't have a matching PMP rule, to denied instead of ignored.
4. On mseccfg we introduce a field in bit 0 called Machine Mode Lockdown (mseccfg.MML). This is a sticky bit, meaning that once set it cannot be unset until a PMP reset. When mseccfg.MML is set the system's behavior changes in the following way:
   a. The meaning of pmpcfg.L changes: Instead of marking a rule as locked and enforced in all modes, it now marks a rule as M-mode-only when set and S/U-mode-only when unset. The formerly reserved encoding of pmpcfg.RW=01, and the encoding pmpcfg.LRWX=1111, now encode a Shared-Region.

      An M-mode-only rule is enforced on Machine mode and denied in Supervisor or User mode. It also remains locked so that any further modifications to its associated configuration or address registers are ignored until a PMP reset, unless mseccfg.RLB is set.

      An S/U-mode-only rule is enforced on Supervisor and User modes and denied on Machine mode.

      A Shared-Region rule is enforced on all modes, with restrictions depending on the pmpcfg.L and pmpcfg.X bits:

      - A Shared-Region rule where pmpcfg.L is not set can be used for sharing data between M-mode and S/U-mode, so is not executable. M-mode has read/write access to that region, and S/U-mode has read access if pmpcfg.X is not set, or read/write access if pmpcfg.X is set.

- A Shared-Region rule where pmpcfg.L is set can be used for sharing code between M-mode and S/U-mode, so is not writeable. Both M-mode and S/U-mode have execute access on the region, and M-mode also has read access if pmpcfg.X is set. The rule remains locked so that any further modifications to its associated configuration or address registers are ignored until a PMP reset, unless mseccfg.RLB is set.
- The encoding pmpcfg.LRWX=1111 can be used for sharing data between M-mode and S/U mode, where both modes only have read-only access to the region. The rule remains locked so that any further modifications to its associated configuration or address registers are ignored until a PMP reset, unless mseccfg.RLB is set.

b. Adding a rule with executable privileges that either is M-mode-only or a locked Shared-Region is not possible and such pmpcfg writes are ignored, leaving pmpcfg unchanged. This restriction can be temporarily lifted e.g. during the boot process, by setting mseccfg.RLB.

c. Executing code with Machine mode privileges is only possible from memory regions with a matching M-mode-only rule or a locked Shared-Region rule with executable privileges. Executing code from a region without a matching rule or with a matching S/U-mode-only rule is denied.

d. If mseccfg.MML is not set, the combination of pmpcfg.RW=01 remains reserved for future standard use.

33.2.1 Truth table when mseccfg.MML is set

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.*	

Fig. 33.1: Truth table when mseccfg.MML is set

: Locked rules cannot be removed or modified until a PMP reset, unless mseccfg.RLB is set.

33.2.2 Visual representation of the proposal

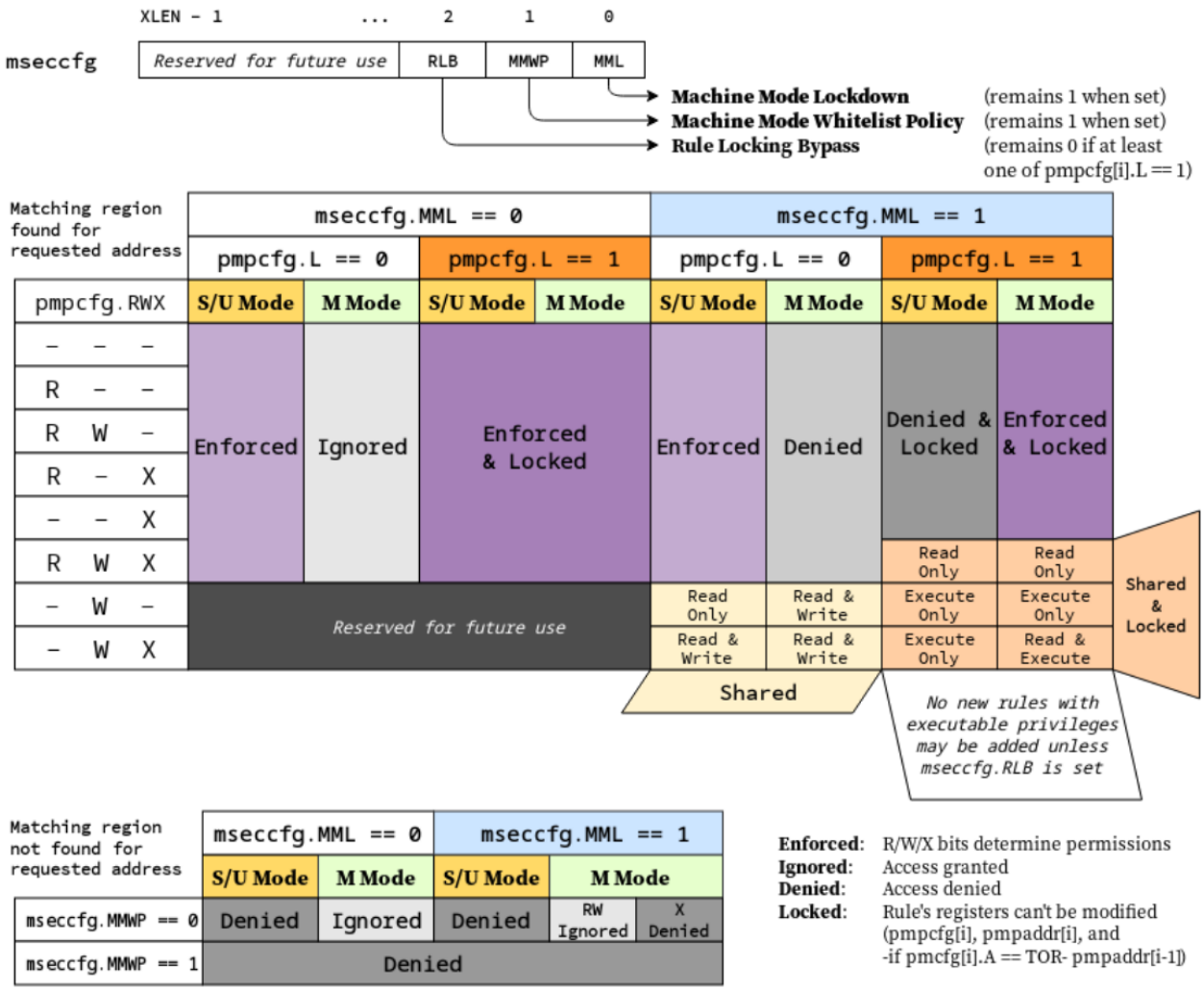


Fig. 33.2: visual representation of the proposal

33.2.3 Smepmp software discovery

Since all fields defined on mseccfg as part of this proposal are locked when set (MMWP/MML) or locked when cleared (RLB), software can't poll them for determining the presence of Smepmp. It is expected that BootROM will set mseccfg.MMWP and/or mseccfg.MML during early boot, before jumping to the firmware, so that the firmware will be able to determine the presence of Smepmp by reading mseccfg and checking the state of mseccfg.MMWP and mseccfg.MML.

33.3 Rationale

1. Since a CSR for security and / or global PMP behavior settings is not available with the current spec, we needed to define a new one. This new CSR will allow us to add further security configuration options in the future and also allow developers to verify the existence of the new mechanisms defined on this proposal.

2. There are use cases where developers want to enforce PMP rules in M-mode during the boot process, that are also able to modify, merge, and / or remove later on. Since a rule that is enforced in M-mode also needs to be locked (or else badly written or malicious M-mode software can remove it at any time), the only way for developers to approach this is to keep adding PMP rules to the chain and rely on rule priority. This is a waste of PMP rules and since it's only needed during boot, mseccfg.RLB is a simple workaround that can be used temporarily and then disabled and locked down.

   Also when mseccfg.MML is set, according to 4b it's not possible to add a Shared-Region rule with executable privileges. So RLB can be set temporarily during the boot process to register such regions. Note that it's still possible to register executable Shared-Region rules using initial register settings (that may include mseccfg.MML being set and the rule being set on PMP registers) on PMP reset, without using RLB.

   Note: Be aware that RLB introduces a security vulnerability if left set after the boot process is over and in general it should be used with caution, even when used temporarily. Having editable PMP rules in M-mode gives a false sense of security since it only takes a few malicious instructions to lift any PMP restrictions this way. It doesn't make sense to have a security control in place and leave it unprotected. Rule Locking Bypass is only meant as a way to optimize the allocation of PMP rules, catch errors durring debugging, and allow the bootrom/firmware to register executable Shared-Region rules. If developers / vendors have no use for such functionality, they should never set mseccfg.RLB and if possible hard-wire it to 0. In any case RLB should be disabled and locked as soon as possible.

   Note: If mseccfg.RLB is not used and left unset, it wil be locked as soon as a PMP rule/entry with the pmpcfg.L bit set is configured.

   Note: Since PMP rules with a higher priority override rules with a lower priority, locked rules must precede non-locked rules.

   ---

3. With the current spec M-mode can access any memory region unless restricted by a PMP rule with the pmpcfg.L bit set. There are cases where this approach is overly permissive, and although it's possible to restrict M-mode by adding PMP rules during the boot process, this can also be seen as a waste of PMP rules. Having the option to block anything by default, and use PMP as a whitelist for M-mode is considered a safer approach. This functionality may be used during the boot process or upon PMP reset, using initial register settings.

4. The current dual meaning of the pmpcfg.L bit that marks a rule as Locked and enforced on all modes is neither flexible nor clean. With the introduction of Machine Mode Lock-down the pmpcfg.L bit distinguishes between rules that are enforced only in M-mode (M-mode-only) or only in S/U-modes (S/U-mode-only). The rule locking becomes part of the definition of an M-mode-only rule, since when a rule is added in M mode, if not locked, can be modified or removed in a few instructions. On the other hand, S/U modes can't modify PMP rules anyway so locking them doesn't make sense.

   a. This separation between M-mode-only and S/U-mode-only rules also allows us to distinguish which regions are to be used by processes in Machine mode (pmpcfg.L == 1) and which by Supervisor or User mode processes (pmpcfg.L == 0), in the same way the U bit on the Virtual Memory's PTEs marks which Virtual Memory pages are to be used by User mode applications (U=1) and which by the Supervisor / OS (U=0). With this distinction in place we are able to implement memory access and execution prevention in M-mode for any physical memory region that is not M-mode-only.

      An attacker that manages to tamper with a memory region used by S/U mode, even after successfully tricking a process running in M-mode to use or execute that region, will fail to perform a successful attack since that region will be S/U-mode-only hence any access when in M-mode will trigger an access exception.

      In order to support zero-copy transfers between M-mode and S/U-mode we need to either allow shared memory regions, or introduce a mechanism similar to the sstatus.SUM bit to temporary allow the high-privileged mode

(in this case M-mode) to be able to perform loads and stores on the region of a less-privileged process (in this case S/U-mode). In our case after discussion within the group it seemed a better idea to follow the first approach and have this functionality encoded on a per-rule basis to avoid the risk of leaving a temporary, global bypass active when exiting M-mode, hence rendering memory access prevention useless.

Although it's possible to use mstatus.MPRV in M-mode to read/write data on an S/U-mode-only region using general purpose registers for copying, this will happen with S/U-mode permissions, honoring any MMU restrictions put in place by S-mode. Of course it's still possible for M-mode to tamper with the page tables and / or add S/U-mode-only rules and bypass the protections put in place by S-mode but if an attacker has managed to compromise M-mode to such extent, no security guarantees are possible in any way. Also note that the threat model we present here assumes buggy software in M-mode, not compromised software. We considered disabling mstatus.MPRV but it seemed too much and out of scope.

Shared-region rules can be used both for zero-copy data transfers and for sharing code segments. The latter may be used for example to allow S/U-mode to execute code by the vendor, that makes use of some vendor-specific ISA extension, without having to go through the firmware with an ecall. This is similar to the vDSO approach followed on Linux, that allows userspace code to execute kernel code without having to perform a system call.

To make sure that shared data regions can't be executed and shared code regions can't be modified, the encoding changes the meaning of the pmpcfg.X bit. In case of shared data regions, with the exception of the pmpcfg.LRWX=1111 encoding, the pmpcfg.X bit marks the capability of S/U-mode to write to that region, so it's not possible to encode an executable shared data region. In case of shared code regions, the pmpcfg.X bit marks the capability of M-mode to read from that region, and since pmpcfg.RW=01 is used for encoding the shared region, it's not possible to encode a shared writable code region.

Note: For adding Shared-region rules with executable privileges to share code segments between M-mode and S/U-mode, mseccfg.RLB needs to be implemented, or else such rules can only be added together with mseccfg.MML being set on PMP Reset. That's because the reserved encoding pmpcfg.RW=01 being used for Shared-region rules is only defined when mseccfg.MML is set, and 4b prevents the adition of rules with executable privileges on M-mode after mseccfg.MML is set unless mseccfg.RLB is also set.

Using the pmpcfg.LRWX=1111 encoding for a locked shared read-only data region was decided later on, its initial meaning was an M-mode-only read/write/execute region. The reason for that change was that the already defined shared data regions were not locked, so r/w access to M-mode couldn't be restricted. In the same way we have execute-only shared code regions for both modes, it was decided to also be able to allow a least-privileged shared data region for both modes. This approach allows for example to share the .text section of an ELF with a shared code region and the .rodata section with a locked shared data region, without allowing M-mode to modify .rodata. We also decided that having a locked read/write/execute region in M-mode doesn't make much sense and could be dangerous, since M-mode won't be able to add further restrictions there (as in the case of S/U-mode where S-mode can further limit access to an pmpcfg.LWRX=0111 region through the MMU), leaving the possibility of modifying an executable region in M-mode open.

For encoding Shared-region rules initially we used one of the two reserved bits on pmpcfg (bit 5) but in order to avoid allocating an extra bit, since those bits are a very limited resource, it was decided to use the reserved R=0,W=1 combination.

b. The idea with this restriction is that after the Firmware or the OS running in M-mode is initialized and mseccfg.MML is set, no new code regions are expected to be added since nothing else is expected to run in M-mode (everything else will run in S/U mode). Since we want to limit the attack surface of the system as much as possible, it makes sense to disallow any new code regions which may include malicious code, to be added/executed in M-mode.

c. In case mseccfg.MMWP is not set, M-mode can still access and execute any region not covered by a PMP rule. Since we try to prevent M-mode from executing malicious code and since an attacker may manage to place code on some region not covered by PMP (e.g. a directly-addressable flash memory), we need to ensure that M-mode can only execute the code segments initialized during firmware / OS initialization.

d. We are only using the encoding pmpcfg.RW=01 together with mseccfg.MML, if mseccfg.MML is not set the encoding remains usable for future use.

34 WorldGuard Specification

34.1 Revision History

Rev.	Revision Date	Revised Section	Revised Content
0.1	2022/10/31	N/A	1. Initial draft, from WorldGuard HAS Specification 1.1
0.2	2022/11/20	N/A	1. Second draft, for donation to RVIA
0.3	2023/4/03	N/A	1. Donated to RVIA
0.4	2023/4/27	N/A	1. Clarified checker operation. 2. Made clear that accesses other than 32b can be used. 3. Removed optional on NAPOT, as whole design is only a recommendation. 4. Removed requirement for hardware to reset bad address to bottom of checker range.

Note: 0.4 This update adds clarifications to the generic wgChecker design spec, and simplifies its handling of bad addresses.

34.2 Glossary/ Acronyms

Term	Meaning
CSR	Control and Status Register
MMU	Memory Management Unit
Privilege modes	The RISC-V privilege specification defines up to five privilege modes: M, [HS], U, VS, VU
REE	Rich Execution Environment
TEE	Trusted Execution Environment
WID	WorldGuard world identifier
WG	WorldGuard
XLEN	Refers to the width of an integer register in bits (either 32 or 64)

34.3 WorldGuard Overview

WorldGuard (WG) provides isolation in a hardware platform by constraining access to system physical addresses. WG provides Worlds , which are execution contexts that include agents (such as harts and devices) that can initiate a transaction on a physical address within a world, and resources (such as memories and peripheral devices) that respond to transactions at a physical address within a world. Worlds are created and configured by a trusted execution environment, usually at system boot time.

Worlds are uniquely identified by a hardware World Identifier (WID) and the maximum number of unique WIDs on a platform is NWorlds. Increasing NWorlds increases the hardware cost, and in practice, 2-8 WIDs (requiring 1-3 bits to represent) are sufficient for many use cases.

Hardware agents (harts and devices) that can initiate transactions on physical addresses may support multiple contexts, with each context potentially being in a different world. A software context running on a hart agent is present in one world at a time. A hardware context on a device agent is present in one world at a time.

Resources are identified by system physical addresses. A world may be granted read, write, or both, access permissions on a physical address. A resource can optionally be shared between worlds, with independent access permissions for each world.

WG is designed for the case where the allocation of agent contexts and resources to worlds is performed before or at reset/boot time, and not changed dynamically when the system is running unless there is a system reset. Efficiently changing WG configurations dynamically while the system is running is not a goal of the current specification.

When an agent context initiates a transaction to a physical address, hardware marks the transaction with the WID of the agent context. The transaction is only allowed to complete successfully if the targeted resource has the appropriate access permissions (read or write) on that address for the WID on the transaction. The permission checks might be performed at the agent, at the resource, or anywhere along the path the transaction takes through the platform's bus hierarchy. The bus transaction carries the WID through the interconnect and all elements on the path toward the targeted resource until access permissions can be checked. The method of propagating and checking the WID on busses is platform-specific, with different bus fabrics supporting WIDs in bus-specific ways.

Note: Theoretically all permissions checks on transactions could be performed at the source agent to prevent any illegal transactions from entering the bus fabric. But in practice, replicating and checking the entire platform permissions map at each agent is prohibitively expensive, particularly when permissions are configurable, and so WG assumes permissions checking is distributed out in the bus fabric and attached resources.

If the permission check fails, the transaction is terminated or modified to avoid violating world isolation and the failure may be reported. Failures may be reported in a number of ways depending on the platform, the agent, the resource, and the transaction type. Failures may be reported to the initiating agent, and optionally one or more other agents. In some cases, the failure cannot be directly reported to the initiating agent and the transaction is modified to be ignored or to return benign data. In these cases, the failure may still be reported to a different agent through an alternate mechanism.

34.4 RISC-V ISA WorldGuard Extensions

RISC-V harts that support WorldGuard associate a WID with all memory accesses initiated by that hart. The WorldGuard extensions allow different privilege modes on a hart to be tagged with different WIDs.

There are three levels of WG support on RISC-V harts. The first level does not require an ISA extension and fixes the WID for all privilege modes on a hart. The second level is the Smwg extension, which enables M-mode to control the WID of lower-privilege modes. The third level is the Smwgd extension, which further enables M-mode to delegate to [H]S-mode the ability to assign the WID of lower-privilege modes, thereby adding the Sswg extension to [H]S-mode.

All accesses, including implicit memory references such as instruction fetches and page-table walks, must be tagged with the appropriate WID. For the purposes of WG permissions, instruction fetches are treated as memory reads.

Note: Bus fabrics typically do not differentiate instruction fetches from memory reads, and so WG permissions checkers located on the other side of the bus fabric are unable to distinguish these two cases.

WorldGuard does not allow a privilege mode to change its own WID. The WID of M-mode on a hart is set by the external environment and does not change between resets.

Note: Different harts in a system may have different WIDs in M-mode.

34.4.1 WorldGuard CSRs

The WorldGuard Smwg, Smwgd, and Sswg extensions allow a hart to assign WIDs to its privilege modes. The new CSRs are listed in the Table below.

Size bits	Register	Access	Proposed CSR Address	Description
XLEN	mlwid	MRW	0x390	WID used for lower privilege modes. Ceil(Log2NWorlds) LSBs are used, others are zero.
XLEN	mwiddeleg	MRW	0x748	Set of WID values delegated to [HS]-mode, represented as a bit vector. NWorlds LSBs are used, others are zero.
XLEN	slwid	[HS]RW	0x190	WID value used in lower modes (i.e., U, VS, or VU). Ceil(Log2NWorlds) LSBs are used, others are zero.

These extensions supports NWorlds $\leq$ XLEN.

34.4.2 One world per hart

In this case, there are no ISA-visible additions to the RISC-V hart. The hart is reset into a single world and all transactions from that hart, regardless of privilege mode, are tagged with the WID of that world. How the hart is assigned to a world, or whether and how a hart is allowed to determine any information about WG configuration is platform-specific.

34.4.3 Smwg extension

The Smwg extension adds support for M-mode to control the world used by less-privileged modes, and can only be added to harts with at least two privilege modes.

The Smwg extension adds the mlwid CSR, which is an M-mode read-write CSR, whose least- significant bits set the WID to be used by lower-privilege modes.

Note: If the system supports demand-paged virtual memory, then any address-translation caches must ensure that translations are cached separately for each WID. A simple implementation can flush address-translation caches on any mlwid write.

The mlwid CSR is WARL, and if an illegal WID is written, the lowest-numbered legal WID is returned. It is platform-specific which worlds can be used by lower-privilege modes on this hart.

Note: M-mode software can use the WARL property of mlwid to discover which worlds are available to be assigned to lower modes, though in normal use, platform software will have predetermined allocations for the worlds on a platform.

Note: We constrain WARL behavior to reduce compatibilty-testing effort. Also, having a defined value slightly reduces time to dynamically search for valid WIDs.

M-mode may have a different WID than any assignable by mlwid to lower-privilege modes.

Note: The platform is not required to allow lower-privilege modes to be in the same world as M-mode.

At reset, mlwid must hold a WID that can be allocated to lower modes.

34.4.4 Smwgd / Sswg extensions

The Smwgd extension requires the Smwg extension and allows M-mode to delegate to [H]S-mode the ability to allocate WIDs to privilege modes lower than [H]S. The Smwgd extension optionally enables the Sswg extension for [H]S-mode.

The Smwgd extension adds the mwiddeleg M-mode read-write CSR. The mwiddeleg register represents a set of WIDs as a bit vector with WID i represented by bit i of the register. The mwiddeleg CSR is a WARL register where each bit that can be set indicates a WID that is delegated to [H]S-mode. The set of worlds that can be delegated to [H]S-mode on a hart is platform-specific.

Note: Different harts on a platform can have different sets of delegatable worlds.

The Sswg extension adds the [H]S-mode read-write slwid CSR, which sets the WID used for modes lower than [H]S-mode. The slwid is WARL, with mwiddeleg specifying the legal values for slwid. If an illegal WID is written to slwid, the lowest-numbered legal WID is returned. When Sswg is present, mlwid now specifies the WID for [H]S-mode only, and slwid specifies the WID for lower-privilege modes (U, VS, VU).

If the value in mwiddeleg is non-zero, the Sswg extension is enabled. When mwiddeleg is set to a non- zero value, slwid is initialized to the lowest-numbered world present in mwiddeleg.

If the value in mwiddeleg is zero, then the Sswg extension is disabled and accesses to slwid raise an illegal instruction exception.

Note: If the system supports demand-paged virtual memory, then any address-translation caches must ensure that translations are cached separately for each WID. A simple implementation can flush address-translation caches on any write to mwiddeleg or slwid.

When the hypervisor extension is present slwid sets the WID for both VS and VU mode, as well as U mode.

Note: The Sswg extension is not available to a guest OS.

There is no requirement for mwiddeleg to contain the WID in mlwid, i.e., S-mode can be set to a different world than the ones it is allowed to assign to lower modes using slwid when Sswg is enabled.

At reset, mwiddeleg is set to zero and hence Sswg is disabled.

34.4.5 Response to permission violations

When a hart attempts an explicit or implicit memory access that fails a WG permissions check, the access may or may not raise an access-fault exception of the appropriate type (i.e., instruction- access fault, load-access fault, or store/AMO-access fault). When an access-fault exception cannot be raised, the instruction performing the memory access can be retired but any writes to the protected physical memory location are ignored and any memory reads return data independent of the value in the protected physical memory location to avoid violating memory isolation.

Note: Secure systems will typically ensure that some agent is notified when an illegal access is attempted, even when an access-fault exception cannot be raised on the hart context.

Note: We cannot require that reads that fail permissions checks but that do not raise access-fault exceptions return a specific value (e.g, zero) to the hart as this is incompatible with some cache-coherence protocols, which may require cache- resident data be modifiable even when the underlying physical memory locations are protected and the bus responses previously returned zero.

---

35 RISC-V Advanced Core Local Interruptor Specification

35.1 Revision History

Rev.	Revision Date	Revised Section	Revised Content
1.0	2022/1/10	N/A	1. Dedicated chapter on synchronizing multiple MTIMER devices 2. Initial release with MTIMER, MSWI, and SSWI devices

35.2 Introduction

This RISC-V ACLINT specification defines a set of memory mapped devices which provide inter- processor interrupts (IPI) and timer functionalities for each HART on a multi-HART RISC-V platform. These HART-level IPI and timer functionalities are required by operating systems, bootloaders and firmwares running on a multi-HART RISC-V platform.

The SiFive Core-Local Interruptor (CLINT) device has been widely adopted in the RISC-V world to provide machine-level IPI and timer functionalities. Unfortunately, the SiFive CLINT has a unified register map for both IPI and timer functionalities and it does not provide supervisor-level IPI functionality.

The RISC-V ACLINT specification takes a more modular approach by defining separate memory mapped devices for IPI and timer functionalities. This modularity allows RISC-V platforms to omit some of the RISC-V ACLINT devices for when the platform has an alternate mechanism. In addition to modularity, the RISC-V ACLINT specification also defines a dedicated memory mapped device for supervisor-level IPIs. The *ACLINT Devices* (page 364) below shows the list of devices defined by the RISC-V ACLINT specification.

Table 35.2: ACLINT Devices

Name	Privilege Level	Functionality
MTIMER	Machine	Fixed-frequency counter and timer events
MSWI	Machine	Inter-processor(or software) interrupts
SSWI	Supervisor	Inter-processor(or software) interrupts

35.2.1 Backward Compatibility With SiFive CLINT

The RISC-V ACLINT specification is defined to be backward compatible with the SiFive CLINT specification. The register definitions and register offsets of the MTIMER and MSWI devices are compatible with the timer and IPI registers defined by the SiFive CLINT specification. A SiFive CLINT device on a RISC-V platform can be logically seen as one MSWI device and one MTIMER devices placed next to each other in the memory address space as shown in Table *One Sifive CLINT device is equivalent to two ACLINT devices* (page 365).

Table 35.3: One Sifive CLINT device is equivalent to two ACLINT devices

SiFive CLINT Offset Range	ACLINT Device	Functionality
0x0000_0000 - 0x0000_3fff	MSWI	Machine-level inter-processor (or software) interrupts
0x0000_4000 - 0x0000_bfff	MTIMER	Machine-level fixed-frequency counter and timer events

35.3 Machine-level Timer Device (MTIMER)

The MTIMER device provides machine-level timer functionality for a set of HARTs on a RISC-V platform. It has a single fixed-frequency monotonic time counter (MTIME) register and a time compare register (MTIMECMP) for each HART connected to the MTIMER device. A MTIMER device not connected to any HART should only have a MTIME register and no MTIMECMP registers.

On a RISC-V platform with multiple MTIMER devices:

- Each MTIMER device provides machine-level timer functionality for a different (or disjoint) set of

HARTs. A MTIMER device assigns a HART index starting from zero to each HART associated with it. The HART index assigned to a HART by the MTIMER device may or may not have any relationship with the unique HART identifier (hart ID) that the RISC-V Privileged Architecture assigns to the HART.

- Two or more MTIMER devices can share the same physical MTIME register while having their own separate MTIMECMP registers.
- The MTIMECMP registers of a MTIMER device must only compare again the MTIME register of the same MTIMER device for generating machine-level timer interrupt.

The maximum number of HARTs supported by a single MTIMER device is 4095 which is equivalent to the maximum number of MTIMECMP registers.

35.3.1 Register Map

A MTIMER device has two separate base addresses: one for the MTIME register and another for the MTIMECMP registers. These separate base addresses of a single MTIMER device allows multiple MTIMER devices to share the same physical MTIME register.

The Table *ACLINT MTIMER Time Register Map* (page 365) below shows map of the MTIME register whereas the Table *ACLINT MTIMER Compare Register Map* (page 365) below shows map of the MTIMECMP registers relative to separate base addresses.

Table 35.4: ACLINT MTIMER Time Register Map

Offset	Width	Attr	Name	Description
0x0000_0000	8B	RW	MTIME	Machine-level time counter

Table 35.5: ACLINT MTIMER Compare Register Map

Offsete	Width	Attr	Name	Description
0x0000_0000	8B	RW	MTIMECMP0	HART index 0 Machine-level time compare
0x0000_0000	8B	RW	MTIMECMP1	HART index 1 Machine-level time compare

continues on next page

Table 35.5 – continued from previous page

Offsete	Width	Attr	Name	Description
...				
0x0000_7ff0	8B	RW	MTIMECMP4094	HART index 4094 Machine-level time compare

35.3.2 MTIME Register (Offset: 0x00000000)

The MTIME register is a 64-bit read-write register that contains the number of cycles counted based on a fixed reference frequency.

On MTIMER device reset, the MTIME register is cleared to zero.

35.3.3 MTIMECMP Registers (Offsets: 0x00000000 - 0x00007FF0)

The MTIMECMP registers are per-HART 64-bit read-write registers. It contains the MTIME register value at which machine-level timer interrupt is to be triggered for the corresponding HART. The machine-level timer interrupt of a HART is pending whenever MTIME is greater than or equal to the value in the corresponding MTIMECMP register whereas the machine-level timer interrupt of a HART is cleared whenever MTIME is less than the value of the corresponding MTIMECMP register. The machine-level timer interrupt is reflected in the MTIP bit of the mip CSR. On MTIMER device reset, the MTIMECMP registers are in unknown state.

35.3.4 Synchronizing Multiple MTIME Registers

A RISC-V platform can have multiple HARTs grouped into hierarchical topology groups (such as clusters, nodes, or sockets) where each topology group has it's own MTIMER device. Further, such RISC-V platforms can also allow clock-gating or powering off for a topology group (including the MTIMER device) at runtime.

On a RISC-V platform with multiple MTIMER devices residing on the same die, each device must satisfy the RISC-V architectural requirement that all the MTIME registers with respect to each other, and all the per-HART time CSRs with respect to each other, are synchronized to within one MTIME tick period. For example, if the MTIME tick period is 10ns, then the MTIME registers, and their associated time CSRs, should respectively be synchronized to within 10ns of each other.

On a RISC-V platform with multiple MTIMER devices on different die, the MTIME registers (and their associated time CSRs) on different die may be synchronized to only within a specified interval of each other that is larger than the MTIME tick period. A platform may define a maximum allowed interval. To satisfy the preceding MTIME synchronization requirements:

- All MTIME registers should have the same input clock so as to avoid runtime drift between separate MTIME registers (and their associated time CSRs)
- Upon system reset, the hardware must initialize and synchronize all MTIME registers to zero
- When a MTIMER device is stopped and started again due to, say, power management actions, the

software should re-synchronize this MTIME register with all other MTIME registers

When software updates one, multiple, or all MTIME registers, it must maintain the preceding synchronization requirements (through measuring and then taking into account the differing latencies of performing reads or writes to the different MTIME registers).

As an example, the below RISC-V 64-bit assembly sequence can be used by software to synchronize a MTIME register with reference to another MTIME register.

```
/*
 * unsigned long aclint_mtime_sync(unsigned long target_mtime_address,
 *                                 unsigned long reference_mtime_address)
 */
   .globl aclint_mtime_sync
aclint_mtime_sync:
   /* Read target MTIME register in T0 register */
```

(continues on next page)

(continued from previous page)

```
ld    t0, (a0)
fence i, i

/* Read reference MTIME register in T1 register */
ld    t1, (a1)
fence i, i

/* Read target MTIME register in T2 register */
ld    t2, (a0)
fence i, i

/*
* Compute target MTIME adjustment in T3 register
* T3 = T1 - ((T0 + t2) / 2)
*/
srli t0, t0, 1
srli t2, t2, 1
add  t3, t0, t2
sub  t3, t1, t3

/* Update target MTIME register */
ld  t4, (a0)
add t4, t4, t3
sd  t4, (a0)

/* Return MTIME adjustment value */
add a0, t3, zero
ret
```

Note: On some RISC-V platforms, the MTIME synchronization sequence (i.e. the aclint_mtime_sync() function above) will need to be repeated few times until delta between target MTIME register and reference MTIME register is zero (or very close to zero).

35.4 Machine-level Software Interrupt Device (MSWI)

The MSWI device provides machine-level IPI functionality for a set of HARTs on a RISC-V platform. It has an IPI register (MSIP) for each HART connected to the MSWI device.

On a RISC-V platform with multiple MSWI devices, each MSWI device provides machine-level IPI functionality for a different (or disjoint) set of HARTs. A MSWI device assigns a HART index starting from zero to each HART associated with it. The HART index assigned to a HART by the MSWI device may or may not have any relationship with the unique HART identifier (hart ID) that the RISC-V Privileged Architecture assigns to the HART.

The maximum number of HARTs supported by a single MSWI device is 4095 which is equivalent to the maximum number of MSIP registers.

35.4.1 Register Map

Table 35.6: ACLINT MSWI Device Register Map

Offset	Width	Attr	Name	Description
0x0000_0000	4B	RW	MSIP0	HART index 0 machine-level IPI register
0x0000_0004	4B	RW	MSIP1	HART index 1 machine-level IPI register
...				
0x0000_3FFC	4B		RESERVED	Reserved for future use.

35.4.2 MSIP Registers(Offsets: 0x00000000 - 0x00003FF8)

Each MSIP register is a 32-bit wide WARL register where the upper 31 bits are wired to zero. The least significant bit is reflected in MSIP of the mip CSR. A machine-level software interrupt for a HART is pending or cleared by writing 1 or 0 respectively to the corresponding MSIP register. On MSWI device reset, each MSIP register is cleared to zero.

35.5 Supervisor-level Software Interrupt Device (SSWI)

The SSWI device provides supervisor-level IPI functionality for a set of HARTs on a RISC-V platform. It provides a register to set an IPI (SETSSIP) for each HART connected to the SSWI device.

On a RISC-V platform with multiple SSWI devices, each SSWI device provides supervisor-level IPI functionality for a different (or disjoint) set of HARTs. A SSWI device assigns a HART index starting from zero to each HART associated with it. The HART index assigned to a HART by the SSWI device may or may not have any relationship with the unique HART identifier (hart ID) that the RISC-V Privileged Architecture assigns to the HART.

The maximum number of HARTs supported by a single SSWI device is 4095 which is equivalent to the maximum number of SETSSIP registers.

35.5.1 Register Map

Table 35.7: ACLINT SSWI Device Register Map

Offset	Width	Attr	Name	Description
0x0000_0000	4B	RW	SETSSIP0	HART index 0 set supervisor-level IPI register
0x0000_0004	4B	RW	SETSSIP1	HART index 1 set supervisor-level IPI register
...				
0x0000_3FFC	4B		RESERVED	Reserved for future use.

35.5.2 SETSSIP Registers (Offsets: 0x00000000 - 0x00003FF8)

Each SETSSIP register is a 32-bit wide WARL register where the upper 31 bits are wired to zero. The least significant bit of a SETSSIP register always reads 0. Writing 0 to the least significant bit of a SETSSIP register has no effect whereas writing 1 to the least significant bit sends an edge-sensitive interrupt signal to the corresponding HART causing the HART to set SSIP in the mip CSR. Writes to a SETSSIP register are guaranteed to be reflected in SSIP of the corresponding HART but not necessarily immediately.

Note: The RISC-V Privileged Architecture defines SSIP in mip and sip CSRs as a writeable bit so the M-mode or S-mode software can directly clear SSIP.

36 Appendix

- **Nuclei RISC-V IP Products**: https://www.nucleisys.com/product.php
- **Nuclei Spec Documentation**: https://nucleisys.com/download.php#spec
- **Nuclei RISCV Tools and Documents**: https://nucleisys.com/download.php
- **Nuclei Prebuilt Toolchain and IDE**: https://nucleisys.com/download.php#tools
- **NMSIS**: https://github.com/Nuclei-Software/NMSIS
- **Nuclei SDK**: https://github.com/Nuclei-Software/nuclei-sdk
- **Nuclei Linux SDK**: https://github.com/Nuclei-Software/nuclei-linux-sdk
- **Nuclei Software Organization in Github**: https://github.com/Nuclei-Software/
- **RISC-V MCU Organization in Github**: https://github.com/riscv-mcu/
- **RISC-V MCU Community Website**: https://www.rvmcu.com/
- **Nuclei riscv-openocd**: https://github.com/riscv-mcu/riscv-openocd
- **Nuclei riscv-gnu-toolchain**: https://github.com/riscv-mcu/riscv-gnu-toolchain