



RISC-V Specification for CHERI Extensions

Authors: Hesham Almatary, Andres Amaya Garcia, John Baldwin, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Franz A. Fuchs, Timothy Hutt, Alexandre Joannou, Tariq Kurd, Ben Laurie, A. Theodore Marketos, David McKay, Jamie Melling, Stuart Menefy, Simon W. Moore, Peter G. Neumann, Robert Norton, Alexander Richardson, Michael Roe, Peter Rugg, Peter Sewell, Carl Shaw, Robert N. M. Watson, Jonathan Woodruff

Version v0.8.0-prerelease, 2024-04-15: Release

Table of Contents

Preamble	1
Copyright and license information	2
Contributors	3
1. Introduction	4
1.1. CHERI Concepts and Terminology	4
1.2. CHERI Extensions to RISC-V	5
1.3. Risks and Known Uncertainty	6
1.3.1. Pending Extensions	6
1.3.2. Incompatible Extensions	6
2. Anatomy of Capabilities in Zcheri_purecap	7
2.1. Capability Encoding	7
2.2. Components of a Capability	8
2.2.1. Tag	8
2.2.2. Address	8
2.2.3. Architectural Permissions (AP)	8
Description	8
Permission Encoding	9
Permission Transitions	11
2.2.4. Software-Defined Permissions (SDP)	11
2.2.5. Sealed (S) Bit	11
2.2.6. Bounds (EF, T, TE, B, BE)	12
Concept	12
Decoding	13
Malformed Capability Bounds	15
2.3. Special Capabilities	16
2.3.1. NULL Capability	16
2.3.2. Infinite Capability	16
2.4. Representable Range Check	17
2.4.1. Concept	17
2.4.2. Practical Information	17
3. Integrating Zcheri_purecap with the RISC-V Base Integer Instruction Set	19
3.1. Memory	19
3.2. Programmer's Model for Zcheri_purecap	19
3.2.1. PCC - The Program Counter Capability	19
3.3. Capability Instructions	20
3.3.1. Capability Inspection Instructions	20
3.3.2. Capability Manipulation Instructions	21
3.3.3. Capability Load and Store Instructions	21
3.4. Existing RISC-V Instructions	22
3.4.1. Integer Computational Instructions	22

3.4.2. Control Transfer Instructions	22
Unconditional Jumps	23
Conditional Branches	23
3.4.3. Integer Load and Store Instructions	23
3.5. Zicsr, Control and Status Register (CSR) Instructions	24
3.5.1. CSR Instructions	24
3.6. Control and Status Registers (CSRs)	25
3.7. Machine-Level CSRs	25
3.7.1. Machine Status Registers (mstatus and mstatush)	26
3.7.2. Machine Trap-Vector Base-Address Registers (mtvec)	26
3.7.3. Machine Trap-Vector Base-Address Capability Registers (mtvecc)	26
3.7.4. Machine Scratch Register (mscratch)	27
3.7.5. Machine Scratch Register Capability (mscratchc)	27
3.7.6. Machine Exception Program Counter (mepc)	27
3.7.7. Machine Exception Program Counter Capability (mepcc)	27
3.7.8. Machine Cause Register (mcause)	28
3.7.9. Machine Trap Delegation Register (medeleg)	30
3.7.10. Machine Trap Value Register (mtval)	30
3.8. Supervisor-Level CSRs	31
3.8.1. Supervisor Trap Vector Base Address Registers (stvec)	31
3.8.2. Supervisor Trap Vector Base Address Registers (stvecc)	31
3.8.3. Supervisor Scratch Register (sscratch)	32
3.8.4. Supervisor Scratch Registers (sscratchc)	32
3.8.5. Supervisor Exception Program Counter (sepc)	32
3.8.6. Supervisor Exception Program Counter Capability (sepcc)	32
3.8.7. Supervisor Cause Register (scause)	32
3.8.8. Supervisor Trap Value Register (stval)	33
3.9. Unprivileged CSRs	34
3.10. CHERI Exception handling	34
3.11. CHERI Exceptions and speculative execution	36
3.12. Physical Memory Attributes (PMA)	36
3.13. Page-Based Virtual-Memory Systems	36
3.13.1. Invalid Address Handling	36
Accessing CSRs	37
Branches and Jumps	37
Memory Accesses	37
3.14. Integrating Zcheri_purecap with Sdext	38
3.14.1. Debug Mode	39
3.14.2. Core Debug Registers	39
3.14.3. Debug Program Counter (dpc)	39
3.14.4. Debug Program Counter Capability (dpcc)	39
3.14.5. Debug Scratch Register 0 (dscratch0)	40
3.14.6. Debug Scratch Register 0 (dscratch0c)	40

3.14.7. Debug Scratch Register 1 (dscratch1)	40
3.14.8. Debug Scratch Register 1 (dscratch1c)	40
3.14.9. Debug Infinite Capability Register (dinfrc)	41
3.15. Integrating Zcheri_purecap with Sdtrig	41
4. "Zcheri_pte" Extension for CHERI Page-Based Virtual-Memory Systems	43
4.1. Extending the Page Table Entry Format	43
4.2. Extending the Machine Environment Configuration Register (menvcfg)	44
5. "Zcheri_legacy" Extension for CHERI Legacy Mode	46
5.1. CHERI Execution Mode	46
5.2. CHERI Execution Mode Encoding	46
5.3. Zcheri_legacy Instructions	47
5.3.1. Capability Load and Store Instructions	47
5.3.2. Capability Manipulation Instructions	47
5.3.3. Mode Change Instructions	47
5.4. Existing RISC-V Instructions	48
5.4.1. Control Transfer Instructions	48
5.4.2. Conditional Branches	48
5.4.3. Load and Store Instructions	48
5.4.4. CSR Instructions	48
5.5. Integrating Zcheri_legacy with Sdext	49
5.6. Debug Default Data Capability (dddc)	50
5.7. Disabling CHERI Registers	50
5.8. Added CLEN-wide CSRs	51
5.8.1. Machine Status Registers (mstatus and mstatush)	51
5.8.2. Machine Trap Default Capability Register (mtdc)	52
5.8.3. Machine Security Configuration Register (mseccfg)	52
5.8.4. Machine Environment Configuration Register (menvcfg)	52
5.8.5. Supervisor Trap Default Capability Register (stdc)	53
5.8.6. Supervisor Environment Configuration Register (senvcfg)	53
5.8.7. Default Data Capability (ddc)	53
6. "Zstid Extension for Thread Identification	55
6.1. Control and Status Registers (CSRs)	55
6.2. Supervisor-Level and Unprivileged CSRs	55
6.2.1. Supervisor Thread Identifier (stid)	55
6.2.2. User Thread Identifier (utid)	55
6.2.3. Supervisor Thread Identifier Capability (stidc)	55
6.2.4. User Thread Identifier Capability (utidc)	56
6.3. CHERI Compartmentalization	56
7. RISC-V Instructions and Extensions Reference	57
7.1. "Zcheri_purecap" and "Zcheri_legacy" Extensions for CHERI	58
7.1.1. CMV	59
7.1.2. MODESW	60
7.1.3. CADDI	61

7.1.4. CADD	61
7.1.5. SCADDR	63
7.1.6. ACPERM	64
7.1.7. SCMODE	66
7.1.8. SCHI	67
7.1.9. SCEQ	68
7.1.10. SENTRY	69
7.1.11. SCSS	70
7.1.12. CBLD	71
7.1.13. GCTAG	72
7.1.14. GCPERM	73
7.1.15. GCHI	74
7.1.16. GCBASE	75
7.1.17. GCLEN	76
7.1.18. SCBNDSI	77
7.1.19. SCBNDS	77
7.1.20. SCBNDSR	78
7.1.21. CRAM	79
7.1.22. LC	80
7.1.23. SC	82
7.2. RV32I/E and RV64I/E Base Integer Instruction Sets	84
7.2.1. AUIPC	85
7.2.2. BEQ, BNE, BLT[U], BGE[U]	86
7.2.3. JR	87
7.2.4. JALR	87
7.2.5. J	89
7.2.6. JAL	89
7.2.7. LD	90
7.2.8. LWU	90
7.2.9. LW	90
7.2.10. LHU	90
7.2.11. LH	90
7.2.12. LBU	90
7.2.13. LB	91
7.2.14. SD	93
7.2.15. SW	93
7.2.16. SH	93
7.2.17. SB	94
7.2.18. SRET	96
7.2.19. MRET	96
7.2.20. DRET	97
7.3. "A" Standard Extension for Atomic Instructions	98
7.3.1. AMO<OP>.W	99

7.3.2. AMO<OP>.D	100
7.3.3. AMOSWAP.C	102
7.3.4. LR.D	104
7.3.5. LR.W	104
7.3.6. LR.H	104
7.3.7. LR.B	105
7.3.8. LR.C	107
7.3.9. SC.D	109
7.3.10. SC.W	109
7.3.11. SC.H	109
7.3.12. SC.B	110
7.3.13. SC.C	112
7.4. "Zicsr", Control and Status Register (CSR) Instructions	114
7.4.1. CSRRW	115
7.4.2. CSRRWI	116
7.4.3. CSRRS	116
7.4.4. CSRRSI	116
7.4.5. CSRRC	116
7.4.6. CSRRCI	117
7.5. "Zfh", "Zfhmin", "F" and "D" Standard Extension for Floating-Point	119
7.5.1. FLD	120
7.5.2. FLW	120
7.5.3. FLH	121
7.5.4. FSD	123
7.5.5. FSW	123
7.5.6. FSH	124
7.6. "C" Standard Extension for Compressed Instructions	126
7.6.1. C.BEQZ, C.BNEZ	127
7.6.2. C.MV	128
7.6.3. C.ADDI16SP	129
7.6.4. C.ADDI4SPN	130
7.6.5. C.MODESW	131
7.6.6. C.JALR	132
7.6.7. C.JR	133
7.6.8. C.JAL	134
7.6.9. C.J	135
7.6.10. C.LD	136
7.6.11. C.LW	137
7.6.12. C.LWSP	139
7.6.13. C.LDSP	140
7.6.14. C.FLW	142
7.6.15. C.FLWSP	142
7.6.16. C.FLD	143

7.6.17. C.FLDSP	143
7.6.18. C.LC	145
7.6.19. C.LCSP	145
7.6.20. C.SD	147
7.6.21. C.SW	148
7.6.22. C.SWSP	150
7.6.23. C.SDSP	151
7.6.24. C.FSW	153
7.6.25. C.FSWSP	153
7.6.26. C.FSD	154
7.6.27. C.FSDSP	154
7.6.28. C.SC	156
7.6.29. C.SCSP	156
7.7. "Zicbom", "Zicbop", "Zicboz" Standard Extensions for Base Cache Management Operations ...	158
7.7.1. CBO.CLEAN	159
7.7.2. CBO.FLUSH	160
7.7.3. CBO.INVALID	161
7.7.4. CBO.ZERO	162
7.7.5. PREFETCH.I	164
7.7.6. PREFETCH.R	165
7.7.7. PREFETCH.W	166
7.8. "Zba" Extension for Bit Manipulation Instructions	167
7.8.1. ADD.UW	168
7.8.2. SH1ADD	169
7.8.3. SH2ADD	169
7.8.4. SH3ADD	170
7.8.5. SH1ADD.UW	171
7.8.6. SH2ADD.UW	171
7.8.7. SH3ADD.UW	172
7.8.8. SH4ADD	173
7.8.9. SH4ADD.UW	174
7.9. "Zcb" Standard Extension For Code-Size Reduction	175
7.9.1. C.LH	176
7.9.2. C.LHU	176
7.9.3. C.LBU	177
7.9.4. C.SH	179
7.9.5. C.SB	180
7.10. "Zcmp" Standard Extension For Code-Size Reduction	182
7.10.1. CM.PUSH	183
7.10.2. CM.POP	184
7.10.3. CM.POPRET	185
7.10.4. CM.POPRETZ	187
7.10.5. CM.MVSA01	189

7.10.6. CM.MVA01S	190
7.11. "Zcmt" Standard Extension For Code-Size Reduction	191
7.11.1. Jump Vector Table CSR (jvt)	191
7.11.2. Jump Vector Table CSR (jvtc)	191
7.11.3. CM.JALT	192
7.11.4. CM.JT	194
8. Extension summary	196
8.1. Zabhlrsc	196
8.2. Zcheri_purecap	196
8.3. Zcheri_legacy	203
8.4. Instruction Modes	204
9. Capability Width CSR Summary	212
9.1. Other tables	215
Bibliography	218

Preamble



This document is in the [Development state](#)

Expect potential changes. This draft specification is likely to evolve before it is accepted as a standard. Implementations based on this draft may not conform to the future standard.

Copyright and license information

This specification is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). The full license text is available at creativecommons.org/licenses/by/4.0/.

Copyright 2024 by RISC-V International.

Contributors

This RISC-V specification has been contributed to directly or indirectly by:

- Hesham Almatary <hesham.almatary@cl.cam.ac.uk>
- Andres Amaya Garcia <andres.amaya@codasip.com>
- John Baldwin <jhb61@cl.cam.ac.uk>
- David Chisnall <david.chisnall@cl.cam.ac.uk>
- Jessica Clarke <jessica.clarke@cl.cam.ac.uk>
- Brooks Davis <brooks.davis@sri.com>
- Nathaniel Wesley Filardo <nwf20@cam.ac.uk>
- Franz A. Fuchs <faf28@cam.ac.uk>
- Timothy Hutt <timothy.hutt@codasip.com>
- Alexandre Joannou <alexandre.joannou@cl.cam.ac.uk>
- Tariq Kurd <tariq.kurd@codasip.com>
- Ben Laurie <benl@google.com>
- A. Theodore Markettos <theo.markettos@cl.cam.ac.uk>
- David McKay <david.mckay@codasip.com>
- Jamie Melling <jamie.melling@codasip.com>
- Stuart Menefy <stuart.menefy@codasip.com>
- Simon W. Moore <simon.moore@cl.cam.ac.uk>
- Peter G. Neumann <neumann@csl.sri.com>
- Robert Norton <robert.norton@cl.cam.ac.uk>
- Alexander Richardson <alexrichardson@google.com>
- Michael Roe <mr101@cam.ac.uk>
- Peter Rugg <peter.rugg@cl.cam.ac.uk>
- Peter Sewell <peter.sewell@cl.cam.ac.uk>
- Carl Shaw <carl.shaw@codasip.com>
- Robert N. M. Watson <robert.watson@cl.cam.ac.uk>
- Jonathan Woodruff <jonathan.woodruff@cl.cam.ac.uk>

Chapter 1. Introduction

1.1. CHERI Concepts and Terminology

Current CPU architectures (including RISC-V) allow memory access solely by specifying and dereferencing a memory address stored as an integer value in a register or in memory. Any accidental or malicious action that modifies such an integer value can result in unrestricted access to the memory that it addresses. Unfortunately, this weak memory protection model has resulted in the majority of software security vulnerabilities present in software today.

CHERI enables software to efficiently implement fine-grained memory protection and scalable software compartmentalization by providing strong, efficient hardware mechanisms to support software execution and enable it to prevent and mitigate vulnerabilities.

Design goals include incremental adoptability from current ISAs and software stacks, low performance overhead for memory protection, significant performance improvements for software compartmentalization, formal grounding, and programmer-friendly underpinnings. It has been designed to provide strong, non-probabilistic protection rather than depending on short random numbers or truncated cryptographic hashes that can be leaked and reinjected, or that could be brute forced.

CHERI enhances the CPU to add hardware memory access control. It has an additional memory access mechanism that protects *references to code and data* (pointers), rather than the *location of code and data* (integer addresses). This mechanism is implemented by providing a new primitive, called a **capability**, that software components can use to implement strongly protected pointers within an address space.

Capabilities are unforgeable and delegatable tokens of authority that grant software the ability to perform a specific set of operations. In CHERI, integer-based pointers can be replaced by capabilities to provide memory access control. In this case, a memory access capability contains an integer memory address that is extended with metadata to protect its integrity, limit how it is manipulated, and control its use. This metadata includes:

- an out-of-band *tag* implementing strong integrity protection (differentiating valid and invalid capabilities), This prevents confusion between data and capabilities.
- *bounds* limiting the range of addresses that may be dereferenced
- *permissions* controlling the specific operations that may be performed
- *sealing* which is used to support higher-level software encapsulation

The CHERI model is motivated by the *principle of least privilege*, which argues that greater security can be obtained by minimizing the privileges accessible to running software. A second guiding principle is the *principle of intentional use*, which argues that, where many privileges are available to a piece of software, the privilege to use should be explicitly named rather than implicitly selected. While CHERI does not prevent the expression of vulnerable software designs, it provides strong vulnerability mitigation: attackers have a more limited vocabulary for attacks, and should a vulnerability be successfully exploited, they gain fewer rights, and have reduced access to further attack surfaces.

Protection properties for capabilities include the ISA ensuring that capabilities are always derived via valid manipulations of other capabilities (*provenance*), that corrupted in-memory capabilities cannot be dereferenced (*integrity*), and that rights associated with capabilities shall only ever be equal or less permissive (*monotonicity*). Tampering or modifying capabilities in an attempt to elevate their rights will

yield an invalid capability as the tag will be cleared. Attempting to dereference via an invalid capability will result in a hardware exception.

CHERI capabilities may be held in registers or in memories, and are loaded, stored, and dereferenced using CHERI-aware instructions that expect capability operands rather than integer addresses. On hardware reset, initial capabilities are made available to software via capability registers. All other capabilities will be derived from these initial valid capabilities through valid capability transformations.

Developers can use CHERI to build fine-grained spatial and temporal memory protection into their system software and applications and significantly improve their security.

1.2. CHERI Extensions to RISC-V

This specification is based on publicly available documentation including ([Watson et al., 2023](#)) and ([Woodruff et al., 2019](#)). It defines the following extensions to support CHERI alongside RISC-V:

Zcheri_purecap

Introduces key, minimal CHERI concepts and features to the RISC-V ISA. The resulting extended ISA is not backwards-compatible with RISC-V

Zcheri_legacy

Extends Zcheri_purecap with features to ensure that the ISA extended with CHERI allows backwards binary compatibility with RISC-V. It also adds a mode bit in the encoding of capabilities to allow changing the current CHERI execution mode using indirect jump instructions

Zcheri_pte

CHERI extension for RISC-V harts supporting page-based virtual-memory

Zcheri_vectorcap

CHERI extension for the RISC-V Vector (V) extension. It adds support for storing CHERI capabilities in vector registers, intended for vectorised memory copying

Zstid

Extension for supporting thread identifiers. This extension improves software compartmentalization on CHERI systems.



The extension names are provisional and subject to change.

Zcheri_purecap is defined as the base extension which all CHERI RISC-V implementations must support. Zcheri_legacy and Zcheri_pte are optional extensions in addition to Zcheri_purecap.

If a standard vector extension is present (indicated in this document as "V", but it could equally be one of the subsets defined by a Zve* extension) then Zcheri_vectorcap may optionally be added in addition to Zcheri_purecap.

We refer to software as *purecap* if it utilizes CHERI capabilities for all memory accesses – including loads, stores and instruction fetches – rather than integer addresses. Purecap software requires the CHERI RISC-V hart to support Zcheri_purecap. We refer to software as *hybrid* if it uses integer addresses or CHERI capabilities for memory accesses. Hybrid software requires the CHERI RISC-V hart to support Zcheri_purecap and Zcheri_legacy.

See [Chapter 7](#) for compatibility with other RISC-V extensions.

1.3. Risks and Known Uncertainty

- All extensions could be divided up differently in the future, including after ratification
- The RISC-V Architecture Review Committee (ARC) are likely to update all encodings
- The ARC are likely to update all CSR addresses
- Instruction mnemonics may be renamed
 - Any changes will affect assembly code, but assembler aliases can provide backwards compatibility
- There is no clarity on how the new Page Table Entry (PTE) bits from Zcheri_pte will be implemented
 - The PTE bits introduce a dependency between exceptions and the stored tag bit
- There is debate on whether different permission encodings are needed for MXLEN=32 and MXLEN=64

1.3.1. Pending Extensions

The base RISC-V ISAs, along with most extensions, have been reviewed for compatibility with CHERI. However, the following extensions are yet to be reviewed:

- "V" Standard Extension for Vector Operations
- "H" Hypervisor Extension
- Core-Local Interrupt Controller (CLIC)



The list above is not complete!

1.3.2. Incompatible Extensions

There are RISC-V extensions in development that may duplicate some aspects of CHERI functionality or directly conflict with CHERI and should not be available on a CHERI-enabled hart. These include:

- RISC-V CFI specification
- "J" Pointer Masking



The list above is not complete!

Chapter 2. Anatomy of Capabilities in Zcheri_purecap

RISC-V defines variants of the base integer instruction set characterized by the width of the integer registers and the corresponding size of the address space. There are two primary ISA variants, RV32I and RV64I, which provide 32-bit and 64-bit address spaces respectively. The term XLEN refers to the width of an integer register in bits (either 32 or 64). The value of XLEN may change dynamically at run-time depending on the values written to CSRs, so we define capability behavior in terms of MXLEN, which is the value of XLEN used in machine mode and the widest XLEN the implementation supports.



Zcheri_purecap assumes a version of the privileged architecture which defines MXLEN as constant and requires higher privilege modes to have at least the same XLEN as lower privilege modes; these changes are present in the current draft and expected to be part of privileged architecture 1.13.

Zcheri_purecap defines capabilities of size CLEN corresponding to $2 * MXLEN$ without including the tag bit. The value of CLEN is always calculated based on MXLEN regardless of the effective XLEN value.

2.1. Capability Encoding



CHERI v9 Note: *The encoding changes eliminate the concept of the in-memory format, and also increase precision for RV32.*

The components of a capability, except the tag, are encoded as shown in [Figure 1](#) for MXLEN=32 and [Figure 2](#) for MXLEN=64. Each memory location or register able to hold a capability must also store the tag as out of band information that software cannot directly set or clear. The capability metadata is held in the most significant bits and the address is held in the least significant bits.

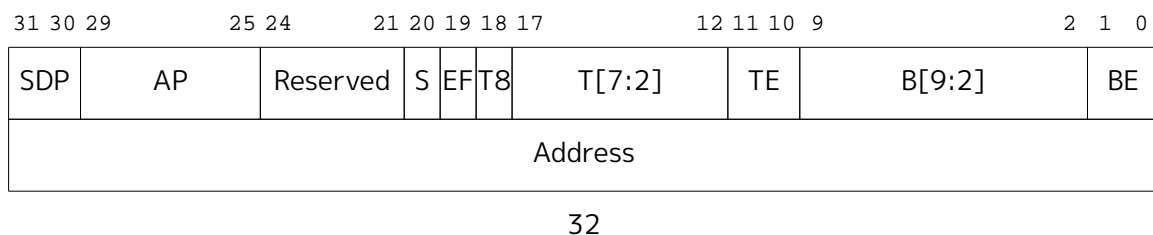


Figure 1. Capability encoding for MXLEN=32

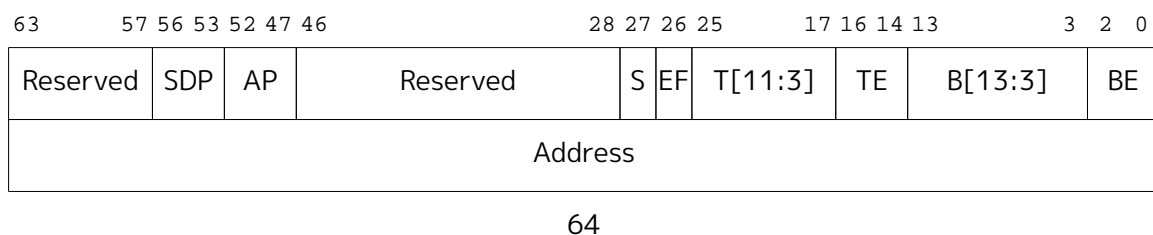


Figure 2. Capability encoding for MXLEN=64

Reserved bits are available for future extensions to Zcheri_purecap.



Reserved bits must be 0 in valid capabilities.

2.2. Components of a Capability

Capabilities contain the software accessible fields described in this section.

2.2.1. Tag

The tag is an additional hardware managed bit added to addressable memory and registers. It is stored separately and may be referred to as "out of band". It indicates whether a register or CLEN-aligned memory location contains a valid capability. If the tag is set, the capability is valid and can be dereferenced (contingent on checks such as permissions or bounds).

The capability is invalid if the tag is clear. Using an invalid capability to dereference memory or authorize any operation gives rise to exceptions. All capabilities derived from invalid capabilities are themselves invalid i.e. their tags are 0.

All locations in registers or memory able to hold a capability are CLEN+1 bits wide including the tag bit. Those locations are referred as being *CLEN-bit* or *capability* wide in this specification.

2.2.2. Address

The byte-address of a memory location is encoded as MXLEN integer value.

Table 1. Address widths depending on MXLEN

MXLEN	Address width
32	32
64	64

2.2.3. Architectural Permissions (AP)

Description



CHERI v9 Note: *The permissions are encoded differently in this specification.*

This field encodes architecturally defined permissions of the capability. Permissions grant access subject to the tag being set, the capability being unsealed (see [Section 2.2.5](#)), and bounds checks (see [Section 2.2.6](#)). An operation is also contingent on requirements imposed by other RISC-V architectural features, such as virtual memory, PMP and PMAs, even if the capability grants sufficient permissions. The permissions currently defined in Zcheri_purecap are listed below.

Read Permission (R)

Allow reading integer data from memory. Tags are always read as zero when reading integer data.

Write Permission (W)

Allow writing integer data to memory. Tags are always written as zero when writing integer data. Every CLEN aligned word in memory has a tag, if any byte is overwritten with integer data then the tag for all CLEN-bits is cleared.

Capability Permission (C)

Allow reading capability data from memory if the authorising capability also grants [R-permission](#).

Allow writing capability data to memory if the authorising capability also grants [W-permission](#).

Execute Permission (X)

Allow instruction execution.

Access System Registers Permission (ASR)

Allow access to privileged CSRs.

Permission Encoding

The bit width of the permissions field depends on the value of MXLEN as shown in [Table 2](#). A 5-bit vector encodes the permissions when MXLEN=32. For this case, the legal encodings of permissions are listed in [Table 3](#). Certain combinations of permissions are impractical. For example, [C-permission](#) is superfluous when the capability does not grant either [R-permission](#) or [W-permission](#). Therefore, it is only possible to encode a subset of all combinations.

Table 2. Permissions widths depending on MXLEN

MXLEN	Permissions width	Comment
32	5	Encodes some combinations of 5 permission bits and the M-bit
64	6	Separate bits for each permission and the M-bit

For MXLEN=32, the permissions encoding is split into four quadrants. The quadrant is taken from bits [4:3] of the permissions encoding. The meaning for bits [2:0] are shown in [Table 3](#) for each quadrant.

Quadrants 2 and 3 are arranged to implicitly grant future permissions which may be added with the existing allocated encodings. Quadrant 0 does the opposite - the encodings are allocated *not* to implicitly add future permissions, and so granting future permissions will require new encodings. Quadrant 1 encodes permissions for executable capabilities and the [M-bit](#).

Table 3. Encoding of architectural permissions for MXLEN=32

Encoding[2:0]	R	W	C	X	ASR	Mode	Notes
Quadrant 0: Non-capability data read/write							
bit[2] - write, bit[1] - reserved (0), bit[0] - read							
Reserved bits for future extensions are 0 so new permissions are not implicitly granted							
0						N/A	No permissions
1	✓					N/A	Data RO
2-3	reserved						
4		✓				N/A	Data WO
5	✓	✓				N/A	Data RW
6-7	reserved						
Quadrant 1: Executable capabilities							

Encoding[2:0]	R	W	C	X	ASR	Mode	Notes
bit[0] - M-bit (1-capability, 0-legacy)							
0-1	✓	✓	✓	✓	✓	Mode	Execute + ASR (see Infinite)
2-3	✓		✓	✓		Mode	Execute + Data & Cap RO
4-5	✓	✓	✓	✓		Mode	Execute + Data & Cap RW
6-7	✓	✓		✓		Mode	Execute + Data RW
Quadrant 2: Reserved							
<i>Reserved bits for future extensions must be 1 so they are implicitly granted</i>							
0-7	reserved						
Quadrant 3: Capability data read/write							
[2] - write. R and C implicitly granted.							
<i>Reserved bits for future extensions must be 1 so they are implicitly granted</i>							
0-2	reserved						
3	✓		✓			N/A	Data & Cap RO
4-6	reserved						
7	✓	✓	✓			N/A	Data & Cap RW



When MXLEN=32 there are many reserved permission encodings (see [Table 3](#)). It is not possible for a tagged capability to have one of these values since **ACPERM** will never create it. It is possible for untagged capabilities to have reserved values. **GCPERM** will interpret reserved values as if it were 0b000000 (no permissions). Future extensions may assign meanings to the reserved bit patterns, in which case **GCPERM** is allowed to report a non-zero value.

A 6-bit vector encodes the permissions and **M-bit** when MXLEN=64. In this case, there is a bit per permission as shown in [Table 4](#). A permission is granted if its corresponding bit is set, otherwise the capability does not grant that permission.

Table 4. Encoding of architectural permissions for MXLEN=64

Bit	Name
0	C-permission
1	W-permission
2	R-permission
3	X-permission
4	ASR-permission
5	M-bit

For both MXLEN=32 and MXLEN=64, the **M-bit** is only assigned meaning when the implementation supports Zcheri_legacy, otherwise the bit is reserved and must be 0 in valid capabilities.

Permission Transitions

Executing [ACPERM](#) can result in sets of permissions which cannot be represented when MXLEN=32 (see [Table 3](#)) or permission combinations which are not useful for MXLEN=64, such as [ASR-permission](#) set without [X-permission](#).

These cases are defined to return useful minimal sets of permissions, which may be no permissions. See [ACPERM](#) for these rules.



Future extensions may allow more combinations of permissions, especially for MXLEN=64.

2.2.4. Software-Defined Permissions (SDP)



CHERI v9 Note: *CHERI v9 had no software-defined permissions for RV32*

A bit vector used by the kernel or application programs for software-defined permissions (SDP).



Software is completely free to define the usage of these bits. For example, a program may decide to use an SDP bit to indicate the "ownership" of objects. Therefore, a capability grants permission to free the memory it references if that SDP bit is set because it "owns" that object.

Table 5. SDP widths depending on MXLEN

MXLEN	SDP width
32	2
64	4

2.2.5. Sealed (S) Bit



CHERI v9: *The sealing bit is new (1-bit otype) and the old CHERI v9 otype no longer exists. Please note that this bit indicates the result of two instructions in CHERI v9: CSEAL for sealed capabilities and CSEALENTRY for sealed entry capabilities.*

This bit indicates that a capability is sealed if the bit is 1 or unsealed if it is 0.

The sealing bit conflates two concepts in one bit: Sealing data capabilities and creating sealed entry capabilities as described below.

Sealed capabilities cannot be dereferenced to access memory and are immutable such that modifying any of its fields clears the tag of the output capability.



Sealed capabilities might be useful to software as tokens that can be passed around. The only way of removing the seal bit of a capability is by rebuilding it via a superset capability with [CBLD](#). Zcheri_purecap does not offer an unseal instruction.

For code capabilities, the sealing bit is used to implement immutable capabilities that describe function entry points. Such capabilities can be leveraged to establish a form of control-flow integrity between mutually distrusting code. These capabilities are known as sealed entry (sentry) capabilities. A program may jump to a sentry capability to begin executing the instructions it references. The jump instruction automatically unseals the capability and installs it to the program counter capability (see [Section 3.2](#)). The [JALR](#) instruction also seals the return address capability (if any) since it is the entry

point to the caller function.

2.2.6. Bounds (EF, T, TE, B, BE)

Concept



CHERI v9 Note: The bounds mantissa width is different in MXLEN=32. Also, the old IE bit is renamed to Exponent Format (EF); the function of IE is the inverse of EF i.e. IE=0 has the same effect as EF=1.



CHERI v9 Note: The mantissa width for RV32 was increased to 10.



CHERI v9 Note: The sense of the exponent is reversed, so an encoded value of 0 represents CAP_MAX_E, and CAP_MAX_E represents 0 from the previous specification.

The bounds encode the base and top addresses that constrain memory accesses. The capability can be used to access any memory location A in the range $\text{base} \leq A < \text{top}$. The bounds are encoded in compressed format, so it is not possible to encode any arbitrary combination of base and top addresses. An invalid capability with tag cleared is produced when attempting to construct a capability that is not *representable* because its bounds cannot be correctly encoded. The bounds are decoded as described in [Section 2.1](#).

The bounds field has the following components:

- **T:** Value substituted into the capability's address to decode the top address
- **B:** Value substituted into the capability's address to decode the base address
- **E:** Exponent that determines the position at which B and T are substituted into the capability's address
- **EF:** Exponent format flag indicating the encoding for T, B and E
 - The exponent is stored in T and B if EF=0, so it is 'internal'
 - The exponent is zero if EF=1

The bit width of T and B are defined in terms of the mantissa width (MW) which is set depending on the value of MXLEN as shown in [Table 6](#).

Table 6. Mantissa width (MW) values depending on MXLEN

MXLEN	MW
32	10
64	14

The exponent E indicates the position of T and B within the capability's address as described in [Section 2.1](#). The bit width of the exponent (EW) is set depending on the value of MXLEN. The maximum value of the exponent is calculated as follows:

$$\text{CAP_MAX_E} = \text{MXLEN} - \text{MW} + 2$$

The possible values for EW and CAP_MAX_E are shown in [Table 7](#).

Table 7. Exponent widths and CAP_MAX_E depending on MXLEN

MXLEN	EW	CAP_MAX_E
32	5	24
64	6	52



The address and bounds must be representable in valid capabilities i.e. when the tag is set (see [Section 2.2.6.3](#)).

Decoding

The metadata is encoded in a compressed format ([Woodruff et al., 2019](#)). It uses a floating point representation to encode the bounds relative to the capability address. The base and top addresses from the bounds are decoded as shown below.



TODO: The pseudo-code below does not have a formal notation. It is simply a place-holder while the Sail implementation is available. In this notation, / means "integer division", [] are the bit-select operators, and arithmetic is signed.



CHERI v9 Note: The IE bit from CHERI v9 is renamed EF and its value is inverted to ensure that the [NULL](#) capability is encoded as zero without the need for CHERI v9's in-memory format.

When EF=1, the exponent E=0, so the address bits $a[MW - 1:0]$ are replaced with T and B to form the top and base addresses respectively.

When EF=0, the exponent $E = CAP_MAX_E - ((MXLEN == 32) ? \{T8, TE, BE\} : \{TE, BE\})$, so the address bits $a[E + MW - 1:E]$ are replaced with T and B to form the top and base addresses respectively. E is computed by subtracting from the maximum possible exponent CAP_MAX_E which can be efficiently implemented in hardware assuming that T and B are at bit CAP_MAX_E and performing a logical bitwise shift right by E. In contrast, CHERI v9 implementations computed the top and base addresses by assuming that T and B are at bit 0 and performing a logical bitwise shift left by E.

```
EW      = (MXLEN == 32) ? 5 : 6
CAP_MAX_E = MXLEN - MW + 2

If EF = 1:
    E      = 0
    T[EW / 2 - 1:0] = TE
    B[EW / 2 - 1:0] = BE
    LCout   = (T[MW - 3:0] < B[MW - 3:0]) ? 1 : 0
    LMSB    = (MXLEN == 32) ? T8 : 0
else:
    E      = CAP_MAX_E - ((MXLEN == 32) ? {T8, TE, BE} : {TE, BE})
    T[EW / 2 - 1:0] = 0
    B[EW / 2 - 1:0] = 0
    LCout   = (T[MW - 3:EW / 2] < B[MW - 3:EW / 2]) ? 1 : 0
    LMSB    = 1
```

Reconstituting the top two bits of T:

```
T[MW - 1:MW - 2] = B[MW - 1:MW - 2] + LCout + LMSB
```

Decoding the bounds:

```
top:    t = { a[MXLEN - 1:E + MW] + ct, T[MW - 1:0] , {E{1'b0}} }
```

```
base:  b = { a[MXLEN - 1:E + MW] + cb, B[MW - 1:0] , {E{1'b0}} }
```

The corrections c_t and c_b are calculated as shown below using the definitions in [Table 8](#) and [Table 9](#).

```
A = a[E + MW - 1:E]
R = B - 2MW-2
```

Table 8. Calculation of top address correction

$A < R$	$T < R$	c_t
false	false	0
false	true	+1
true	false	-1
true	true	0

Table 9. Calculation of base address correction

$A < R$	$B < R$	c_b
false	false	0
false	true	+1
true	false	-1
true	true	0

The base, b , and top, t , addresses are derived from the address by substituting $a[E + MW - 1:E]$ with B and T respectively and clearing the lower E bits. The most significant bits of a may be adjusted up or down by 1 using corrections c_b and c_t to allow encoding memory regions that span alignment boundaries.

The EF bit selects between two cases:

1. EF = 1: The exponent is 0 for regions less than 2^{MW-2} bytes long
2. EF = 0: The exponent is *internal* with E stored in the lower bits of T and B along with T_8 when $MXLEN=32$. E is chosen so that the most significant non-zero bit of the length of the region aligns with $T[MW - 2]$ in the decoded top. Therefore, the most significant two bits of T can be derived from B using the equality $T = B + L$, where $L[MW - 2]$ is known from the values of EF and E and a carry out is implied if $T[MW - 3:0] < B[MW - 3:0]$ since it is guaranteed that the top is larger than the base.

The compressed bounds encoding allows the address to roam over a large *representable* region while maintaining the original bounds. This is enabled by defining a lower boundary R from the out-of-bounds values that allows us to disambiguate the location of the bounds with respect to an out-of-bounds address. R is calculated relative to the base by subtracting 2^{MW-2} from B . If B , T or $a[E + MW - 1:E]$ is less than R , it is inferred that they lie in the 2^{E+MW} aligned region above R labelled $space_U$ in [Figure 3](#) and the corrections c_t and c_b are computed accordingly. The overall effect is that the address can roam $2^{E+MW}/4$ bytes below the base address and at least $2^{E+MW}/4$ bytes above the top address while still allowing the bounds to be correctly decoded.

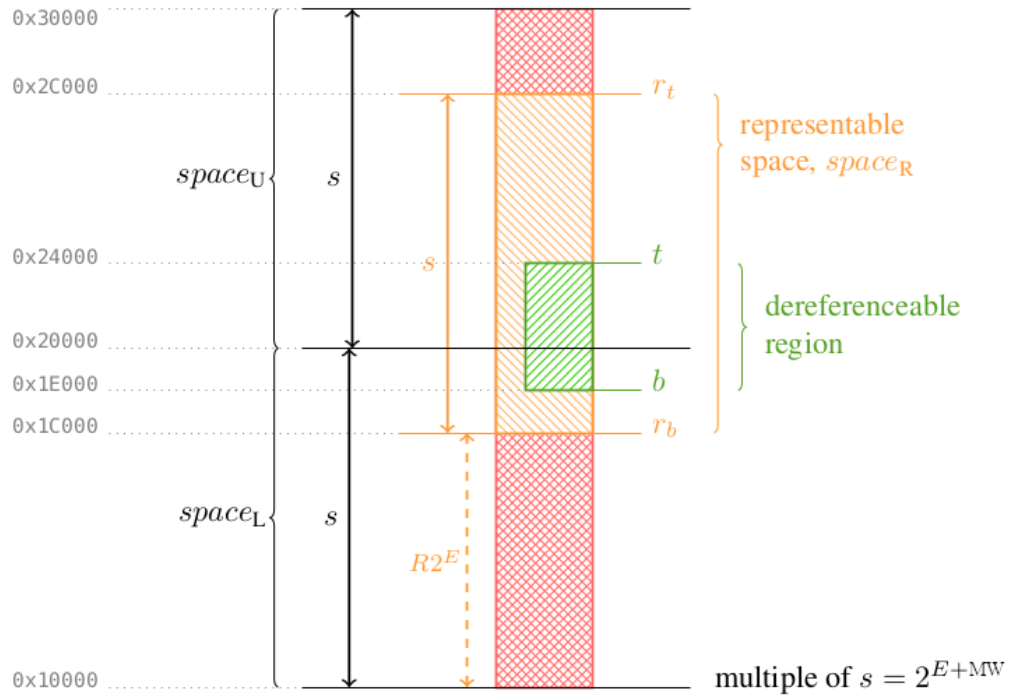


Figure 3. Memory address bounds encoded within a capability

A capability has *infinite* bounds if its bounds cover the entire address space such that the base address $b=0$ and the top address $t \geq 2^{\text{MXLEN}}$, i.e. t is a $\text{MXLEN} + 1$ bit value. However, b is a MXLEN bit value and the size mismatch introduces additional complications when decoding, so the following condition is required to correct t for capabilities whose [Representable Range](#) wraps the edge of the address space:

```
if ( (E < (CAP_MAX_E - 1)) & (t[MXLEN: MXLEN - 1] - b[MXLEN - 1] > 1) )
    t[MXLEN] = !t[MXLEN]
```

That is, invert the most significant bit of t if the decoded length of the capability is larger than E .



A capability has infinite bounds if $E=\text{CAP_MAX_E}$ and it is not malformed (see [Section 2.2.6.3](#)); this check is equivalent to $b=0$ and $t \geq 2^{\text{MXLEN}}$.

Malformed Capability Bounds

A capability is *malformed* if its encoding does not describe a valid capability because its bounds cannot be correctly decoded. The following check indicates whether a capability is malformed.

```
malformedMSB = (E == CAP_MAX_E && B != 0)
               || (E == CAP_MAX_E - 1 && B[MW - 1] != 0)
malformedLSB = (E < 0)
malformed    = !EF && (malformedMSB || malformedLSB)
```



The check is for malformed bounds, so it does not include reserved bits!

Capabilities with malformed bounds are always invalid anywhere in the system i.e. their tags are always 0.

2.3. Special Capabilities

2.3.1. NULL Capability



CHERI v9 Note: Encoding *NULL* as zeros removes the need for the difference between in-memory and architectural format.

The *NULL* capability is represented with 0 in all fields. This implies that it has no permissions and its exponent E is CAP_MAX_E (52 for $MXLEN=64$, 24 for $MXLEN=32$), so its bounds cover the entire address space such that the expanded base is 0 and top is 2^{MXLEN} .

Table 10. Field values of the *NULL* capability

Field	Value	Comment
Tag	zero	Capability is not valid
SDP	zeros	Grants no permissions
AP	zeros	Grants no permissions
S	zero	Unsealed
EF	zero	Internal exponent format
T_8	zeros	Top address bit ($MXLEN=32$ only)
T	zeros	Top address bits
T_E	zeros	Exponent bits
B	zeros	Base address bits
B_E	zeros	Exponent bits
Address	zeros	Capability address
Reserved	zeros	All reserved fields

2.3.2. Infinite Capability

The *Infinite* capability grants all permissions while its bounds also cover the whole address space.



The *Infinite* capability is also known as 'default', 'almighty', or 'root' capability.

Table 11. Field values of the *Infinite* capability

Field	Value	Comment
Tag	one	Capability is valid
SDP	ones	Grants all permissions
AP ($MXLEN=32$)	0x8 (see Table 3)	Grants all permissions
AP ($MXLEN=64$)	0x1F (see Table 4)	Grants all permissions
S	zero	Unsealed
EF	zero	Internal exponent format
T_8	zeros	Top address bit ($MXLEN=32$ only)
T	zeros	Top address bits

Field	Value	Comment
T _E	zeros	Exponent bits
B	zeros	Base address bits
B _E	zeros	Exponent bits
Address	zeros	Capability address
Reserved	zeros	All reserved fields



The AP fields both set Legacy Mode (if Zcheri_legacy is supported)

2.4. Representable Range Check

2.4.1. Concept

The new address, after updating the address of a capability, is within the *representable range* if decompressing the capability's bounds with the original and new addresses yields the same base and top addresses.

In other words, given a capability with address a and the new address $a' = a + x$, the bounds b and t are decoded using a and the new bounds b' and t' are decoded using a' . The new address is within the capability's *representable range* if $b == b' \ \&\& \ t == t'$.

Changing a capability's address to a value outside the *representable range* unconditionally clears the capability's tag. Examples are:

- Instructions such as [CADD](#) which include pointer arithmetic.
- The [SCADDR](#) instruction which updates the capability address field.

2.4.2. Practical Information

In the bounds encoding in this specification, the top and bottom capability bounds are formed of two or three sections:

- Upper bits from the address
- Middle bits from T and B decoded from the metadata
- Lower bits are set to zero
 - This is only if there is an internal exponent (EF=0)

Table 12. Composition of address bounds

Configuration	Upper section	Middle Section	Lower section
EF=0	address[MXLEN-1:E + MW] + ct	T[MW - 1:0]	{E{1'b0}}
EF=1, i.e. E=0	address[MXLEN-1:MW] + ct	T[MW - 1:0]	

The *representable range* defines the range of addresses which do not corrupt the bounds encoding. The encoding was first introduced in [Section 2.1](#), and is repeated in a different form in [Table 12](#) to aid this

description.

For the address to be valid for the current bounds encoding, the value in the *Upper Section* of [Table 12](#) *must not change* as this will change the meaning of the bounds.

This gives a range of $s=2^{E+MW}$, which as shown in [Figure 3](#).

The gap between the object bounds and the bound of the representable range is always guaranteed to be at least 1/4 of s . This is represented by $R = B - 2^{MW-2}$ in [Section 2.1](#). This gives useful guarantees, such that if an executed instruction is in [pcc](#) bounds, then it is also guaranteed that the next linear instruction is *representable*.

Chapter 3. Integrating Zcheri_purecap with the RISC-V Base Integer Instruction Set

Zcheri_purecap is an extension to the RISC-V ISA. The extension adds a carefully selected set of instructions and CSRs that are sufficient to implement new security features in the ISA. To ensure compatibility, Zcheri_purecap also requires some changes to the primary base integer variants: RV32I, providing 32-bit addresses with 64-bit capabilities, and RV64I, providing 64-bit addresses with 128-bit capabilities. The remainder of this chapter describes these changes for both the unprivileged and privileged components of the base integer RISC-V ISAs.



The changes described in this specification also ensure that Zcheri_purecap is compatible with RV32E.



RV128 is not currently supported by any CHERI extension.

3.1. Memory

A hart supporting Zcheri_purecap has a single byte-addressable address space of 2^{XLEN} bytes for all memory accesses. Each memory region capable of holding a capability also stores a tag bit for each naturally aligned CLEN bits (e.g. 16 bytes in RV64), so that capabilities with their tag set can only be stored in naturally aligned addresses. Tags must be atomically bound to the data they protect.

The memory address space is circular, so the byte at address $2^{\text{XLEN}} - 1$ is adjacent to the byte at address zero. A capability's [Representable Range](#) described in [Section 2.1](#) is also circular, so address 0 is within the [Representable Range](#) of a capability where address $2^{\text{MXLEN}} - 1$ is within the bounds.

3.2. Programmer's Model for Zcheri_purecap

For Zcheri_purecap, the 32 unprivileged **x** registers of the base integer ISA are extended so that they are able to hold a capability as well as renamed to **c** registers. Therefore, each **c** register is CLEN bits wide and has an out-of-band tag bit. The **x** notation refers to the address field of the capability in an unprivileged register while the **c** notation is used to refer to the full capability (i.e. address, metadata and tag) held in the same unprivileged register.

The tag of the unprivileged **c** registers must be reset to zero. The reset values of the metadata and address fields are UNSPECIFIED for all unprivileged **c** registers except **c0**.

Register **c0** is hardwired with all bits, including the capability metadata and tag, equal to 0. In other words, **c0** is hardwired to the [NULL](#) capability.

3.2.1. PCC - The Program Counter Capability

An authorising capability with appropriate permissions is required to execute instructions in Zcheri_purecap. Therefore, the unprivileged program counter (**pc**) register is extended so that it is able to hold a capability. The extended register is called the program counter capability (**pcc**). The **pcc** address field is effectively the **pc** in the base RISC-V ISA so that the hardware automatically increments as instructions are executed. The **pcc**'s metadata and tag are reset to the [Infinite](#) capability

metadata and tag with the address field set to the core boot address.

The hardware performs the following checks on `pcc` for each instruction executed in addition to the checks already required by the base RISC-V ISA. A failing check causes a CHERI exception.

- The tag must be set
- The capability must not be sealed
- The capability must grant execute permission
- All bytes of the instruction must be in bounds



Operations that update `pcc`, such as changing privilege or executing jump instructions, unseal capabilities prior to writing. Therefore, implementations do not need to check that that `pcc` is unsealed when executing each instruction. However, this property has not yet been formally verified and may not hold if additional CHERI extensions beyond `Zcheri_purecap` are implemented.



It is common for implementations to not allow executing `pc` relative instructions, such as `AUIPC` or `JAL`, in debug mode.

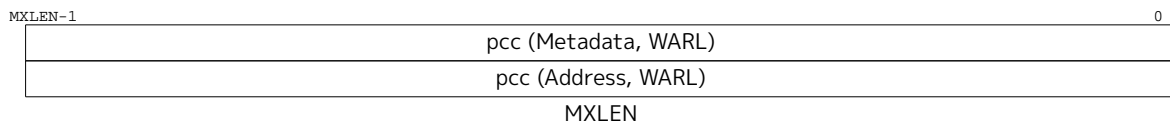


Figure 4. Program Counter Capability

`pcc` is an executable vector, so it need not be able to hold all possible invalid addresses.

3.3. Capability Instructions



CHERI v9 Note: Some instructions from the original CHERI specification were removed to save encoding space, or because they relate to features which are not yet in this specification. Instructions were removed if they do not harm performance and can be emulated using other instructions.

`Zcheri_purecap` introduces new instructions to the base RISC-V integer ISA to inspect and operate on capabilities held in registers.

3.3.1. Capability Inspection Instructions

These instructions allow software to inspect the fields of a capability held in a `c` register. The output is an integer value written to an `x` register representing the decoded field of the capability, such as the permissions or bounds. These instructions do not cause exceptions.

- `GCTAG`: inspects the tag of the input capability. The output is 1 if the tag is set and 0 otherwise
- `GCPERM`: outputs the architectural (AP) and software-defined (SDP) permission fields of the input capability
- `GCBASE`: outputs the expanded base address of the input capability
- `GCLLEN`: outputs the length of the input capability. Length is defined as `top - base`. The output is $2^{\text{MXLEN}} - 1$ when the capability's length is 2^{MXLEN}
- `CRAM`: outputs the nearest bounds alignment that a valid capability can represent

- **GCHI**: outputs the compressed capability metadata
- **SCEQ**: compares two capabilities including tag, metadata and address
- **SCSS**: tests whether the bounds and permissions of a capability are a subset of those from another capability



***GCBASE** and **GCLEN** output 0 when a capability with malformed bounds is provided as an input (see [Section 2.2.6.3](#)).*

3.3.2. Capability Manipulation Instructions

These instructions allow software to manipulate the fields of a capability held in a **c** register. The output is a capability written to a **c** register with its fields modified. The output capability has its tag set to 0 if the input capability did not have a tag set, the output capability has more permissions or larger bounds compared to the input capability, or the operation results in a capability with malformed bounds. These instructions do not give rise to exceptions.

- **SCADDR**: set the address of a capability to an arbitrary address
- **CADD**, **CADDI**: increment the address of the input capability by an arbitrary offset
- **SCHI**: replace a capability's metadata with an arbitrary value. The output tag is always 0
- **ACPERM**: bitwise AND of a mask value with a bit map representation of the architectural (AP) and software-defined (SDP) permissions fields
- **SCBNDS**: set the base and length of a capability. The tag is cleared, if the encoding cannot represent the bounds exactly
- **SCBNDSR**: set the base and length of a capability. The base will be rounded down and/or the length will be rounded up if the encoding cannot represent the bounds exactly
- **SENTRY**: seal capability as a sentry capability
- **CBLD**: replace the base, top, address, permissions and mode fields of a capability with the fields from another capability
- **CMV**: move a capability from a **c** register to another **c** register



***CBLD** outputs a capability with tag set to 0 if the input capability's bounds are malformed.*



***CHERI v9 Note:** **SCBNDS** and **SCBNDSI** perform the role of the old **CSETBOUNDSEXACT** while the **SCBNDSR** is the old **CSETBOUNDS**.*

3.3.3. Capability Load and Store Instructions

A load capability instruction, **LC**, reads **CLEN** bits from memory together with its tag and writes the result to a **c** register. The capability authorising the memory access is provided in a **c** source register, so the effective address is obtained by incrementing that capability with the sign-extended 12-bit offset.

A store capability instruction, **SC**, writes **CLEN** bits and the tag in a **c** register to memory. The capability authorising the memory access is provided in a **c** source register, so the effective address is obtained by incrementing that capability with the sign-extended 12-bit offset.

LC and **SC** instructions cause **CHERI** exceptions if the authorising capability fails any of the following checks:

- The tag is zero
- The capability is sealed
- At least one byte of the memory access is outside the capability's bounds
- For loads, the read permission must be set in AP
- For stores, the write permission must be set in AP

Capability load and store instructions also cause load or store/AMO address misaligned exceptions if the address is not naturally aligned to a CLEN boundary.

Misaligned capability loads and stores are errors. Implementations must generate exceptions for misaligned capability loads and stores even if they allow misaligned integer loads and stores to complete normally. Execution environments must report misaligned capability loads and stores as errors and not attempt to emulate them using byte access. The Zicclsm extension does not affect capability loads and stores. Software which uses capability loads and stores to copy data other than capabilities must ensure that addresses are aligned.



Since there is only one tag per aligned CLEN bit block in memory, it is not possible to represent a capability value complete with its tag at an address not aligned to CLEN. Therefore, [LC](#) and [SC](#) give rise to misaligned address fault exceptions when the effective address to access is misaligned, even if the implementation supports Zicclsm. To transfer CLEN misaligned bits without a tag, use integer loads and stores.

For loads, the tag of the capability loaded from memory is cleared if the authorising capability does not grant permission to read capabilities (i.e. both [R-permission](#) and [C-permission](#) must be set in AP). For stores, the tag of the capability written to memory is cleared if the authorising capability does not grant permission to write capabilities (i.e. both [W-permission](#) and [C-permission](#) must be set in AP).



TODO: these cases may cause exceptions in the future - we need a way for software to discover and/or control the behaviour

3.4. Existing RISC-V Instructions

The operands or behavior of some instructions in the base RISC-V ISA changes in Zcheri_purecap.

3.4.1. Integer Computational Instructions

Most integer computational instructions operate on XLEN bits of values held in **x** registers. Therefore, these instructions only operate on the address field if the input register of the instruction holds a capability. The output is XLEN bits written to an **x** register; the tag and capability metadata of that register are zeroed.

The add upper immediate to [pcc](#) instruction ([AUIPC](#)) is used to build [pcc](#)-relative capabilities. [AUIPC](#) forms a 32-bit offset from the 20-bit immediate and filling the lowest 12 bits with zeros. The [pcc](#) address is then incremented by the offset and a representability check is performed so the capability's tag is cleared if the new address is outside the [pcc](#)'s [Representable Range](#). The resulting CLEN value along with the new tag are written to a **c** register.

3.4.2. Control Transfer Instructions

Control transfer instructions operate as described in the base RISC-V ISA. They also may cause

metadata updates and/or cause exceptions in addition to the base behaviour as described below.

Unconditional Jumps

JAL sign-extends the offset and adds it to the address of the jump instruction to form the target address. The target address is installed in the address field of **pcc**. The capability with the address of the instruction following the jump (**pcc** + 4) is written to a **c** register.

JALR allows unconditional jumps to a target capability. The target capability is obtained by incrementing the capability in the **c** register operand by the sign-extended 12-bit immediate if the immediate is not zero, then setting the least significant bit of the result to zero, then unsealing. The capability with the address of the instruction following the jump (**pcc** + 4) is sealed and written to a **c** register.

All jumps cause CHERI exceptions when a minimum sized instruction at the target address is not within the bounds of the **pcc**.

JALR causes a CHERI exception when:

- The target capability's tag is zero
- The target capability is sealed and the immediate is not zero
- A minimum sized instruction at the target capability's address is not within bounds
- The target capability does not grant execute permission

JAL and **JALR** can also cause instruction address misaligned exceptions following the standard RISC-V rules.

Conditional Branches

Branch instructions (see [Conditional branches \(BEQ, BNE, BLT\[U\], BGE\[U\]\)](#)) encode signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to form the target address.

Branch instructions compare two **x** registers as described in the base RISC-V ISA, so the metadata and tag values are disregarded in the comparison if the operand registers hold capabilities. If the comparison evaluates to true, then the target address is installed in the **pcc**'s address field. These instructions cause CHERI exceptions when a minimum sized instruction at the target address is not within the **pcc**'s bounds.

3.4.3. Integer Load and Store Instructions

Integer load and store instructions transfer the amount of integer data described in the base RISC-V ISA between the registers and memory. For example, **LD** and **LW** load 64-bit and 32-bit values respectively from memory into an **x** register. However, the address operands for load and store instructions are interpreted differently in Zcheri_purecap: the capability authorising the access is in the **c** register operand and the memory address is given by incrementing the address of that capability by the sign-extended 12-bit immediate offset.

All load and store instructions cause CHERI exceptions if the authorising capability fails any of the following checks:

- The tag is set

- The capability is unsealed
- All bytes of accessed memory are inside the capability's bounds
- For loads, the read permission must be set in AP
- For stores, the write permission must be set in AP

Integer load instructions always zero the tag and metadata of the result register.

Integer stores write zero to the tag associated with the memory locations that are naturally aligned to CLEN. Therefore, misaligned stores may clear up to two tag bits in memory.

3.5. Zicsr, Control and Status Register (CSR) Instructions

Zcheri_purecap requires that RISC-V CSRs intended to hold addresses, like [mtvec](#), are now able to hold capabilities. Therefore, such registers are renamed and extended to CLEN-bit in Zcheri_purecap.

Reading or writing any part of a CLEN-bit CSR may cause side effects. For example, the CSR's tag bit may be cleared if a new address is outside the [Representable Range](#) of a CSR capability being written.

This section describes how the CSR instructions operate on these CSRs in Zcheri_purecap.

The CLEN-bit CSRs are summarised in [Chapter 9](#).

3.5.1. CSR Instructions



CHERI v9 Note: *CSpecialRW is removed. Its role is assumed by [CSRRW](#).*

All CSR instructions atomically read-modify-write a single CSR. If the CSR accessed is of capability size then the capability's tag, metadata and address are all accessed atomically.

When the [CSRRW](#) instruction is accessing a capability width CSR, then the source and destination operands are c registers and it atomically swaps the values in the whole CSR with the CLEN width register operand.

There are special rules for updating specific CLEN-wide CSRs as shown in [Table 36](#).

When [CSRRS](#) and [CSRRC](#) instructions are accessing a capability width CSR, such as [mtvecc](#), then the destination operand is a c register and the source operand is an x register. Therefore, the instructions atomically read CLEN bits from the CSR, calculate the final address using standard RISC-V behaviour (set bits, clear bits, etc.), and that final address is written to the CSR capability's address field. The update typically uses the semantics of a [SCADDR](#) instruction which clears the tag if the capability is sealed, or if the updated address is not representable. [Table 36](#) shows the exact action taken for each capability width CSR.

The [CSRRWI](#), [CSRRSI](#) and [CSRRCI](#) variants are similar to [CSRRW](#), [CSRRS](#), and [CSRRC](#) respectively, when accessing a capability width CSR except that they update the capability's address only using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate field.

All CSR instructions cause CHERI exceptions if the [pcc](#) does not grant [ASR-permission](#) and the CSR accessed is privileged.

3.6. Control and Status Registers (CSRs)

Zcheri_purecap extends the CSRs listed in [Table 13](#), [Table 14](#), [Table 15](#) and [Table 16](#) from the base RISC-V ISA and its extensions. The CSRs are renamed to reflect the fact that they are extended to CLLEN+1 bits wide, as the **x** registers are renamed to **c** registers.

Table 13. Renamed debug-mode CSRs in Zcheri_purecap

Zcheri_purecap CSR	Addresses	Extended CSR	Prerequisites	Permissions	Description
dpcc	0x7b1	dpc	Sdext	DRW	Debug Program Counter Capability
dscratch0c	0x7b2	dscratch0	Sdext	DRW	Debug Scratch Capability 0
dscratch1c	0x7b3	dscratch1	Sdext	DRW	Debug Scratch Capability 1

Table 14. Renamed machine-mode CSRs in Zcheri_purecap

Zcheri_purecap CSR	Address	Extended CSR	Prerequisites	Permissions	Description
mtvecc	0x305	mtvec	M-mode	MRW, ASR-permission	Machine Trap-Vector Base-Address Capability
mscratchc	0x340	mscratch	M-mode	MRW, ASR-permission	Machine Scratch Capability
mepcc	0x341	mepc	M-mode	MRW, ASR-permission	Machine Exception Program Counter Capability

Table 15. Renamed supervisor-mode CSRs in Zcheri_purecap

Zcheri_purecap CSR	Address	Extended CSR	Prerequisites	Permissions	Description
stvecc	0x105	stvec	S-mode	SRW, ASR-permission	Supervisor Trap-Vector Base-Address Capability
sscratchc	0x140	sscratch	S-mode	SRW, ASR-permission	Supervisor Scratch Capability
sepc	0x141	sepc	S-mode	SRW, ASR-permission	Supervisor Exception Program Counter Capability

Table 16. Renamed user-mode CSRs in Zcheri_purecap

Zcheri_purecap CSR	Address	Extended CSR	Prerequisite	Permission	Description
jvtc	0x017	jvt	Zcmt	URW	Jump Vector Table Capability

3.7. Machine-Level CSRs

Zcheri_purecap extends some M-mode CSRs to hold capabilities or otherwise add new functions. [pcc](#) must grant [ASR-permission](#) to access M-mode CSRs regardless of the RISC-V privilege mode.

3.7.1. Machine Status Registers (mstatus and mstatush)

The **mstatus** and **mstatush** registers operate as described in (RISC-V, 2023) except for the SXL and UXL fields that control the value of XLEN for S-mode and U-mode, respectively, and the MBE, SBE, and UBE fields that control the memory system endianness for M-mode, S-mode, and U-mode, respectively.

The encoding of the SXL and UXL fields is the same as the MXL field of **misa**. Only 1 and 2 are supported values for SXL and UXL and the fields must be read-only in implementations supporting Zcheri_purecap. The effective XLEN in S-mode and U-mode are termed SXLEN and UXLEN, respectively.

The MBE, SBE, and UBE fields determine whether explicit loads and stores performed from M-mode, S-mode, or U-mode, respectively, are little endian (xBE = 0) or big endian (xBE = 1). MBE must be read only. SBE and UBE must be read only and equal to MBE, if S-mode or U-mode, respectively, is implemented, or read only zero otherwise.



A further CHERI extension, Zcheri_legacy, optionally makes SXL, UXL, MBE, SBE, and UBE writeable, so implementations that support multiple base ISAs must support both Zcheri_purecap and Zcheri_legacy.

3.7.2. Machine Trap-Vector Base-Address Registers (mtvec)

The **mtvec** register is as defined in (RISC-V, 2023). It is an MXLEN-bit register used as the executable vector jumped to when taking traps into machine mode. It is extended into **mtvecc**.



Figure 5. Machine-mode trap-vector base-address register

3.7.3. Machine Trap-Vector Base-Address Capability Registers (mtvecc)

The **mtvecc** register is a renamed extension of **mtvec** that holds a capability. Its reset value is the **Infinite** capability. The capability represents an executable vector.

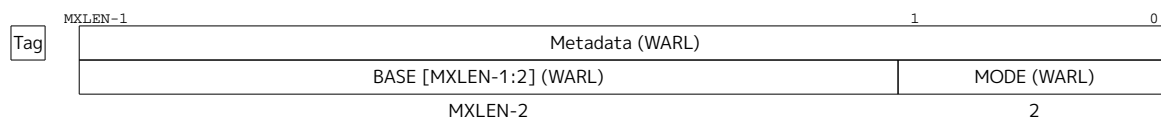


Figure 6. Machine-mode trap-vector base-capability register

The metadata is WARL as not all fields need to be implemented, for example the reserved fields will always read as zero.

When interpreting **mtvecc** as a capability, as for **mtvec**, address bits [1:0] are always zero (as they are reused by the MODE field).

When MODE=Vectored, all synchronous exceptions into machine mode cause the **pcc** to be set to the capability, whereas interrupts cause the **pcc** to be set to the capability with its address incremented by four times the interrupt cause number.

Capabilities written to **mtvecc** also include writing the MODE field in **mtvecc.address[1:0]**. As a result, a representability and sealing check is performed on the capability with the legalized (WARL) MODE

field included in the address. The tag of the capability written to `mtvecc` is cleared if either check fails.

Additionally, when `MODE=Vectored` the capability has its tag bit cleared if the capability address + 4 x `HICAUSE` is not within the representable bounds. `HICAUSE` is the largest exception cause value that the implementation can write to `mcause` when an interrupt is taken.



When `MODE=Vectored`, it is only required that `address + 4 x HICAUSE` is within representable bounds instead of the capability's bounds. This ensures that software is not forced to allocate a capability granting access to more memory for the trap-vector than necessary to handle the trap causes that actually occur in the system.

3.7.4. Machine Scratch Register (`mscratch`)

The `mscratch` register is as defined in (RISC-V, 2023). It is an `MXLEN`-bit read/write register dedicated for use by machine mode. Typically, it is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler. `mscratch` is extended into `mscratchc`.



Figure 7. Machine-mode scratch register

3.7.5. Machine Scratch Register Capability (`mscratchc`)

The `mscratchc` register is a renamed extension of `mscratch` that is able to hold a capability.

The tag of the CSR must be reset to zero. The reset values of the metadata and address fields are UNSPECIFIED.

It is not WARL, all capability fields must be implemented.

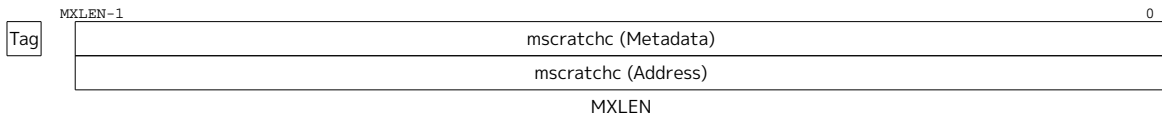


Figure 8. Machine-mode scratch capability register

3.7.6. Machine Exception Program Counter (`mepc`)

The `mepc` register is as defined in (RISC-V, 2023). It is extended into `mepcc`.

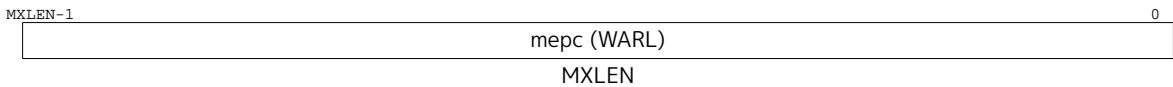


Figure 9. Machine exception program counter register

3.7.7. Machine Exception Program Counter Capability (`mepcc`)

The `mepcc` register is a renamed extension of `mepc` that is able to hold a capability. Its reset value is the `Infinite` capability.

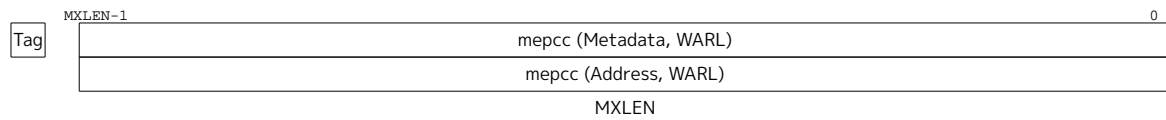


Figure 10. Machine exception program counter capability register

Capabilities written to `mepcc` must be legalised by implicitly zeroing bit `mepcc[0]`. Additionally, if an implementation allows `IALIGN` to be either 16 or 32, then whenever `IALIGN=32`, the capability read from `mepcc` must be legalised by implicitly zeroing bit `mepcc[1]`. Therefore, the capability read or written has its tag bit cleared if the legalised address is not within the [Representable Range](#).



When reading or writing a sealed capability in `mepcc`, the tag is not cleared if the original address equals the legalized address.

When a trap is taken into M-mode, `mepcc` is written with the `pcc` including the virtual address of the instruction that was interrupted or that encountered an exception. Otherwise, `mepcc` is never written by the implementation, though it may be explicitly written by software.

As shown in [Table 37](#), `mepcc` is an executable vector, so it does not need to be able to hold all possible invalid addresses. Additionally, the capability in `mepcc` is unsealed when it is installed in `pcc` on execution of an `MRET` instruction.

3.7.8. Machine Cause Register (mcause)

Zcheri_purecap adds a new exception code for CHERI exceptions that `mcause` must be able to represent. The new exception code and its priority are listed in [Table 17](#) and [Table 18](#) respectively. The behavior and usage of `mcause` otherwise remains as described in ([RISC-V, 2023](#)).

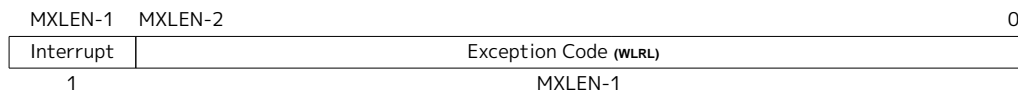


Figure 11. Machine cause register

Table 17. Machine cause register (`mcause`) values after trap. Entries added in Zcheri_purecap are in **bold**

Interrupt	Exception Code	Description
1	0	<i>Reserved</i>
1	1	Supervisor software interrupt
1	2	<i>Reserved</i>
1	3	Machine software interrupt
1	4	<i>Reserved</i>
1	5	Supervisor timer interrupt
1	6	<i>Reserved</i>
1	7	Machine timer interrupt
1	8	<i>Reserved</i>
1	9	Supervisor external interrupt
1	10	<i>Reserved</i>
1	11	Machine external interrupt
1	12-15	<i>Reserved</i>
1	≥16	<i>Designated for platform use</i>

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	<i>Reserved</i>
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	<i>Reserved</i>
0	15	Store/AMO page fault
0	16-23	<i>Reserved</i>
0	24-27	<i>Designated for custom use</i>
0	28	CHERI fault
0	29-31	<i>Designated for custom use</i>
0	32-47	<i>Reserved</i>
0	48-63	<i>Designated for custom use</i>
	≥64	<i>Reserved</i>

Table 18. Synchronous exception priority in decreasing priority order. Entries added in Zcheri_purecap are in **bold**

Priority	Exc.Code	Description
<i>Highest</i>	3	Instruction address breakpoint
	28	Prior to instruction address translation: CHERI fault due to PCC checks (tag, execute permission, invalid address and bounds)
	12, 1	During instruction address translation: First encountered page fault or access fault
	1	With physical address for instruction: Instruction access fault
	2 0 8,9,11 3 3	Illegal instruction Instruction address misaligned Environment call Environment break Load/store/AMO address breakpoint
	28	CHERI faults due to: PCC ASR-permission clear Branch/jump target address checks (tag, execute permissions, invalid address and bounds)

Priorit y	Exc.Cod e	Description
	28	Prior to address translation for an explicit memory access: Load/store/AMO capability address misaligned CHERI fault due to capability checks (tag, permissions, invalid address and bounds)
	4,6	Optionally: Load/store/AMO address misaligned
	13, 15, 5, 7	During address translation for an explicit memory access: First encountered page fault or access fault
	5,7	With physical address for an explicit memory access: Load/store/AMO access fault
Lowest	4,6	If not higher priority: Load/store/AMO address misaligned

 The full details of the CHERI exceptions are in [Table 22](#).

3.7.9. Machine Trap Delegation Register (medeleg)

Bit 28 of [medeleg](#) now refers to a valid exception and so can be used to delegate CHERI exceptions to supervisor mode.

3.7.10. Machine Trap Value Register (mtval)

 **CHERI v9 Note:** Encoding and values changed, and generally were simplified.

The [mtval](#) register is an MXLEN-bit read-write register. When a CHERI fault is taken into M-mode, [mtval](#) is written with additional CHERI-specific exception information with the format shown in [Figure 12](#) to assist software in handling the trap.

If the hardware platform specifies that no exceptions set [mtval](#) to a nonzero value, then [mtval](#) is read-only zero.

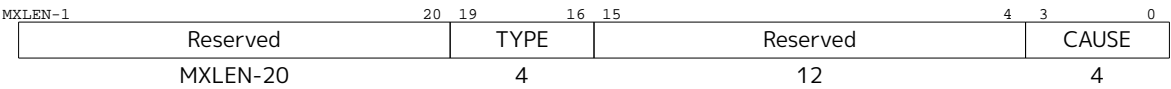


Figure 12. Machine trap value register

TYPE is a CHERI-specific fault type that caused the exception while CAUSE is the cause of the fault. The possible CHERI types and causes are encoded as shown in [Table 19](#) and [Table 20](#) respectively.

Table 19. Encoding of TYPE field

CHERI Type Code	Description
0	CHERI instruction access fault
1	CHERI data fault due to load, store or AMO
2	CHERI jump or branch fault
3-15	Reserved

Table 20. Encoding of CAUSE field

CHERI Cause Code	Description
0	Tag violation
1	Seal violation
2	Permission violation
3	Invalid address violation
4	Length violation
5-15	Reserved

CHERI violations have the following order in priority:

1. Tag violation (*Highest*)
2. Seal violation
3. Permission violation
4. Invalid address violation
5. Length violation (*Lowest*)

3.8. Supervisor-Level CSRs

Zcheri_purecap extends some of the existing RISC-V CSRs to be able to hold capabilities or with other new functions. `pcc` must grant [ASR-permission](#) to access S-mode CSRs regardless of the RISC-V privilege mode.

3.8.1. Supervisor Trap Vector Base Address Registers (stvec)

The `stvec` register is as defined in ([RISC-V, 2023](#)). It is an SXLEN-bit register used as the executable vector jumped to when taking traps into supervisor mode. It is extended into `stvecc`.

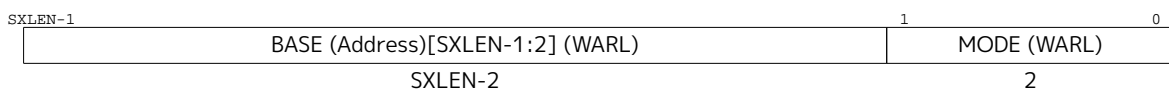


Figure 13. Supervisor trap-vector base-address register

3.8.2. Supervisor Trap Vector Base Address Registers (stvecc)

The `stvec` register is an SXLEN-bit WARL read/write register that holds the trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE). The `stvecc` register is a renamed extension of `stvec` that is able to hold a capability. Its reset value is the [Infinite](#) capability.

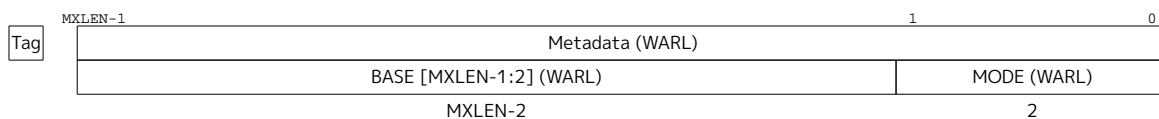


Figure 14. Supervisor trap-vector base-capability register

The handling of `stvecc` is otherwise identical to `mtvecc`, but in supervisor mode.

3.8.3. Supervisor Scratch Register (sscratch)

The [sscratch](#) register is as defined in ([RISC-V, 2023](#)). It is an MXLEN-bit read/write register dedicated for use by supervisor mode. Typically, it is used to hold a pointer to a supervisor-mode hart-local context space and swapped with a user register upon entry to an S-mode trap handler. [sscratch](#) is extended into [sscratchc](#).

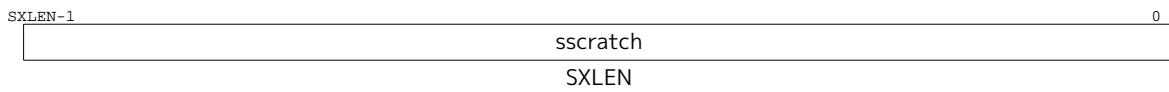


Figure 15. Supervisor-mode scratch register

3.8.4. Supervisor Scratch Registers (sscratchc)

The [sscratchc](#) register is a renamed extension of [sscratch](#) that is able to hold a capability.

The tag of the CSR must be reset to zero. The reset values of the metadata and address fields are UNSPECIFIED.

It is not WARL, all capability fields must be implemented.

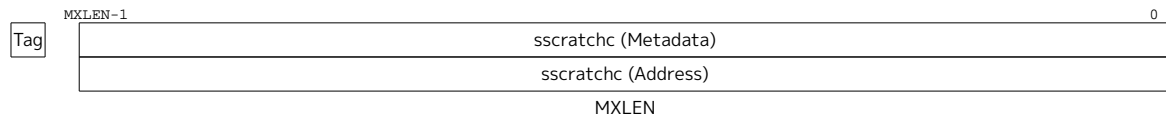


Figure 16. Supervisor scratch capability register

3.8.5. Supervisor Exception Program Counter (sepc)

The [sepc](#) register is as defined in ([RISC-V, 2023](#)). It is extended into [sepcc](#).

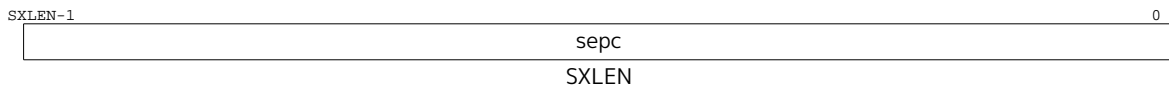


Figure 17. Supervisor exception program counter register

3.8.6. Supervisor Exception Program Counter Capability (sepcc)

The [sepcc](#) register is a renamed extension of [sepc](#) that is able to hold a capability. Its reset value is the [Infinite](#) capability.

As shown in [Table 37](#), [sepcc](#) is an executable vector, so it need not be able to hold all possible invalid addresses. Additionally, the capability in [sepcc](#) is unsealed when it is installed in [pcc](#) on execution of an [SRET](#) instruction. The handling of [sepcc](#) is otherwise identical to [mepcc](#), but in supervisor mode.

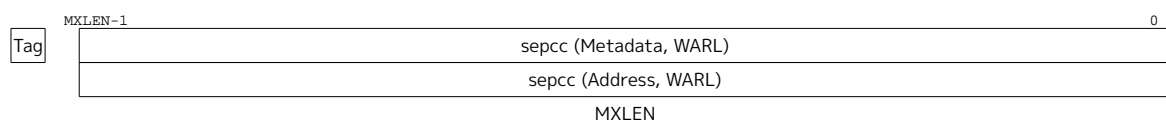


Figure 18. Supervisor exception program counter capability register

3.8.7. Supervisor Cause Register (scause)

Zcheri_purecap adds a new exception code for CHERI exceptions that [scause](#) must be able to represent. The new exception code and its priority are listed in [Table 21](#) and [Table 18](#) respectively. The

behavior and usage of [scause](#) otherwise remains as described in ([RISC-V, 2023](#)).

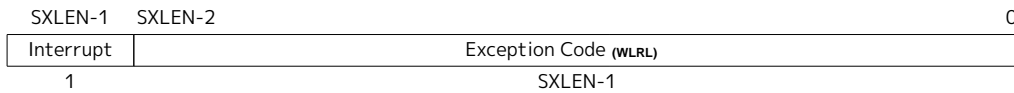


Figure 19. Supervisor cause register

Table 21. Supervisor cause register (*scause*) values after trap. Causes added in Zcheri_purecap are in **bold**

Interrupt	Exception Code	Description
1	0	<i>Reserved</i>
1	1	Supervisor software interrupt
1	2-4	<i>Reserved</i>
1	5	Supervisor timer interrupt
1	6-8	<i>Reserved</i>
1	9	Supervisor external interrupt
1	10-15	<i>Reserved</i>
1	≥16	<i>Designated for platform use</i>
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10-11	<i>Reserved</i>
0	12	Instruction page fault
0	13	Load page fault
0	14	<i>Reserved</i>
0	15	Store/AMO page fault
0	16-23	<i>Reserved</i>
0	24-27	<i>Designated for custom use</i>
0	28	CHERI fault
0	29-31	<i>Designated for custom use</i>
0	32-47	<i>Reserved</i>
0	48-63	<i>Designated for custom use</i>
	≥64	<i>Reserved</i>

3.8.8. Supervisor Trap Value Register (stval)

The [stval](#) register is an SXLEN-bit read-write register. When a CHERI fault is taken into S-mode, [stval](#) is written with additional CHERI-specific exception information with the format shown in [Figure 20](#) to assist software in handling the trap.

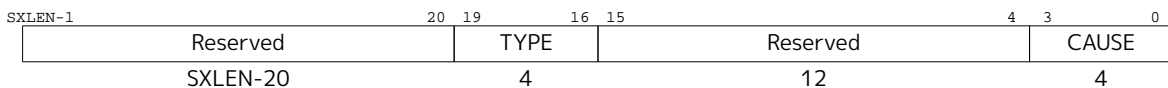


Figure 20. Supervisor trap value register

TYPE is a CHERI-specific fault type that caused the exception while CAUSE is the cause of the fault. The possible CHERI types and causes are encoded as shown in [Table 19](#) and [Table 20](#) respectively.

3.9. Unprivileged CSRs

Unlike machine and supervisor level CSRs, Zcheri_purecap does not require [pcc](#) to grant [ASR-permission](#) to access unprivileged CSRs.

3.10. CHERI Exception handling



`auth_cap` is [ddc](#) for Legacy Mode and `cs1` for Capability Mode

Table 22. Valid CHERI exception combination description

Instructions	Xcause	Xtval. TYPE	Xtval. CAUSE	Description	Check
All instructions have these exception checks first					
All	28	0	0	pcc tag	not(pcc.tag)
All	28	0	1	pcc seal	isCapSealed(pcc) ¹
All	28	0	2	pcc permission	not(pcc.X-permission)
All	28	0	3	pcc length	Any byte of current instruction out of pcc bounds
CSR/Xret additional exception check					
CSR*, MRET , SRET	28	0	2	pcc permission	not(pcc.ASR-permission) when required for CSR access or execution of MRET / SRET
direct jumps additional exception check					
JAL , Conditional branches (BEQ , BNE , BLT[U] , BGE[U])	28	2	3	pcc length	any byte of minimum length instruction at target out of pcc bounds
indirect jumps additional exception checks					
indirect jumps	28	2	0	<code>cs1</code> tag	not(<code>cs1.tag</code>)
indirect jumps	28	2	1	<code>cs1</code> seal	isCapSealed(<code>cs1</code>) and imm12 != 0
indirect jumps	28	2	2	<code>cs1</code> permission	not(<code>cs1.X-permission</code>)
indirect jumps	28	2	3	<code>cs1</code> length	any byte of minimum length instruction at target out of <code>cs1</code> bounds
Load additional exception checks					
all loads	28	1	0	<code>auth_cap</code> tag	not(<code>auth_cap.tag</code>)
all loads	28	1	1	<code>auth_cap</code> seal	isCapSealed(<code>auth_cap</code>)

Instructions	Xcause	Xtval. TYPE	Xtval. CAUSE	Description	Check
all loads	28	1	2	auth_cap permission	not(auth_cap . R-permission)
all loads	28	1	3	auth_cap length	Any byte of load access out of auth_cap bounds
capability loads	4	N/A	N/A	load address misaligned	Misaligned capability load
Store/atomic/cache-block-operation additional exception checks					
all stores, all atomics, all cbos	28	1	0	auth_cap tag	not(auth_cap .tag)
all stores, all atomics, all cbos	28	1	1	auth_cap seal	isCapSealed(auth_cap)
all atomics, CBO.INVALID [*]	28	1	2	auth_cap permission	not(auth_cap . R-permission)
all stores, all atomics, CBO.INVALID [*] , CBO.ZERO [*]	28	1	2	auth_cap permission	not(auth_cap . W-permission)
CBO.CLEAN [*] , CBO.FLUSH [*]	28	1	2	auth_cap permission	not(auth_cap . R-permission) and not(auth_cap . W-permission)
all stores, all atomics	28	1	3	auth_cap length	any byte of access out of auth_cap bounds
CBO.ZERO [*] , CBO.INVALID [*]	28	1	3	auth_cap length	any byte of cache block out of auth_cap bounds
CBO.CLEAN [*] , CBO.FLUSH [*]	28	1	3	auth_cap length	all bytes of cache block out of auth_cap bounds
CBO.INVALID [*]	28	0	2	pcc permission	not(pcc . ASR-permission)
capability stores	6	N/A	N/A	capability alignment	Misaligned capability store

¹ This check is architecturally required, but is impossible to encounter so may not required in an implementation.



Indirect branches are [JALR](#), conditional branches are [Conditional branches \(BEQ, BNE, BLT\[U\], BGE\[U\]\)](#).



[CBO.ZERO](#) issues as a cache block wide store. All CMOs operate on the cache block which contains the address. Prefetches check that the capability is tagged, not sealed, has the permission ([R-permission](#), [W-permission](#), [X-permission](#)) corresponding to the instruction, and has bounds which include at least one byte of the cache block; if any check fails, the prefetch is not performed but no exception is generated.

3.11. CHERI Exceptions and speculative execution

CHERI adds architectural guarantees that can prove to be microarchitecturally useful. Speculative-execution attacks can — among other factors — rely on instructions that fail CHERI permission checks not to take effect. When implementing any of the extensions proposed here, microarchitects need to carefully consider the interaction of late-exception raising and side-channel attacks.

3.12. Physical Memory Attributes (PMA)

Typically, the entire memory space need not support tagged data. Therefore, it is desirable that harts supporting Zcheri_purecap extend PMAs with a *taggable* attribute indicating whether a memory region allows storing tagged data.

Data loaded from memory regions that are not taggable will always have the tag cleared. When the hart attempts to store data with the tag set to memory regions that are not taggable, the implementation may:

- Cause an access fault exception
- Implicitly set the stored tag to 0

3.13. Page-Based Virtual-Memory Systems

RISC-V's page-based virtual-memory management is generally orthogonal to CHERI. In Zcheri_purecap, capability addresses are interpreted with respect to the privilege level of the processor in line with RISC-V's handling of integer addresses. In machine mode, capability addresses are generally interpreted as physical addresses; if the [mstatus](#) MPRV flag is asserted, then data accesses (but not instruction accesses) will be interpreted as if performed by the privilege mode in [mstatus](#)'s MPP. In supervisor and user modes, capability addresses are interpreted as dictated by the current [satp](#) configuration: addresses are virtual if paging is enabled and physical if not.

Zcheri_purecap requires that the [pcc](#) grants the [ASR-permission](#) to change the page-table root [satp](#) and other virtual-memory parameters as described in [Section 3.8](#).

3.13.1. Invalid Address Handling

When address translation is in effect and XLEN=64, the upper bits of virtual memory addresses must match for the address to be valid:

- For Sv39, bits [63:39] must equal bit 38
- For Sv48, bits [63:48] must equal bit 47
- For Sv57, bits [63:57] must equal bit 56

RISC-V permits that CSRs holding addresses, such as [mtvec](#) and [mepc](#) (see [Table 37](#)) as well as [pc](#), need not hold all possible invalid addresses. Implementations may convert an invalid address into some other invalid address that the register is capable of holding. Therefore, implementations often support area and power optimizations by compressing invalid addresses in a lossy fashion.



Compressing invalid addresses allows implementations to reduce the number of flops required to hold some CSRs, such as [mtvec](#). In CHERI, invalid addresses may also be used to reduce the number of bits to compare during a bounds check, for example, to 40 bits if

using Sv39.

However, the bounds encoding of capabilities in Zcheri_purecap depends on the address value, so implementations must not convert invalid addresses to other arbitrary invalid address in an unrestricted manner. The remainder of this section describes how invalid address handling must be supported in Zcheri_purecap when accessing CSRs, branching and jumping, and accessing memory.

Accessing CSRs

The following procedure must be used when executing instructions, such as CSRRW, that write a capability A to a CSR that cannot hold all invalid addresses:

1. If A's address is invalid and A does not have infinite bounds (see Section 2.1), then A's tag is set to 0.
2. Write the final (potentially modified) version of capability A to the CSR e.g. mtvecc, mepcc, etc.

Branches and Jumps

Control transfer instructions jump or branch to a capability A which can be:

- pcc for branches, direct jumps and any branch when the CHERI execution mode is Legacy (see Chapter 5).
- The capability in the c input register of a jump when the CHERI execution mode is Capability (see Chapter 5).

The following procedure must be used when jumping or branching to the target capability A if the pcc cannot hold all invalid addresses:

1. Calculate the effective target address T of the jump or branch as required by the instruction's behavior.
2. If T is invalid and A does not have infinite bounds (see Section 2.1), then the instruction gives rise to a CHERI fault; the CHERI jump or branch fault is reported in the TYPE field and invalid address violation is reported in the CAUSE field of mtval or stval.
3. If T is invalid and A has infinite bounds (see Section 2.1), then A's tag is unchanged and T is written into A's address field. Attempting to execute the instruction at address T gives rise to an instruction access fault as is usual in RISC-V.
4. Otherwise T is valid and the instruction behaves as normal.



RISC-V harts that do not support Zcheri_purecap normally raise an instruction access fault after jumping or branching to an invalid address. Therefore, Zcheri_purecap aims to preserve that behavior to ensure that harts supporting Zcheri_purecap and Zcheri_legacy are fully compatible with RISC-V harts provided that pcc and ddc are set to the Infinite capability.

Memory Accesses

The following procedure must be used while loading or storing to memory with a capability A when the implementation supports invalid address optimizations:

1. Calculate the effective address T of the memory access as required by the instruction's behavior.
2. If T is invalid and A does not have infinite bounds (see Section 2.1), then the instruction gives rise

to a CHERI fault; the *CHERI data* fault is reported in the TYPE field and invalid address violation is reported in the CAUSE field of `mtval` or `stval`.

3. If T is invalid and A has infinite bounds (see [Section 2.1](#)), the hart will raise an access fault as is usual in RISC-V.
4. Otherwise T is valid and the instruction behaves as normal.

3.14. Integrating Zcheri_purecap with Sdext

This section describes changes to integrate the Sdext ISA and Zcheri_purecap. It must be implemented to make external debug compatible with Zcheri_purecap. Modifications to Sdext are kept to a minimum.



This section is preliminary as no-one has yet built debug support for CHERI-RISC-V so change is likely.

The following features, which are optional in Sdext, must be implemented for use with Zcheri_purecap:

- The `hartinfo` register must be implemented.
- All harts which support Zcheri_purecap must provide `hartinfo.nscratch` of at least 1 and implement the `dscratch0c` register.
- All harts which support Zcheri_purecap must provide `hartinfo.datasize` of at least 1 and `hartinfo.dataaccess` of 0.
- The program buffer must be implemented, with `abstractcs.progbuFSIZE` of at least 4 if `dmstatus.impebreak` is 1, or at least 5 if `dmstatus.impebreak` is 0.

These requirements allow a debugger to read and write capabilities in integer registers without disturbing other registers. These requirements may be relaxed if some other means of accessing capabilities in integer registers, such as an extension of the Access Register abstract command, is added. The following sequences demonstrate how a debugger can read and write a capability in c1 if MXLEN is 64, `hartinfo.dataaccess` is 0, `hartinfo.dataaddr` is 0xBF0, `hartinfo.datasize` is 1, `dmstatus.impebreak` is 0, and `abstractcs.progbuFSIZE` is 5:



```
# Read the high MXLEN bits into data0-data1
csrrw c2, dscratch0c, c2
gchi x2, c1
csrw 0xBF0, x2
csrrw c2, dscratch0c, c2
ebreak

# Read the tag into data0
csrrw c2, dscratch0c, c2
getag x2, c1
csrw 0xBF0, x2
csrrw c2, dscratch0c, c2
ebreak

# Write the high MXLEN bits from data0-data1
csrrw c2, dscratch0c, c2
csrr x2, 0xBF0
schi c1, c1, x2
csrrw c2, dscratch0c, c2
ebreak
```

```
# Write the tag (if nonzero)
csrrw c2, dscratch0c, c2
csrr c2, dinfc
cbld c1, c2, c1
csrrw c2, dscratch0c, c2
ebreak
```

The low **MXLEN** bits of a capability are read and written using normal Access Register abstract commands. If `dscratch0c` were known to be preserved between abstract commands, it would be possible to remove the requirements on `hartinfo.datasize`, `hartinfo.dataaccess`, and `abstractcs.progbuFSIZE`, however there is no way to discover the former property.

3.14.1. Debug Mode

When executing code due to an abstract command, the hart stays in debug mode and the rules outlined in Section 4.1 of (RISC-V, 2022) apply.

3.14.2. Core Debug Registers

Zcheri_purecap renames and extends debug CSRs that are designated to hold addresses to be able to hold capabilities. The renamed debug CSRs are listed in Table 13.

The `pcc` must grant `ASR-permission` to access debug CSRs. This permission is automatically provided when the hart enters debug mode as described in the `dpcc` section. The `pcc` metadata can only be changed if the implementation supports executing control transfer instructions from the program buffer — this is an optional feature according to (RISC-V, 2022).

3.14.3. Debug Program Counter (dpc)

The `dpc` register is as defined in (RISC-V, 2022). It is a `DXLEN`-bit register used as the PC saved when entering debug mode. `dpc` is extended into `dpcc`.

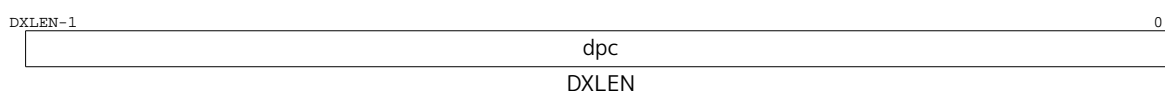


Figure 21. Debug program counter

3.14.4. Debug Program Counter Capability (dpcc)

The `dpcc` register is a renamed extension to `dpc` that is able to hold a capability.

The tag of the CSR must be reset to zero. The reset values of the metadata and address fields are UNSPECIFIED.

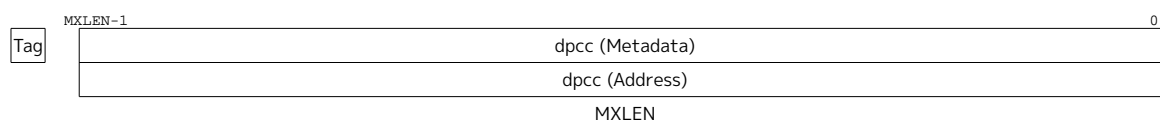


Figure 22. Debug program counter capability

Upon entry to debug mode, (RISC-V, 2022), does not specify how to update the PC, and says PC relative instructions may be illegal. This concept is extended to include any instruction which reads or

updates [pcc](#), which refers to all jumps, conditional branches and [AUIPC](#). The exception is [MODESW](#) which is supported if [Zcheri_legacy](#) is implemented, see [dinf](#) for details.

As a result, the value of [pcc](#) is UNSPECIFIED in debug mode according to this specification. The [pcc](#) metadata has no architectural effect in debug mode. Therefore [ASR-permission](#) is implicitly granted for access to all CSRs and no PCC faults are possible.

[dpcc](#) (and consequently [dpc](#)) are updated with the capability in [pcc](#) whose address field is set to the address of the next instruction to be executed as described in ([RISC-V, 2022](#)) upon debug mode entry.

When leaving debug mode, the capability in [dpcc](#) is unsealed and written into [pcc](#). A debugger may write [dpcc](#) to change where the hart resumes and its mode, permissions, sealing or bounds.

3.14.5. Debug Scratch Register 0 (dscratch0)

The [dscratch0](#) register is as defined in ([RISC-V, 2022](#)). It is an optional DXLEN-bit scratch register that can be used by implementations which need it. [dscratch0](#) is extended into [dscratch0c](#).

The tag of the CSR must be reset to zero. The reset values of the metadata and address fields are UNSPECIFIED.



Figure 23. Debug scratch 0 register

3.14.6. Debug Scratch Register 0 (dscratch0c)

The [dscratch0c](#) register is a CLEN-bit plus tag bit renamed extension to [dscratch0](#) that is able to hold a capability. Its reset value is the [NULL](#) capability.

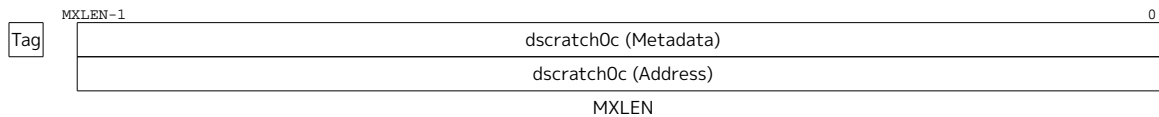


Figure 24. Debug scratch 0 capability register

3.14.7. Debug Scratch Register 1 (dscratch1)

The [dscratch1](#) register is as defined in ([RISC-V, 2022](#)). It is an optional DXLEN-bit scratch register that can be used by implementations which need it. [dscratch1](#) is extended into [dscratch1c](#).

The tag of the CSR must be reset to zero. The reset values of the metadata and address fields are UNSPECIFIED.



Figure 25. Debug scratch 0 register

3.14.8. Debug Scratch Register 1 (dscratch1c)

The [dscratch1c](#) register is a CLEN-bit plus tag bit renamed extension to [dscratch1](#) that is able to hold a capability. Its reset value is the [NULL](#) capability.

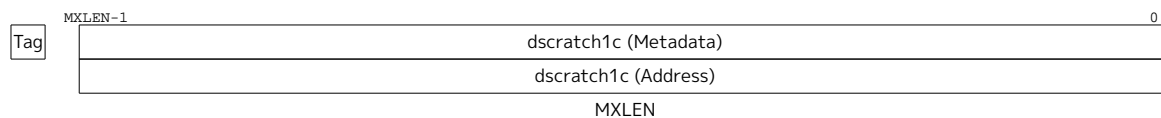


Figure 26. Debug scratch 1 capability register

3.14.9. Debug Infinite Capability Register (dinf)

The [dinf](#) register is a CLEN-bit plus tag bit CSR only accessible in debug mode.

The reset value is the [Infinite](#) capability.

If Zcheri_legacy (see xref:section-cheri-execution-mode) is implemented:

1. the core enters Capability Mode when entering debug mode
 - a. therefore [dinf](#).M is set whenever entering debug mode for any reason.
2. the mode can be optionally switched using [MODESW](#), and the result observed in [dinf](#).M.

[dinf](#) is read/write but with no writeable fields, and so writes are ignored.



A future version of this specification may add writeable fields to allow creation of other capabilities, if, for example, a future extension requires multiple formats for the [Infinite](#) capability.

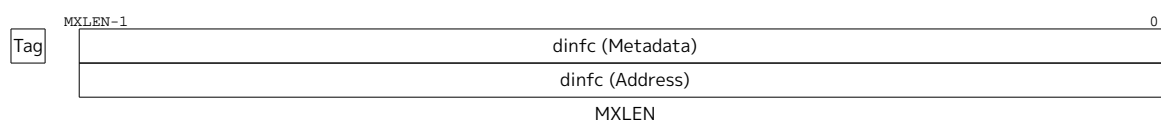


Figure 27. Debug infinite capability register

3.15. Integrating Zcheri_purecap with Sdtrig

The Sdtrig extension is generally orthogonal to Zcheri_purecap. However, the priority of synchronous exceptions and where triggers fit is adjusted as shown in [Table 23](#).

Table 23. Synchronous exception priority (including triggers) in decreasing priority order. Entries added in Zcheri_purecap are in **bold**

Prior ity	Exc.C ode	Description	Trigger
Highe st	3		etrigger
	3		icount
	3		itrigger
	3		mcontrol/mcontrol6 after (on previous instruction)
	3	Instruction address breakpoint	mcontrol/mcontrol6 execute address before
	28	Prior to instruction address translation: CHERI fault due to PCC checks (tag, execute permission, invalid address and bounds)	

Prior ity	Exc.C ode	Description	Trigger
	12, 1	During instruction address translation: First encountered page fault or access fault	
	1	With physical address for instruction: Instruction access fault	
	3		mcontrol/mcontrol6 execute data before
	2 0 8,9,11 3	Illegal instruction Instruction address misaligned Environment call Environment break	
	3	Load/store/AMO address breakpoint	mcontrol/mcontrol6 load/store address before
	3		mcontrol/mcontrol6 store data before
	28	CHERI faults due to: PCC ASR-permission clear Branch/jump target address checks (tag, execute permissions, invalid address and bounds)	
	28	Prior to address translation for an explicit memory access: Load/store/AMO capability address misaligned CHERI fault due to capability checks (tag, permissions, invalid address and bounds)	
	4,6	Optionally: Load/store/AMO address misaligned	
	13, 15, 5, 7	During address translation for an explicit memory access: First encountered page fault or access fault	
	5,7	With physical address for an explicit memory access: Load/store/AMO access fault	
	4,6	If not higher priority: Load/store/AMO address misaligned	
<i>Lowest</i>	3		mcontrol/mcontrol6 load data before

Chapter 4. "Zcheri_pte" Extension for CHERI Page-Based Virtual-Memory Systems

CHERI is a security mechanism that is generally orthogonal to page-based virtual-memory management as defined in (RISC-V, 2023). However, it is helpful in CHERI harts to extend RISC-V's virtual-memory management to control the flow of capabilities in memory at the page granularity. For this reason, the Zcheri_pte extension adds new bits to RISC-V's Page Table Entry (PTE) format.

4.1. Extending the Page Table Entry Format



CHERI v9 Note: The current proposal is provisional and is missing PTE bits when compared to CHERI v9.

The page table entry format remains unchanged for Sv32. However, two new bits, Capability Write (CW) and Capability Dirty (CD), are added to leaf PTEs in Sv39, Sv48 and Sv57 as shown in Figure 28, Figure 29 and Figure 30 respectively.

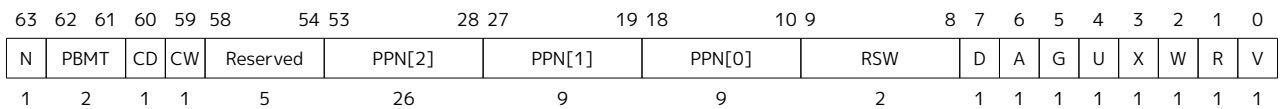


Figure 28. Sv39 page table entry

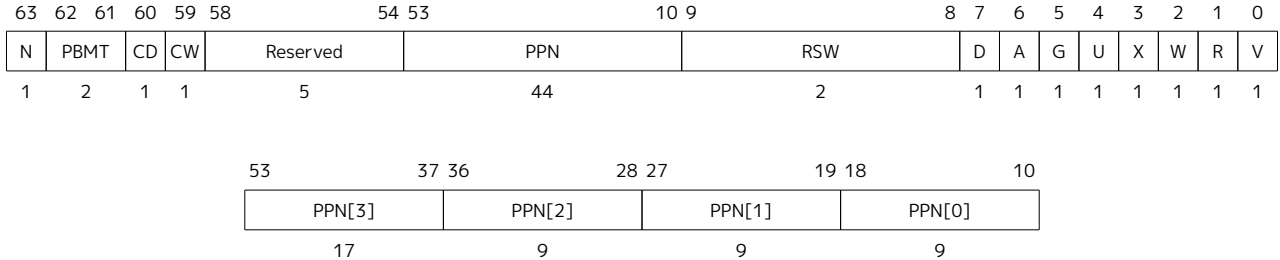


Figure 29. Sv48 page table entry

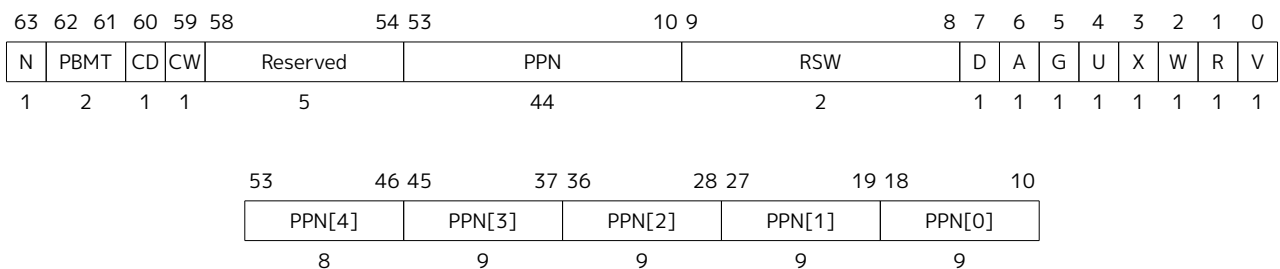


Figure 30. Sv57 page table entry

The CW bit indicates whether writing capabilities with tag set to the virtual page is permitted. Two schemes to manage the CW bit are permitted:

- A store page fault exception is raised when a capability store or AMO instruction is executed, the authorizing capability grants [W-permission](#) and [C-permission](#), and the store address corresponds to a virtual page with the CW bit clear.

- When a capability store or AMO instruction is executed, the implementation clears the tag bit of the capability written to a virtual page with the CW bit clear.



The implementation of the CW bit does not force a dependency on the tag bit's value of the capability written, so implementations must support the CW bit.

The CD bit indicates that a capability with tag set has been written to the virtual page since the last time the CD bit was cleared. Implementations are strongly encouraged, but not required, to support CD. If supported, two schemes to manage the CD bit are permitted:

- A store page fault exception is raised when a capability store or AMO instruction is executed, the authorizing capability grants [W-permission](#) and [C-permission](#), the tag bit of the capability being written is set and the address written corresponds to a virtual page with the CD bit clear.
- When a capability store or AMO instruction is executed, the authorizing capability grants [W-permission](#) and [C-permission](#), the tag bit of the capability being written is set and the store address corresponds to a virtual page with the CD bit clear, the implementation sets the corresponding bit in the PTE. The PTE update must be atomic with respect to other accesses to the PTE, and must atomically check that the PTE is valid and grants sufficient permissions. Updates to the CD bit must be exact (i.e. not speculative), and observed in program order by the local hart. Furthermore, the PTE update must appear in the global memory order no later than the explicit memory access, or any subsequent explicit memory access to that virtual page by the local hart. The ordering on loads and stores provided by FENCE instructions and the acquire/release bits on atomic instructions also orders the PTE updates associated with those loads and stores as observed by remote harts.

The PTE update is not required to be atomic with respect to the explicit memory access that caused the update, and the sequence is interruptible. However, the hart must not perform explicit memory access before the PTE update is globally visible.



The behavior of the CW bit takes priority over the CD bit. Therefore, implementations must not take action to change or raise an exception related to the CD bit when the CW bit is clear.

4.2. Extending the Machine Environment Configuration Register (menvcfg)

The **menvcfg** register is extended to allow discovering whether the implementation supports the CD bit.

The **menvcfg** register operates as described in ([RISC-V, 2023](#)). Zcheri_purecap adds a new enable bit as shown in [Figure 31](#) when XLEN=64.

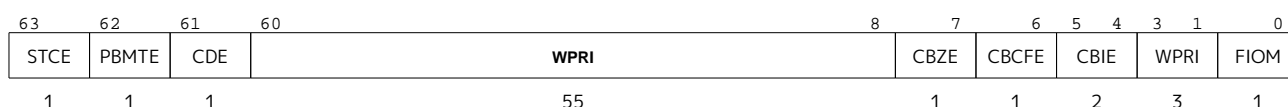


Figure 31. Machine environment configuration register (**menvcfg**)

The Capability Dirty Enable (CDE) bit controls whether the Capability Dirty (CD) bit is available for use in S-mode address translation. When CDE=1, the CD bit is available for S-mode address translation. When CDE=0, the implementation behaves as though the CD bit were not implemented. If CD is not implemented, CDE is read-only zero. If CD is implemented although not configurable,

CDE is read-only one.

Chapter 5. "Zcheri_legacy" Extension for CHERI Legacy Mode

Zcheri_legacy is an optional extension to Zcheri_purecap. Implementations that support Zcheri_purecap and Zcheri_legacy define a variant of the CHERI ISA that is fully binary compatible with existing RISC-V code.

Key features in Zcheri_legacy include a definition of a CHERI execution mode, a new unprivileged register, additional instructions and extensions to some existing CSRs enabling CHERI features. The remainder of this section describes these features in detail as well as their integration with the primary base integer variants of the RISC-V ISA (RV32I and RV64I).

5.1. CHERI Execution Mode

Zcheri_legacy adds CHERI execution modes to ensure backwards compatibility with the base RISC-V ISA while saving instruction encoding space. There are two execution modes: *Capability* and *Legacy*. Additionally, there is a new unprivileged register: the default data capability, `ddc`, that is used to authorise all data memory accesses when the current CHERI mode is Legacy.

The current CHERI execution mode is given by the mode (M) field of `pcc` that is encoded as described in [Section 5.2](#).

The CHERI execution mode impacts the instruction set in the following ways:

- The authorising capability used to execute memory access instructions. In Legacy Mode, `ddc` is implicitly used. In Capability Mode, the authorising capability is supplied as an explicit `c` operand register to the instruction.
- The set of instructions that is available for execution. Some instructions are available in Legacy Mode but not Capability Mode and vice-versa (see [Chapter 7](#)).



The implication is that the CHERI execution mode is always Capability on implementations that support Zcheri_purecap, but not Zcheri_legacy.

The CHERI execution mode is effectively an extension to some RISC-V instruction encodings. For example, the encoding of an instruction like `LW` remains unchanged, but the mode indicates whether the capability authorising the load is the register operand `cs1` (Capability Mode). The mode is shown in the assembly syntax.

The CHERI execution mode is key in providing backwards compatibility with the base RISC-V ISA. RISC-V software is able to execute unchanged in implementations supporting both Zcheri_purecap and Zcheri_legacy provided that the `Infinite` capability is installed in `ddc` and `pcc` (with `M=0`, i.e. CHERI execution mode set to Legacy). Setting both registers to `Infinite` ensures that:

- All permissions are granted
- The bounds authorise accesses to the entire address space i.e base is 0 and top is 2^{MXLEN}

5.2. CHERI Execution Mode Encoding

Zcheri_legacy adds a new CHERI execution mode field (M) to capabilities within the AP field (see

[Section 2.2.3](#)). When MXLEN=32, the mode is encoded in bit O of quadrant 1 from the AP field. When MXLEN=64, the mode is encoded in bit O of the AP field: a new M bit field is added to the capability format as shown in [Table 4](#). In both encodings, M=0 indicates Legacy Mode and M=1 indicates Capability Mode.

The current CHERI execution mode is given by the M field of the [pcc](#) and the CRE bits in [mseccfg](#), [menvcfg](#), and [senvcfg](#) as follows:

- The mode is Capability when the M field of the [pcc](#) is Capability and the effective CRE=1 for the current privilege level
- The mode is Legacy when the effective CRE=0 for the current privilege level or the M field of the [pcc](#) is Legacy

When MXLEN=32, the M field is Legacy Mode in the [Infinite](#) capability, so its AP field is 8; furthermore, the [NULL](#) capability does not grant [X-permission](#), so the M field is superfluous in this case.

When MXLEN=64, the [M-bit](#) can be set to 1 when the capability does not grant [X-permission](#). In this case, the [M-bit](#) is superfluous, so the encoding may be used to support additional features in future extensions; furthermore, the [M-bit](#) is 0 in both the [NULL](#) and [Infinite](#) capabilities.

5.3. Zcheri_legacy Instructions

Zcheri_legacy introduces a small number of new mode-switching instructions to the base RISC-V integer ISA, as shown in [Table 31](#). Additionally, the behavior of some existing instructions changes depending on the current CHERI execution mode.

5.3.1. Capability Load and Store Instructions

The load and store capability instructions change behaviour depending on the CHERI execution mode although the instruction's encoding remains unchanged.

The load capability instruction is [LC](#). When the CHERI execution mode is Capability; the instruction behaves as described in [Section 3.3](#). In Legacy Mode, the capability authorising the memory access is [ddc](#), so the effective address is obtained by adding the [x](#) register to the sign-extended offset.

The store capability instruction is [SC](#). When the CHERI execution mode is Capability; the instruction behaves as described in [Section 3.3](#). In Legacy Mode, the capability authorising the memory access is [ddc](#), so the effective address is obtained by adding the [x](#) register to the sign-extended offset.

5.3.2. Capability Manipulation Instructions

A new [SCMODE](#) instruction allows setting a capability's CHERI execution mode to the indicated value. The output is written to an unprivileged [c](#) register, not [pcc](#).

5.3.3. Mode Change Instructions

A new CHERI execution mode switch ([MODESW](#)) instruction allows software to toggle the hart's current CHERI execution mode. If the current mode in the [pcc](#) is Legacy, then the mode after executing [MODESW](#) is Capability and vice-versa. This instruction effectively writes the CHERI execution mode bit M of the capability currently installed in the [pcc](#).

5.4. Existing RISC-V Instructions

The CHERI execution mode introduced in Zcheri_legacy affects the behaviour of instructions that have at least one memory address operand. When in Capability Mode, the address input or output operands may include *c* registers. When in Legacy Mode, the address input or output operands are *x/f/v* registers; the tag and metadata of that register are implicitly set to 0.

5.4.1. Control Transfer Instructions

The unconditional jump instructions change behaviour depending on the CHERI execution mode although the instruction's encoding remains unchanged.

The jump and link instruction *JAL* when the CHERI execution mode is Capability; behaves as described in [Section 3.4](#). When the mode is Legacy. In this case, the address of the instruction following the jump ($pc + 4$) is written to an *x* register; that register's tag and capability metadata are zeroed.

The jump and link register instruction is *JALR* when the CHERI execution mode is Capability; behaves as described in [Section 3.4](#). When the mode is Legacy. In this case, the target address is obtained by adding the sign-extended 12-bit immediate to the *x* register operand, then setting the least significant bit of the result to zero. The target address is then written to the *pcc* address and a representability check is performed. The address of the instruction following the jump ($pc + 4$) is written to an *x* register; that register's tag and capability metadata are zeroed.

Zcheri_legacy allows changing the current CHERI execution mode when executing *JALR* from Capability Mode.

JAL and *JALR* cause CHERI exceptions when a minimum sized instruction at the target address is not within the bounds of the *pcc*. An instruction address misaligned exception is raised when the target address is misaligned.

5.4.2. Conditional Branches

The behaviour is as shown in [Section 3.4.2.2](#).

5.4.3. Load and Store Instructions

Load and store instructions change behavior depending on the CHERI execution mode although the instruction's encoding remains unchanged.

Loads and stores behave as described in [Section 3.4](#) when the CHERI execution mode is Capability. In Legacy Mode, the instructions behave as described in the RISC-V base ISA (i.e. without the 'C' prefix) and rely on *x* operands only. The capability authorising the memory access is *ddc* and the memory address is given by sign-extending the 12-bit immediate offset and adding it to the base address in the *x* register operand.

The exception cases remain as described in [Section 3.4](#) regardless of the CHERI execution mode.

5.4.4. CSR Instructions



CHERI v9 Note: *CSpecialRW* is removed. Its role is assumed by *CSRRW*.

Zcheri_legacy adds the concept of CSRs which contain a capability where the address field is visible in Legacy Mode (e.g. [mtvec](#)) and the full capability is visible in Capability Mode through a different name (e.g. [mtvecc](#)). These are referred to as *extended CSRs*.

Extended CSRs have only one address; the access width is determined by the execution mode.

When [CSRWR](#) is executed on an extended CSR in Legacy Mode:

- The register operand is an **x** register.
- Only XLEN bits from the **x** source are written to the capability address field.
 - The tag and metadata are updated as specified in [Table 36](#).
- Only XLEN bits are read from the capability address field, which are extended to MXLEN bits according to ([RISC-V, 2023](#)) (3.1.6.2. *Base ISA Control in mstatus Register*) and are then written to the destination **x** register.

When [CSRWR](#) is executed on an extended CSR in Capability Mode:

- The register operand is a **c** register.
- The full capability in the **c** register source is written to the CSR.
 - The capability may require modification before the final written value is determined (see [Table 36](#)).
- The full capability is written to destination **c** register.

When an extended CSR is used with another CSR instruction ([CSRWRI](#), [CSRRC](#), [CSRRCI](#), [CSRRS](#), [CSRRSI](#)):

- The final address is calculated according to the standard RISC-V CSR rules (set bits, clear bits etc).
- The final address is updated as specified in [Table 36](#) for an XLEN write.
- XLEN bits are read from the capability address field and written to an output **x** register.

There is *no distinction* between Legacy and Capability Mode in this case - XLEN access is always performed, and the assembly syntax always uses **x** registers.

All CSR instructions cause CHERI exceptions if the [pcc](#) does not grant [ASR-permission](#) and the CSR accessed is not user-mode accessible.

Accessing a capability CSR other than an extended CSR in Legacy Mode results in an illegal instruction exception. These CSRs are listed in [Table 25](#).

5.5. Integrating Zcheri_legacy with Sdext

A new debug default data capability ([dddc](#)) CSR is added at the CSR number shown in [Table 25](#).

Zcheri_legacy optionally allows [MODESW](#) to execute in debug mode.

When entering debug mode, the core always enters Capability Mode.

1. the mode can be optionally switched using [MODESW](#).
2. the current mode can always be observed in [dinfo.M](#).



CHERI v9 Note: The mode change instruction [MODESW](#) is new and the requirement to optionally support it in debug mode is also new.

5.6. Debug Default Data Capability (dddc)

[dddc](#) is a register that is able to hold a capability. The address is shown in [Table 25](#).

The tag of the CSR must be reset to zero. The reset values of the metadata and address fields are UNSPECIFIED.

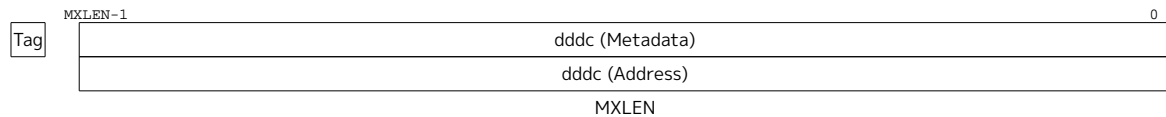


Figure 32. Debug default data capability

Upon entry to debug mode, [ddc](#) is saved in [dddc](#). [ddc](#)'s metadata is set to the [Infinite](#) capability's metadata and [ddc](#)'s address remains unchanged.

When debug mode is exited by executing [DRET](#), the hart's [ddc](#) is updated to the capability stored in [dddc](#). A debugger may write [dddc](#) to change the hart's context.

As shown in [Table 37](#), [dddc](#) is a data pointer, so it does not need to be able to hold all possible invalid addresses.

5.7. Disabling CHERI Registers



CHERI v9 Note: This feature is new and different from CHERI v9's per-privilege enable bits.



CHERI v9 Note: The rules for excepting have been tightened here. Also, it is not possible to disable CHERI checks completely.

Zcheri_legacy includes functions to disable explicit access to CHERI registers. The following occurs when executing code in a privilege mode that has CHERI register access disabled:

- The CHERI instructions in [Section 3.3](#) and [Section 8.4](#) cause illegal instruction exceptions
- Executing CSR instructions accessing any capability wide CSR addresses ([Section 3.6](#)) cause illegal instruction exceptions
- All allowed instructions execute as if the CHERI execution mode is Legacy. The mode bit in [pcc](#) is treated as if it was zero while CHERI register access is disabled.

CHERI register access is disabled if XLEN in the current mode is less than MXLEN, if the endianness in the current mode is not the reset value of [mstatus](#).MBE, or if CRE active at the current mode ([mseccfg](#).CRE for M-mode, [menvcfg](#).CRE for S-mode or [senvcfg](#).CRE for U-mode) is 0.

[mseccfg](#).CRE, [menvcfg](#).CRE, and [senvcfg](#).CRE form a single WARL field. This allows higher privilege software to restrict lower privilege software access to CHERI register state, and the ability to enter Capability Mode. The valid configurations are shown in [Table 24](#).

Table 24. Xenvcfg joint WARL field

mseccfg.CRE	menvcfg.CRE	senvcfg.CRE	Comment
0	read-only 0	read-only 0	mseccfg.CRE =0 completely disables CHERI access
1	0	read-only 0	menvcfg.CRE =0 disables access for privilege less than M-mode
1	1	0/1	senvcfg.CRE can be programmed to enable/disable access for U-mode

The WARL programming nature is extended to include UXLEN and SXLEN, as they can only be programmed to be smaller than MXLEN if the CRE bit active for the current mode is disabled.

Disabling CHERI register access has no effect on implicit accesses or security checks. The last capability installed in [pcc](#) and [ddc](#) before disabling CHERI register access will be used to authorise instruction execution and data memory accesses.



Disabling CHERI register access prevents a low-privileged Legacy Mode from interfering with the correct operation of higher-privileged Legacy Modes that do not perform [ddc](#) switches on trap entry and return.

5.8. Added CLEN-wide CSRs

Zcheri_legacy adds the CLEN-wide CSRs shown in [Table 25](#).

Table 25. CLEN-wide CSRs added in Zcheri_legacy

CLEN CSR	Address	Prerequisites	Permissions	Description
dddc	0x7bc	Sdext	DRW	Debug Default Data Capability (saved/restored on debug mode entry/exit)
mtdc	0x74c	M-mode	MRW, ASR-permission	Machine Trap Data Capability (scratch register)
stdc	0x163	S-mode	SRW, ASR-permission	Supervisor Trap Data Capability (scratch register)
ddc	0x416	none	URW	User Default Data Capability

5.8.1. Machine Status Registers (mstatus and mstatush)

Zcheri_legacy eliminates some restrictions for SXL and UXL imposed in Zcheri_purecap to allow implementations supporting multiple base ISAs. Namely, the SXL and UXL fields may be writable.

Setting the SXL or UXL field to a value that is not MXLEN disables most CHERI features and instructions, as described in [Section 5.7](#), while in that privilege mode.



If CHERI register access must be disabled in a mode for security reasons, software should set CRE to 0 regardless of the SXL and UXL fields.

Whenever XLEN in any mode is set to a value less than MXLEN, standard RISC-V rules from ([RISC-V, 2023](#)) are followed. This means that all operations must ignore source operand register bits above the

configured XLEN, and must sign-extend results to fill all MXLEN bits in the destination register. Similarly, `pc` bits above XLEN are ignored, and when the `pc` is written, it is sign-extended to fill MXLEN. The integer writing rule from CHERI is followed, so that every register write also zeroes the metadata and tag of the destination register.

However, CHERI operations and security checks will continue using the entire hardware register (i.e. CLEN bits) to correctly decode capability bounds.

Zcheri_legacy eliminates some restrictions for MBE, SBE, and UBE imposed in Zcheri_purecap to allow implementations supporting multiple endiannesses. Namely, the MBE, SBE, and UBE fields may be writable if the corresponding privilege mode is implemented.

Setting the MBE, SBE, or UBE field to a value that is not the reset value of MBE disables most CHERI features and instructions, as described in [Section 5.7](#), while in that privilege mode.

5.8.2. Machine Trap Default Capability Register (`mtdc`)

The `mtdc` register is capability width read/write register dedicated for use by machine mode. Typically, it is used to hold a data capability to a machine-mode hart-local context space, to load into `ddc`.

The tag of the CSR must be reset to zero. The reset values of the metadata and address fields are UNSPECIFIED.

value is the `NULL` capability.

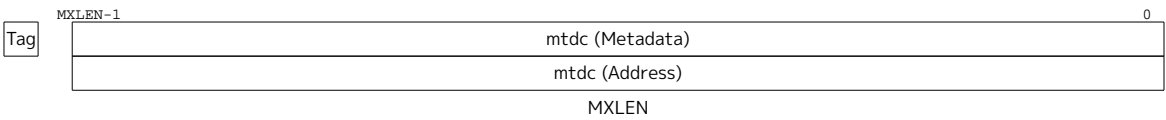


Figure 33. Machine-mode trap data capability register

5.8.3. Machine Security Configuration Register (`mseccfg`)

Zcheri_legacy adds a new enable bit to `mseccfg` as shown in [Figure 34](#).

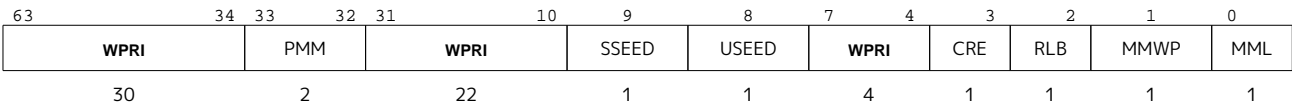


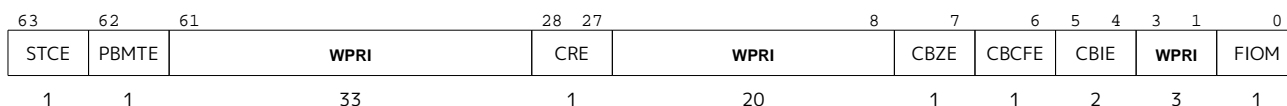
Figure 34. Machine security configuration register (`mseccfg`)

The CHERI Register Enable (CRE) bit controls whether M-mode has access to capability registers and instructions. When `CRE=1`, all CHERI instructions and registers can be accessed. When `CRE=0`, CHERI register and instruction access is prohibited for M-mode and lower privilege levels as described in [Section 5.7](#).

The reset value is 0.

5.8.4. Machine Environment Configuration Register (`menvcfg`)

Zcheri_legacy adds two new enable bits to `menvcfg` as shown in [Figure 35](#).

Figure 35. Machine environment configuration register (**menvcfg**)

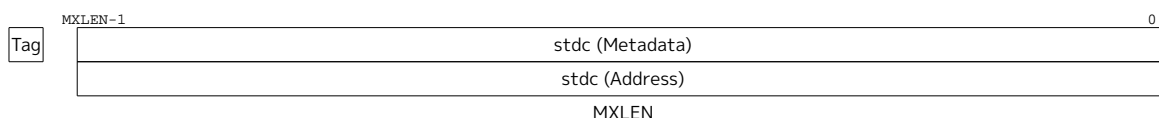
The CHERI Register Enable (CRE) bit controls whether less privileged levels can perform explicit accesses to CHERI registers. When CRE=1, CHERI registers can be read and written by less privileged levels. When CRE=0, CHERI registers are disabled in less privileged levels as described in [Section 5.7](#). CRE is read-only zero if [mseccfg](#).CRE=0.

The reset value is 0.

5.8.5. Supervisor Trap Default Capability Register (**stdc**)

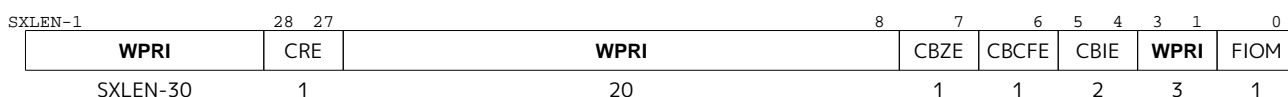
The **stdc** register is capability width read/write register dedicated for use by supervisor mode. Typically, it is used to hold a data capability to a supervisor-mode hart-local context space, to load into **ddc**.

The tag of the CSR must be reset to zero. The reset values of the metadata and address fields are UNSPECIFIED.

Figure 36. Supervisor trap data capability register (**stdc**)

5.8.6. Supervisor Environment Configuration Register (**senvcfg**)

The **senvcfg** register operates as described in the RISC-V Privileged Specification. Zcheri_legacy adds two new enable bits as shown in [Figure 37](#).

Figure 37. Supervisor environment configuration register (**senvcfg**)

The CHERI Register Enable (CRE) bit controls whether U-mode can perform explicit accesses to CHERI registers. When CRE=1, CHERI registers can be read and written by U-mode. When CRE=0, CHERI registers are in U-mode disabled as described in [Section 5.7](#). CRE is read-only zero if [menvcfg](#).CRE=0.

The reset value is 0.

5.8.7. Default Data Capability (**ddc**)

The **ddc** CSR is a read-write capability register implicitly used as an operand to authorise all data memory accesses when the current CHERI mode is Legacy. This register must be readable in any implementation. Its reset value is the [Infinite](#) capability.

As shown in [Table 37](#), **ddc** is a data pointer, so it does not need to be able to hold all possible invalid addresses.

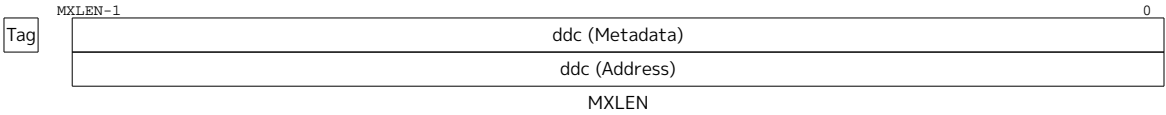


Figure 38. Unprivileged default data capability register

Chapter 6. "Zstid Extension for Thread Identification

Zstid is an optional extension to the RISC-V base ISA. Implementations that support Zcheri_purecap and Zstid. define a variant of the CHERI ISA that allows for more efficient software compartmentalization of CHERI programs.

6.1. Control and Status Registers (CSRs)

Zstid adds two new CSRs to implement a trusted thread identifier (TID) used in compartmentalization. These CSRs are listed in [Table 26](#) and [Table 27](#).

Table 26. Added supervisor-mode CSRs in Zstid

Zstid CSR	Address	Prerequisites	Permissions	Description
stid	0x580	S-mode	SRW, ASR-permission	Supervisor Thread Identifier

Table 27. Added user-mode CSRs in Zstid

Zstid CSR	Address	Prerequisites	Permissions	Description
utid	0xC80	U-mode	URO	User Thread Identifier

6.2. Supervisor-Level and Unprivileged CSRs

6.2.1. Supervisor Thread Identifier (stid)

The [stid](#) register is an SXLEN-bit read-write register. It is used to identify the current thread. The reset value of this register is UNSPECIFIED.

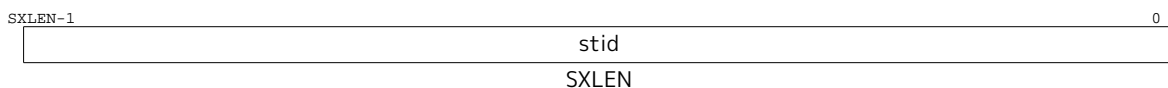


Figure 39. Supervisor thread identifier register

6.2.2. User Thread Identifier (utid)

The [utid](#) register is an UXLEN-bit read-only register. It is a read-only copy of the [stid](#) register. The reset value of this register is UNSPECIFIED.

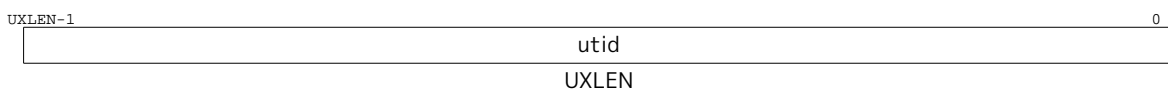


Figure 40. User thread identifier register

When Zcheri_purecap is implemented, the Zstid CSRs are extended as follows:

6.2.3. Supervisor Thread Identifier Capability (stidc)

The [stidc](#) register is an CLEN-bit read-write capability register. It is the capability extension of the [stid](#) register. It is used to identify the current thread. On reset the tag of [stidc](#) will be set to 0 and the

remainder of the data is UNSPECIFIED.

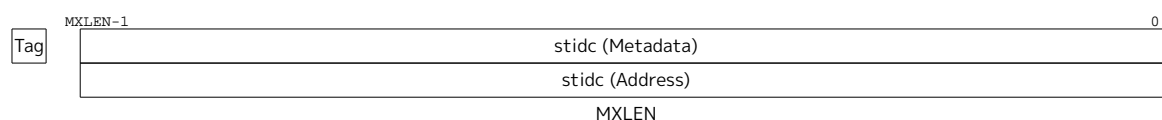


Figure 41. Supervisor thread identifier capability register

6.2.4. User Thread Identifier Capability (utidc)

The **utidc** register is an CLEN-wide read-only capability register. It is the capability extension of the **utid** register. **utidc** is a read-only copy of the **stidc** register. On reset the tag of **utidc** will be set to 0 and the remainder of the data is UNSPECIFIED.

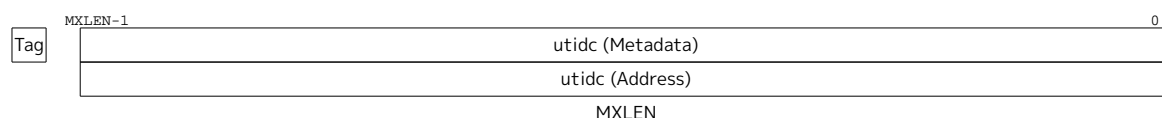


Figure 42. User thread identifier capability register

6.3. CHERI Compartmentalization

This section describes how this specification enables support for compartmentalization for CHERI systems. Compartmentalization seeks to separate the privileges between different protection units, e.g., two or more libraries. Code can be separated by sentries, which allow for giving out code capabilities to untrusted code where the untrusted code can only call the code capability, but not modify it. Sentries can be called from different threads and thus there needs to be a way of identifying the current thread. While identifying the current thread can be done by privileged code, e.g., the kernel, the implied performance overhead of this is not bearable for CHERI systems with many compartments.

The RISC-V ABI includes a *thread pointer* (*tp*) register, which is not usable for the purpose of reliably identifying the current thread because the *tp* register is a general purpose register and can be changed arbitrarily by untrusted code. Therefore, this specification offers two additional CSRs that facilitate a trusted source for the thread ID. The supervisor thread identifier (**stid**) register is readable and writeable with **ASR-permission** from the supervisor mode allowing to change the thread ID on a context switch. The user thread identifier (**utid**) exposes the current value of **stid** as a read-only copy.

This extension extends **stid** to its capability variant **stidc** and **utid** to its capability variant **utidc**. This presents software with the freedom to still use these registers with capabilities or leave the metadata untouched and only use the registers to storage integers.

Chapter 7. RISC-V Instructions and Extensions Reference

These instruction pages are for the new CHERI instructions, and some existing RISC-V instructions where the effect of CHERI needs specific details.

For existing RISC-V instructions, note that:

1. In Legacy Mode, every byte of each memory access is bounds checked against [ddc](#)
2. In Legacy Mode, a minimum length instruction at the target of all indirect jumps is bounds checked against [pcc](#)
3. In Capability Mode a minimum length instruction at the target of all indirect jumps is bounds checked against **cs1** (e.g. [JALR](#))
4. A minimum length instruction at the taken target of all direct jumps and conditional branches is bounds checked against [pcc](#) regardless of CHERI execution mode



Not all RISC-V extensions have been checked against CHERI. Compatible extensions will eventually be listed in a CHERI profile.

7.1. "Zcheri_purecap" and "Zcheri_legacy" Extensions for CHERI

7.1.1. CMV



CHERI v9 Note: This page has **new** encodings.



CHERI v9 Note: this instruction was called *CMOVE*.

Synopsis

Capability move

Mnemonic

`cmv cd, cs1`

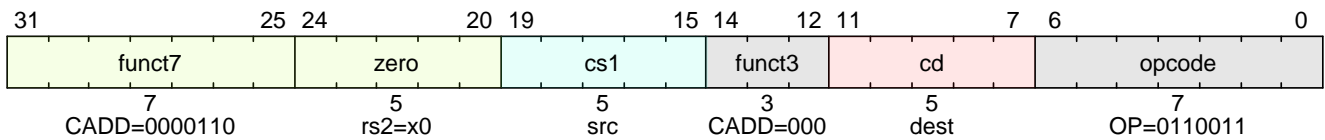
Suggested assembly syntax

`mv cd, cs1`



the suggested assembly syntax distinguishes from integer `mv` by operand type.

Encoding



CMV is encoded as *CADD* with `rs2=x0`.

Description

The contents of capability register `cs1` are written to capability register `cd`. *CMV* unconditionally moves the whole capability to `cd`.

Prerequisites

Zcheri_purecap

Operation

TODO

7.1.2. MODESW



CHERI v9 Note: This page has **new** encodings.

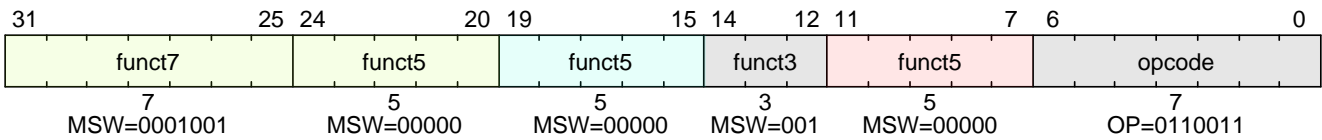
Synopsis

Switch CHERI execution mode

Mnemonics

modesw

Encoding



Description

Toggle the hart’s current CHERI execution mode in [pcc](#). If the current mode in [pcc](#) is Legacy, then the mode bit (M) in [pcc](#) is set to Capability. If the current mode is Capability, then the mode bit (M) in [pcc](#) is set to Legacy.

In debug mode MODESW can still be used to change the operating mode, and the current mode is shown in the M bit of [dinf](#).



Support of MODESW is optional in debug mode. If it is supported then it updates [dinf](#).M instead of [pcc](#).M to show the current mode.

Prerequisites

Zcheri_legacy

Operation

TODO

7.1.3. CADDI

See [CADD](#).

7.1.4. CADD



CHERI v9 Note: This page has **new** encodings.



CHERI v9 Note: these instructions were called *CINCOFFSET* and *CINCOFFSETIMM*.



CHERI v9 Note: the immediate format has changed

Synopsis

Capability pointer increment

Mnemonic

```
cadd cd, cs1, rs2
caddi cd, cs1, imm
```

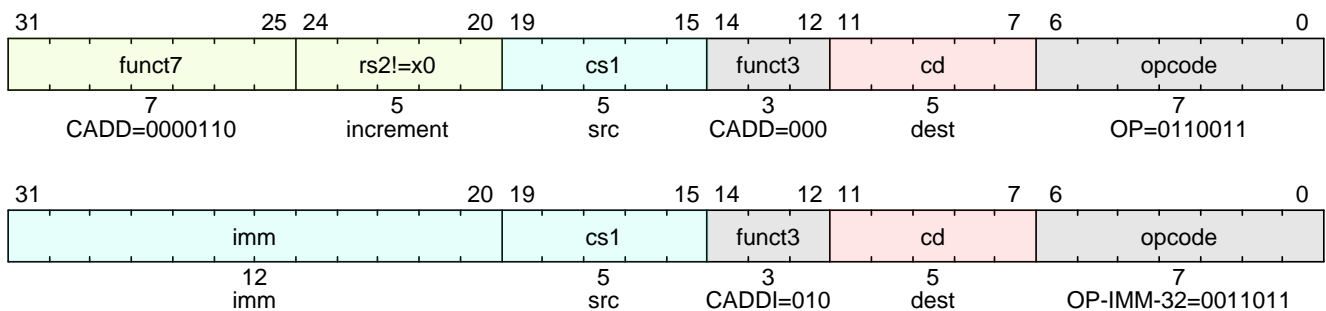
Suggested assembly syntax

```
add cd, cs1, rs2
add cd, cs1, imm
```



the suggested assembly syntax distinguishes from integer **add** by operand type.

Encoding



[CADD](#) with `rs2=x0` is decoded as [CMV](#) instead, the key difference being that tagged and sealed capabilities do not have their tag cleared by [CMV](#).

Description

Increment the address field of the capability `cs1` and write the result to `cd`. The tag bit of the output capability is 0 if `cs1` did not have its tag set to 1, the incremented address is outside `cs1`'s [Representable Range](#) or `cs1` is sealed.

For [CADD](#), the address is incremented by the value in `rs2`.

For [CADDI](#), the address is incremented by the immediate value `imm`.

Prerequisites

Zcheri_purecap

Operation (CADD)

TODO

Operation (CADDI)

TODO

7.1.5. SCADDR



CHERI v9 Note: This page has **new** encodings.



CHERI v9 Note: this instruction was called CSETADDR.

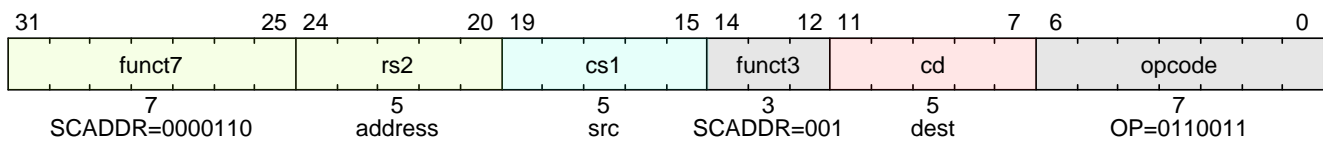
Synopsis

Capability set address

Mnemonic

scaddr cd, cs1, rs2

Encoding



Description

Set the address field of capability **cs1** to **rs2** and write the output capability to **cd**. The tag bit of the output capability is 0 if **cs1** did not have its tag set to 1, **rs2** is outside the [Representable Range](#) of **cs1** or if **cs1** is sealed.

Prerequisites

Zcheri_purecap

Operation

TODO

7.1.6. ACPERM



CHERI v9 Note: The implementation of this instruction changes because the permission fields are encoded differently in the new capability format.



CHERI v9 Note: this instruction was called CANDPERM.



CHERI v9 Note: This page has **new** encodings.

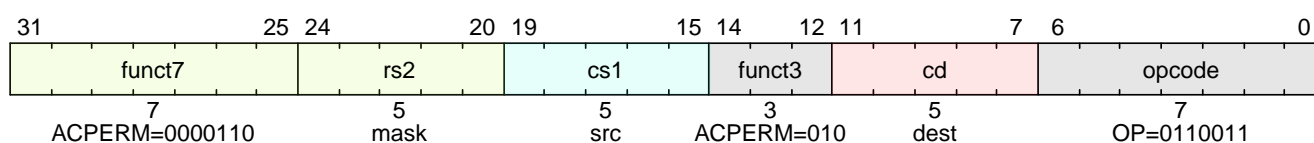
Synopsis

Mask capability permissions

Mnemonics

acperm cd, cs1, rs2

Encoding



Description

Converts the AP and SDP fields of capability **cs1** into a bit field; one bit per permission as shown below. Then calculate the bitwise AND of the bit field with the mask **rs2**. Set the AP and SDP fields of **cs1** as indicated in the resulting bit field – the capability grants a permission if the corresponding bit is set in the bit field – and write the output capability to **cd**. The output capability has its tag set to 0 if **cs1** is sealed.

The AP field is not able to encode all combinations of permissions when MXLEN=32, therefore removing a permission may yield a capability which cannot be encoded. Valid combinations are shown in [Table 3](#).

Additionally some combinations of permissions are not useful when MXLEN=64. These cases are defined to return useful minimal sets of permissions, which may be no permissions.



Future extensions may allow more combinations of permissions, especially for MXLEN=64.

The common rules are:

1. [ASR-permission](#) cannot be set without [X-permission](#) being set
 - a. Clear [ASR-permission](#) unless [X-permission](#) is set
2. [C-permission](#) cannot be set without at least one of [R-permission](#) or [W-permission](#) being set.
 - a. Clear [C-permission](#) unless [R-permission](#) or [W-permission](#) are set.
3. Clear the [M-bit](#) if clearing [X-permission](#).

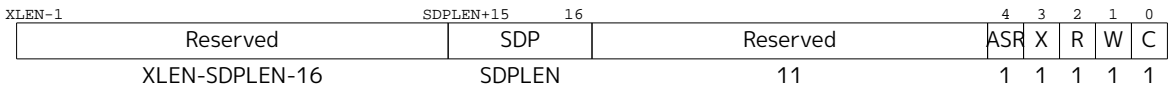


The combination of [X-permission](#) clear and [M-bit](#) set is reserved for future extensions.

The MXLEN=32 additional rules are:

1. Clear [ASR-permission](#) if *all* other permissions are not set

2. Clear C-permission and X-permission if R-permission is not set
3. Clear X-permission if X-permission and R-permission are set, but C-permission and W-permission are not set.



Prerequisites

Zcheri_purecap

Operation

TODO: Sail does not have the new encoding of the permissions field.

7.1.7. SCMODE



CHERI v9 Note: This instruction used to be CSETFLAGS (and previously CSETMODE in this document).



CHERI v9 Note: This page has **new** encodings.

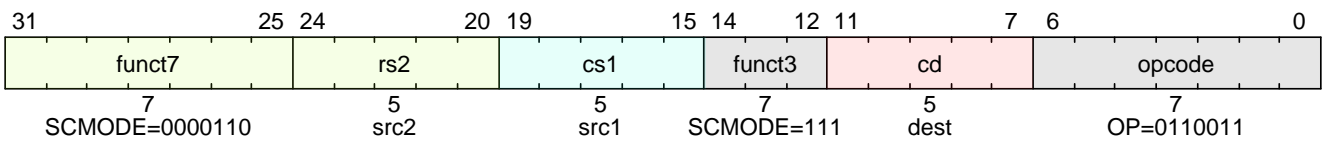
Synopsis

Capability set CHERI execution mode

Mnemonic

scmode cd, cs1, rs2

Encoding



Description

Copy **cs1** to **cd** and set **cd.M** (the mode bit) to the least significant bit of **rs2** . **cd.tag** is set to 0 if **cs1** is sealed.



Setting the mode bit on non-executable capabilities is not supported and may be ignored. The reason for this is that the [RV32 permissions encoding](#) does not allocate space for these.

TODO: Should we require execute permission for both RV32 and RV64?

Prerequisites

Zcheri_legacy

Operation

TODO

7.1.8. SCHI



CHERI v9 Note: *This page has **new** encodings.*



CHERI v9 Note: *this instruction was called CSETHIGH.*

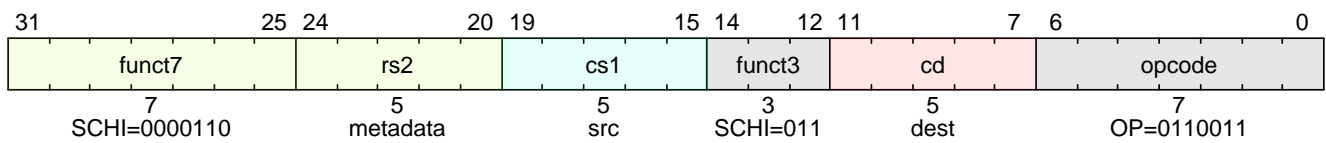
Synopsis

Capability set metadata

Mnemonic

`schi cd, cs1, rs2`

Encoding



Description

Copy `cs1` to `cd`, replace the capability metadata (i.e. bits `[CLLEN-1:MXLEN]`) with `rs2` and set `cd.tag` to 0.



Prerequisites

Zcheri_purecap

Operation

TODO

7.1.9. SCEQ

-  CHERI v9 Note: This page has **new** encodings.
-  CHERI v9 Note: this instruction was called CSETEQUALEXACT.

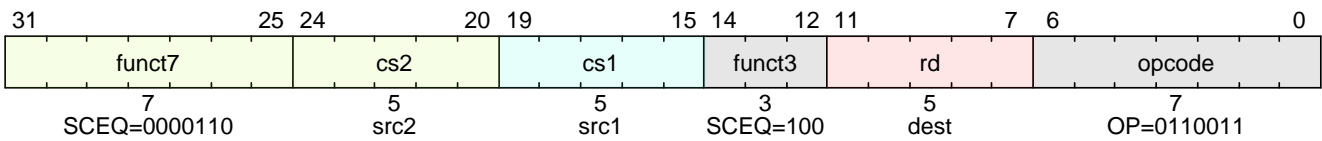
Synopsis

Set if Capabilities are Equal

Mnemonics

sceq rd, cs1, cs2

Encoding



Description

rd is set to 1 if all bits (i.e. CLEN bits and the tag) of capabilities **cs1** and **cs2** are equal, otherwise **rd** is set to 0.



Prerequisites

Zcheri_purecap

Operation

TODO

7.1.10. SENTRY

-  CHERI v9 Note: This page has **new** encodings.
-  CHERI v9 Note: this instruction was called *CSEALENTRY*.

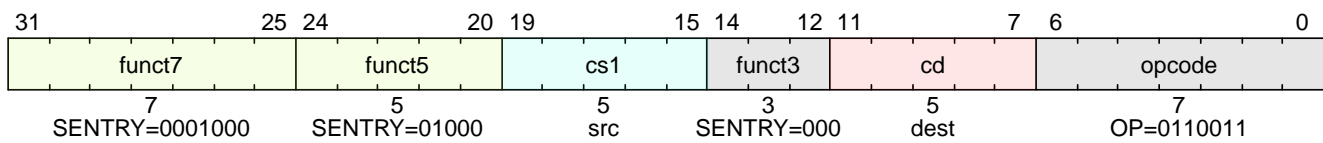
Synopsis

Seal capability as sealed entry.

Mnemonics

sentry cd, cs1

Encoding



Description

Capability **cd** is written with the capability in **cs1** with its seal bit set to 1. Attempting to seal an already sealed capability will lead to the tag of **cd** being set to 0.




Prerequisites

Zcheri_purecap

Operation

TODO

7.1.11. SCSS

-  CHERI v9 Note: *cetestsubset* does not use *ddc* if *cs1*==0
-  CHERI v9 Note: *this instruction was called CTESTSUBSET.*
-  CHERI v9 Note: *This page has new encodings.*

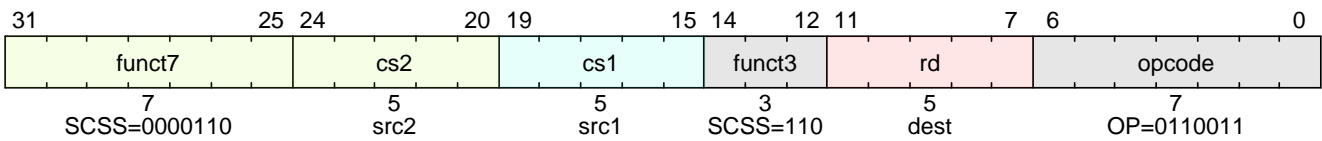
Synopsis

Capability test subset

Mnemonic


`scss rd, cs1, cs2`

Encoding



Description

`rd` is set to 1 if the tag of capabilities `cs1` and `cs2` are equal and the bounds and permissions of `cs2` are a subset of those of `cs1`.

-  *The implementation of this instruction is similar to [CBLD](#), although [SCSS](#) does not include the sealed bit in the check.*

Prerequisites

Zcheri_purecap

Operation

TODO

7.1.12. CBLD



CHERI v9 Note: CBLD does not use *ddc* if *cs1*==0



CHERI v9 Note: this instruction was called *CBUILD CAP*.



CHERI v9 Note: This page has **new** encodings.

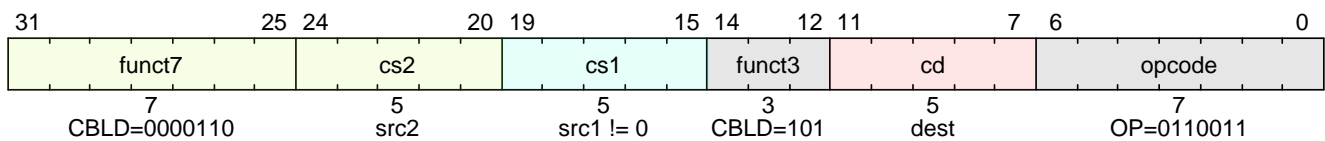
Synopsis

Capability build

Mnemonic

cbld *cd*, *cs1*, *cs2*

Encoding



Description

Copy **cs2** to **cd** and set the tag to 1 if **cs1.tag** is set, **cs1** is not sealed, **cs2** 's permissions and bounds are equal to or a subset of **cs1** 's, **cs2** 's bounds are not malformed (see [Section 2.2.6.3](#)), and all reserved bits in **cs2** 's metadata are 0; otherwise, copy **cs2** to **cd** and clear **cd** 's tag. **CBLD** is typically used alongside **SCHI** to build capabilities from integer values.



Although currently this will set the tag to 0 and leave the metadata otherwise unchanged when **cs1** is **c0**, this may change in future extensions, and so software should not assume this.

Prerequisites

Zcheri_purecap

Simplified Operation TODO not debugged much easier to read than the existing SAIL



```
let cs1_val = C(cs1);
let cs2_val = C(cs2) [with tag=1];
//isCapSubset includes derivability checks on both operands
let subset = isCapSubset(cs1_val, cs2_val);
//Clear cd.tag if cs2 isn't a subset of cs1, or if
//cs1 is untagged or sealed, or if either is underivable
C(cd)      = clearTagIf(cs2_val, not(subset) |
                      not(cs1_val.tag) |
                      isCapSealed(cs1_val));

RETIRE_SUCCESS
```

Operation

TODO: Original Sail looks at otype field, etc that don't exist

7.1.13. GCTAG

-  CHERI v9 Note: This page has **new** encodings.
-  CHERI v9 Note: this instruction was called CGETTAG.

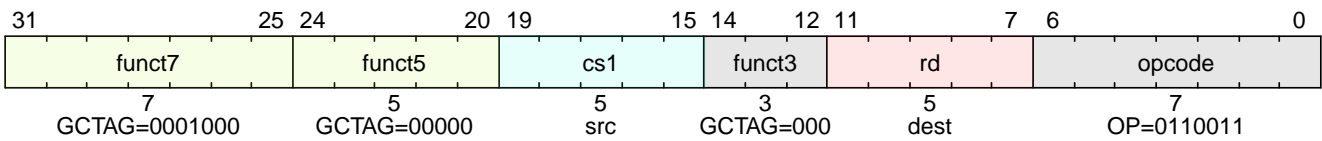
Synopsis

Capability get tag

Mnemonic

`gctag rd, cs1`

Encoding



Description

Zero extend the value of `cs1.tag` and write the result to `rd`.

Prerequisites

Zcheri_purecap

Operation

TODO

7.1.14. GCPERM



CHERI v9 Note: This page has **new** encodings.



CHERI v9 Note: this instruction was called *CGETPERM*.

Synopsis

Capability get permissions

Mnemonic

gcp perm rd, cs1

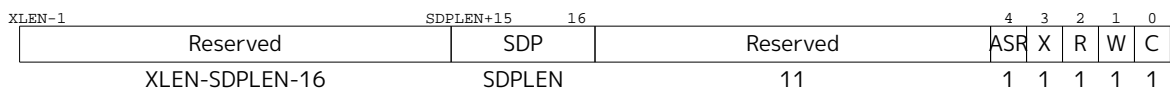
Encoding



Description

Converts the AP and SDP fields of capability **cs1** into a bit field; one bit per permission, as shown below, and write the result to **rd**. A bit set to 1 in the bit field indicates that **cs1** grants the corresponding permission.

If the AP field is a reserved value then all architectural permission bits in **rd** are set to 0. This is only possible for MXLEN=32 and the reserved values are shown in [Table 3](#).





Prerequisites

Zcheri_purecap

Operation

TODO: The encoding of permissions changed.

7.1.15. GCHI

-  CHERI v9 Note: *This page has **new** encodings.*
-  CHERI v9 Note: *this instruction was called CGETHIGH.*

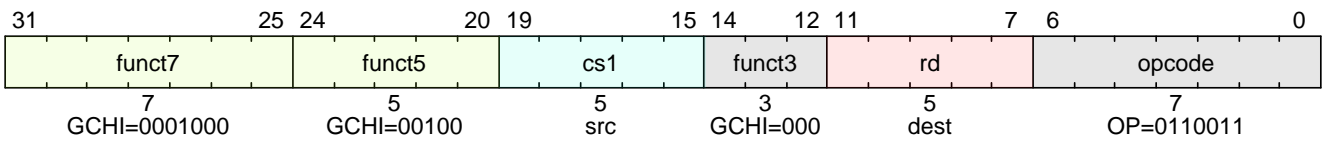
Synopsis

Capability get metadata

Mnemonic

`gchi rd, cs1`

Encoding



Description

Copy the metadata (bits [CLEN-1:MXLEN]) of capability `cs1` into `rd`.

Prerequisites

Zcheri_purecap

Operation

TODO

7.1.16. GCBASE



CHERI v9 Note: This page has **new** encodings.



CHERI v9 Note: this instruction was called *CGETBASE*.

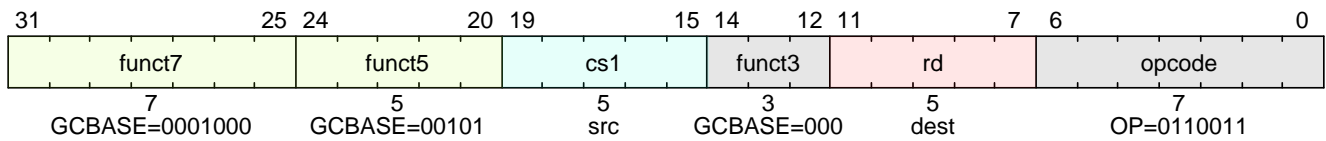
Synopsis

Capability get base address

Mnemonic

gcbase rd, cs1

Encoding



Description

Decode the base integer address from **cs1** 's bounds and write the result to **rd**. It is not required that the input capability **cs1** has its tag set to 1. **GCBASE** outputs 0 if **cs1** 's bounds are malformed (see [Section 2.2.6.3](#)).

Prerequisites



Zcheri_purecap

Operation

TODO need to check that it returns 0 if malformed

TODO

7.1.17. GCLEN

-  CHERI v9 Note: This page has **new** encodings.
-  CHERI v9 Note: this instruction was called CGETLEN.

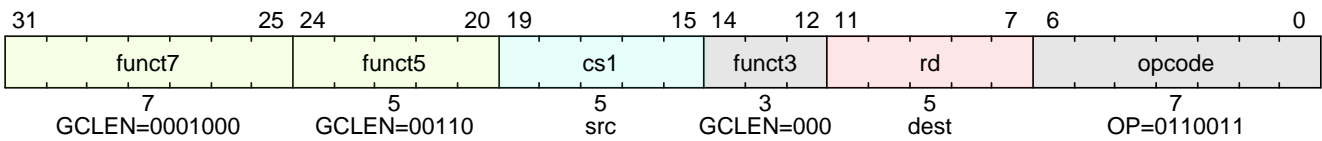
Synopsis

Capability get length

Mnemonic

gclen rd, cs1

Encoding



Description

Calculate the length of **cs1** 's bounds and write the result in **rd**. The length is defined as the difference between the decoded bounds' top and base addresses i.e. **top** - **base**. It is not required that the input capability **cs1** has its tag set to 1. **GCLEN** outputs 0 if **cs1** 's bounds are malformed (see [Section 2.2.6.3](#)), and $2^{\text{MXLEN}}-1$ if the length of **cs1** is 2^{MXLEN} .

Prerequisites

Zcheri_purecap

Operation

TODO

7.1.18. SCBNDSI

See [SCBNDS](#).

7.1.19. SCBNDS



CHERI v9 Note: *SCBNDS* was called *CSETBOUNDSEXACT*.



CHERI v9 Note: *SCBNDSI* would have been *CSETBOUNDSEXACTIMM* if it had existed.



CHERI v9 Note: This page has **new** encodings.



CHERI v9 Note: the immediate format has changed

Synopsis

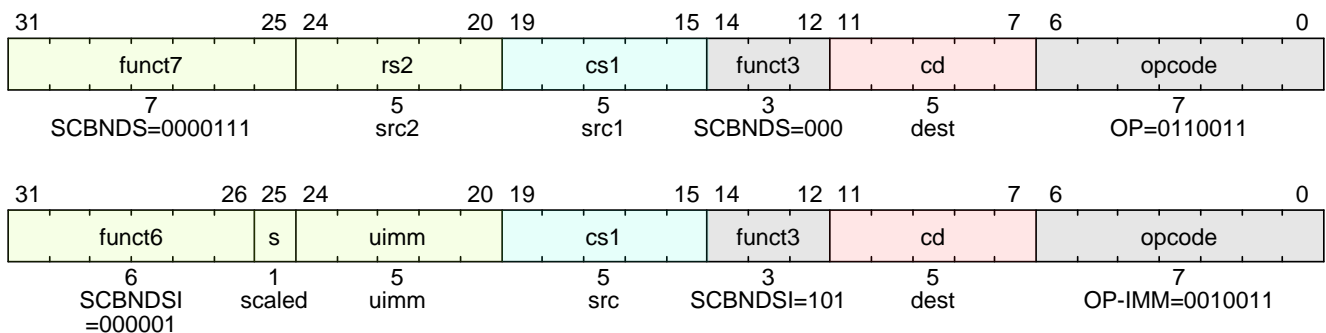
Capability set bounds

Mnemonic

`scbnds cd, cs1, rs2`

`scbndsi cd, cs1, uimm`

Encoding



Description

Capability register **cd** is set to capability register **cs1** with the base address of its bounds replaced with the value of **cs1.address** and the length of its bounds set to **rs2** (or **imm**). If the resulting capability cannot be represented exactly then set **cd.tag** to 0. In all cases, **cd.tag** is set to 0 if its bounds exceed **cs1**'s bounds, **cs1**'s tag is 0 or **cs1** is sealed.

[SCBNDSI](#) uses the **s** bit to scale the immediate by 4 places

$\text{immediate} = \text{ZeroExtend}(s ? \text{uimm} \ll 4 : \text{uimm})$

Prerequisites

Zcheri_purecap



Operation for SCBNDS

TODO

Operation for SCBNDSI

TODO

7.1.20. SCBNDSR

-  **CHERI v9 Note:** *This instruction was called CSETBOUNDS.*
-  **CHERI v9 Note:** *This page has **new** encodings.*

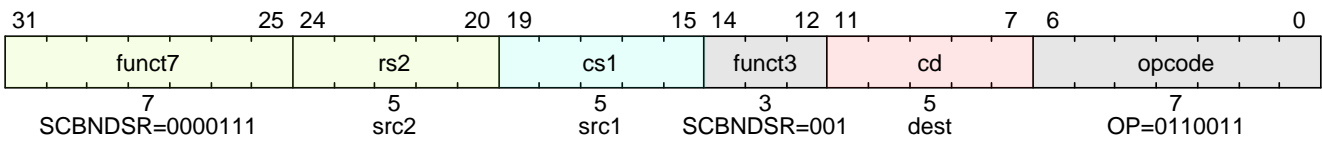
Synopsis

Capability set bounds, rounding up if necessary

Mnemonic

scbndsr cd, cs1, rs2

Encoding



Description

Capability register **cd** is set to capability register **cs1** with the base address of its bounds replaced with the value of **cs1.address** field and the length of its bounds set to **rs2**. The base is rounded down and the length is rounded up by the smallest amount needed to form a representable capability covering the requested bounds. In all cases, **cd.tag** is set to 0 if its bounds exceed **cs1**'s bounds, **cs1**'s tag is 0 or **cs1** is sealed.

Prerequisites

Zcheri_purecap

Operation for SCBNDSR

TODO

7.1.21. CRAM

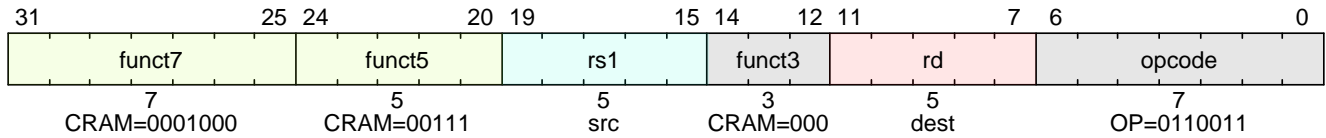
Synopsis

Get Capability Representable Alignment Mask (CRAM)

Mnemonic

`cram rd, rs1`

Encoding



Description

Integer register `rd` is set to a mask that can be used to round addresses down to a value that is sufficiently aligned to set exact bounds for the nearest representable length of `rs1`.

Prerequisites

Zcheri_purecap

Operation

TODO

7.1.22. LC



CHERI v9 Note: This page has **new** encodings.



The RV64 encoding is intended to also allocate the encoding for LQ for RV128.

Synopsis

Load capability

Capability Mode Mnemonics

lc cd, offset(cs1)

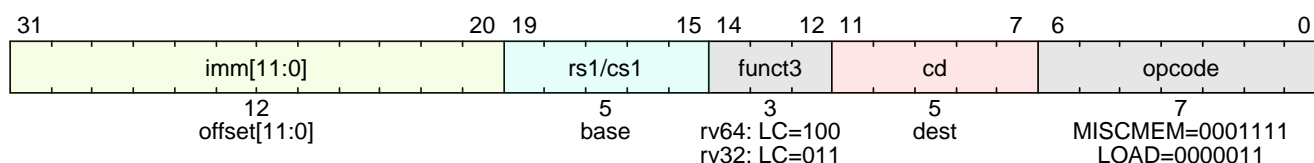
Legacy Mode Mnemonics

lc cd, offset(rs1)



These instructions have different encodings for RV64 and RV32.

Encoding



Capability Mode Description

Load a CLEN+1 bit value from memory and writes it to **cd**. The capability in **cs1** authorizes the operation. The effective address of the memory access is obtained by adding the address of **cs1** to the sign-extended 12-bit offset. The tag value written to **cd** is 0 if the tag of the memory location loaded is 0 or **cs1** does not grant [C-permission](#).

Legacy Mode Description

Loads a CLEN+1 bit value from memory and writes it to **cd**. The capability authorising the operation is [ddc](#). The effective address of the memory access is obtained by adding **rs1** to the sign-extended 12-bit offset. The tag value written to **cd** is 0 if the tag of the memory location loaded is 0 or [ddc](#) does not grant [C-permission](#).

Exceptions

Misaligned address fault exception when the effective address is not aligned to CLEN/8.

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode

Zcheri_purecap

Prerequisites for Legacy Mode

Zcheri_legacy

LC Operation

TODO

7.1.23. SC



The RV64 encoding is intended to also allocate the encoding for SQ for RV128.

Synopsis

Store capability

Capability Mode Mnemonics

sc cs2, offset(cs1)

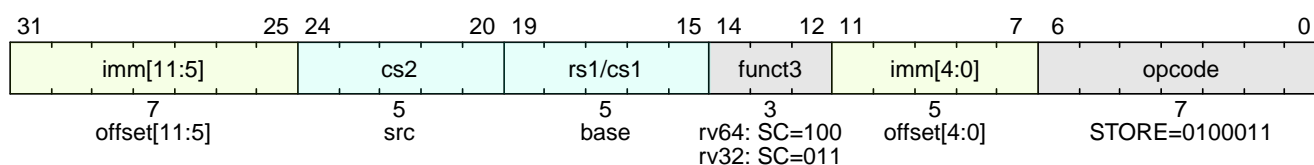
Legacy Mode Mnemonics

sc cs2, offset(rs1)



These instructions have different encodings for RV64 and RV32.

Encoding



Capability Mode Description

Store the CLEN+1 bit value in **cs2** to memory. The capability in **cs1** authorizes the operation. The effective address of the memory access is obtained by adding the address of **cs1** to the sign-extended 12-bit offset. The capability written to memory has the tag set to 0 if the tag of **cs2** is 0 or **cs1** does not grant [C-permission](#).

Legacy Mode Description

Store the CLEN+1 bit value in **cs2** to memory. The capability authorising the operation is [ddc](#). The effective address of the memory access is obtained by adding **rs1** to the sign-extended 12-bit offset. The capability written to memory has the tag set to 0 if **cs2**'s tag is 0 or [ddc](#) does not grant [C-permission](#).

Exceptions

Misaligned address fault exception when the effective address is not aligned to CLEN/8.

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode

Zcheri_purecap

Prerequisites for Legacy Mode

Zcheri_legacy

SC Operation

TODO

7.2. RV32I/E and RV64I/E Base Integer Instruction Sets

7.2.1. AUIPC

Synopsis

Add upper immediate to `pc/pcc`

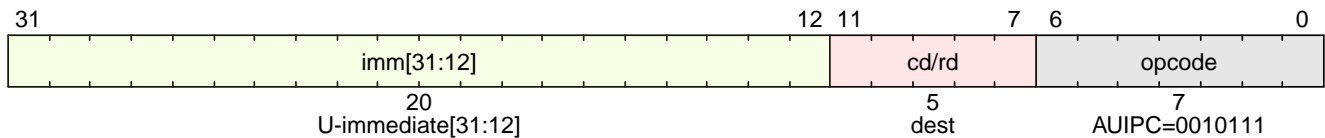
Capability Mode Mnemonic

`auipc cd, imm`

Legacy Mode Mnemonic

`auipc rd, imm`

Encoding



Capability Mode Description

Form a 32-bit offset from the 20-bit immediate filling the lowest 12 bits with zeros. Increment the address of the AUIPC instruction's `pcc` by the 32-bit offset, then write the output capability to `cd`. The tag bit of the output capability is 0 if the incremented address is outside the `pcc`'s [Representable Range](#).

Legacy Mode Description

Form a 32-bit offset from the immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register `rd`.



The instructions on this page are either PC relative or may update the `pcc`. Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the `pcc` in debug mode is UNSPECIFIED by this document.

Prerequisites for Capability Mode

`Zcheri_purecap`

Prerequisites for Legacy Mode

`Zcheri_legacy`

Operation for AUIPC

TODO

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

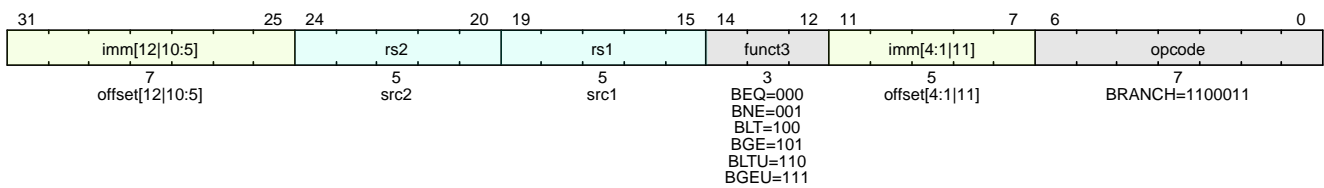
Synopsis

Conditional branches (BEQ, BNE, BLT[U], BGE[U])

Mnemonics

```
beq rs1, rs2, imm
bne rs1, rs2, imm
blt rs1, rs2, imm
bge rs1, rs2, imm
bltu rs1, rs2, imm
bgeu rs1, rs2, imm
```

Encoding



Description

Compare two integer registers `rs1` and `rs2` according to the indicated opcode as described in (RISC-V, 2023). The 12-bit immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. Then the target address is written into the address field of `pcc`.

Exceptions

When the target address is not within the `pcc`'s bounds, and the branch is taken, a *CHERI jump or branch fault* is reported in the TYPE field and Length Violation is reported in the CAUSE field of `mtval` or `stval`:



The instructions on this page are either PC relative or may update the `pcc`. Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the `pcc` in debug mode is UNSPECIFIED by this document.

ERROR: TODO: Sail doesn't have target exceptions - wrong code included?

Operation

TODO

7.2.3. JR

Expands to [JALR](#) following the expansion rule from ([RISC-V, 2023](#)).

7.2.4. JALR

Synopsis

Jump and link register

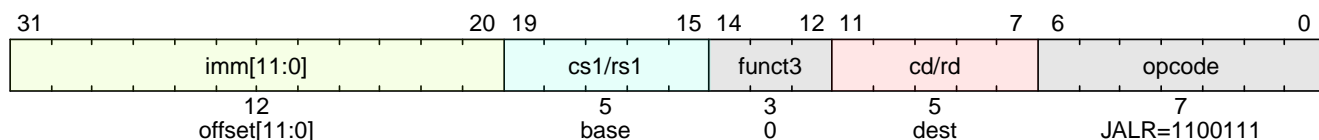
Capability Mode Mnemonic

`jalr cd, cs1, offset`

Legacy Mode Mnemonic

`jalr rd, rs1, offset`

Encoding



Capability Mode Description

JALR allows unconditional, indirect jumps to a target capability. The target capability is obtained by unsealing **cs1** if the immediate is zero and incrementing its address by the sign-extended 12-bit immediate otherwise, and then setting the least-significant bit of the result to zero. The target capability may have [Invalid address conversion](#) performed and is then installed in **pcc**. The **pcc** of the next instruction following the jump (**pcc** + 4) is sealed and written to **cd**.

Legacy Mode Description

JALR allows unconditional, indirect jumps to a target address. The target address is obtained by adding the sign-extended 12-bit immediate to **rs1**, then setting the least-significant bit of the result to zero. The target address is installed in the address field of the **pcc** which may require [Invalid address conversion](#). The address of the instruction following the jump (**pcc** + 4) is written to **rd**.

Exceptions

When these instructions cause CHERI exceptions, *CHERI jump or branch fault* is reported in the TYPE field and the following codes may be reported in the CAUSE field of [mtval](#) or [stval](#):

CAUSE	Legacy Mode	Capability Mode	Reason
Tag violation		✓	cs1 has tag set to 0
Seal violation		✓	cs1 is sealed and the immediate is not 0
Permission violation		✓	cs1 does not grant X-permission
Length violation	✓	✓	Minimum length instruction is not within the target capability's bounds. This check uses the address after it has undergone Invalid address conversion but with the original bounds.



The instructions on this page are either PC relative or may update the [pcc](#). Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the [pcc](#) in debug mode is UNSPECIFIED by this document.

Prerequisites Capability Mode

Zcheri_purecap

Prerequisites Legacy Mode

Zcheri_legacy

Operation

TBD

7.2.5. J

Expands to [JAL](#) following the expansion rule from ([RISC-V, 2023](#)).

7.2.6. JAL

Synopsis

Jump and link

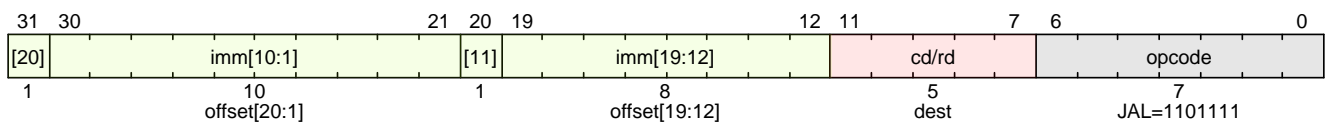
Capability Mode Mnemonic

`jal cd, offset`

Legacy Mode Mnemonic

`jal rd, offset`

Encoding



Capability Mode Description

JAL's immediate encodes a signed offset in multiple of 2 bytes. The [pcc](#) is incremented by the sign-extended offset to form the jump target capability. The target capability is written to [pcc](#). The [pcc](#) of the next instruction following the jump ([pcc](#) + 4) is sealed and written to [cd](#).

Legacy Mode Description

JAL's immediate encodes a signed offset in multiple of 2 bytes. The sign-extended offset is added to the [pcc](#)'s address to form the target address which is written to the [pcc](#)'s address field. The address of the instruction following the jump ([pcc](#) + 4) is written to [rd](#).

Exceptions

CHERI fault exceptions occur when a minimum length instruction at the target address is not within the bounds of the [pcc](#). In this case, *CHERI jump or branch fault* is reported in the TYPE field and Length Violation is reported in the CAUSE field of [mtval](#) or [stval](#).



The instructions on this page are either PC relative or may update the [pcc](#). Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the [pcc](#) in debug mode is UNSPECIFIED by this document.

Prerequisites for Capability Mode

Zcheri_purecap

Prerequisites for Legacy Mode

Zcheri_legacy

Operation

TODO

7.2.7. LD

See [LB](#).

7.2.8. LWU

See [LB](#).

7.2.9. LW

See [LB](#).

7.2.10. LHU

See [LB](#).

7.2.11. LH

See [LB](#).

7.2.12. LBU

See [LB](#).

7.2.13. LB

Synopsis

Load (LD, LW[U], LH[U], LB[U])

Capability Mode Mnemonics (RV64)

```
ld rd, offset(cs1)
lw[u] rd, offset(cs1)
lh[u] rd, offset(cs1)
lb[u] rd, offset(cs1)
```

Legacy Mode Mnemonics (RV64)

```
ld rd, offset(rs1)
lw[u] rd, offset(rs1)
lh[u] rd, offset(rs1)
lb[u] rd, offset(rs1)
```

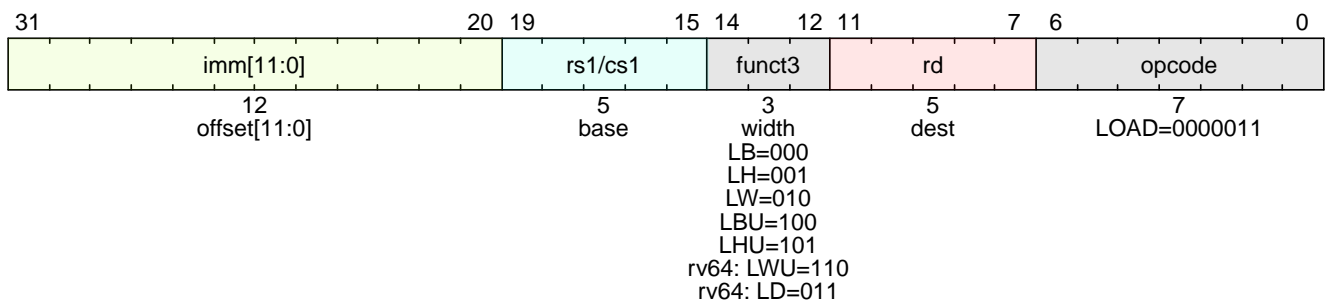
Capability Mode Mnemonics (RV32)

```
lw rd, offset(cs1)
lh[u] rd, offset(cs1)
lb[u] rd, offset(cs1)
```

Legacy Mode Mnemonics (RV32)

```
lw rd, offset(rs1)
lh[u] rd, offset(rs1)
lb[u] rd, offset(rs1)
```

Encoding



Capability Mode Description

Load integer data of the indicated size (byte, halfword, word, double-word) from memory. The effective address of the load is obtained by adding the sign-extended 12-bit offset to the address of **cs1**. The authorising capability for the operation is **cs1**. A copy of the loaded value is written to **rd**.

Legacy Mode Description

Load integer data of the indicated size (byte, halfword, word, double-word) from memory. The effective address of the load is obtained by adding the sign-extended 12-bit offset to **rs1**. The authorising capability for the operation is [ddc](#). A copy of the loaded value is written to **rd**.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode LD

RV64, Zcheri_purecap

Prerequisites for Legacy Mode LD

RV64, Zcheri_legacy

Prerequisites for Capability Mode LW[U], LH[U], LB[U]

Zcheri_purecap, OR

Zcheri_legacy

Capability Mode Operation

TBD

Legacy Mode Operation

TODO

7.2.14. SD

See [SB](#)

7.2.15. SW

See [SB](#)

7.2.16. SH

See [SB](#)

7.2.17. SB

Synopsis

Stores (SD, SW, SH, SB)

Capability Mode Mnemonics (RV64)

sd rs2, offset(cs1)
sw rs2, offset(cs1)
sh rs2, offset(cs1)
sb rs2, offset(cs1)

Legacy Mode Mnemonics (RV64)

sd rs2, offset(rs1)
sw rs2, offset(rs1)
sh rs2, offset(rs1)
sb rs2, offset(rs1)

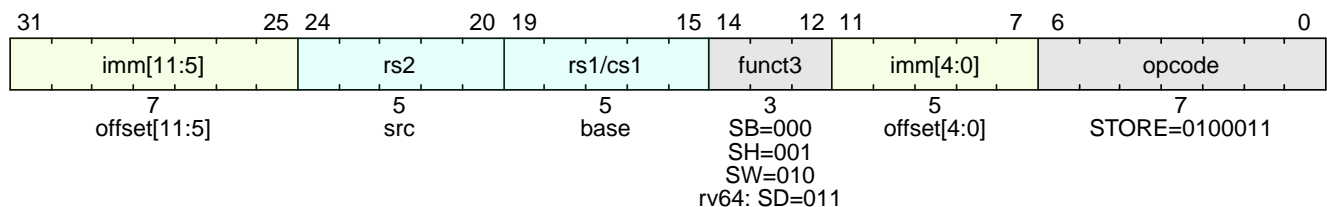
Capability Mode Mnemonics (RV32)

sw rs2, offset(cs1)
sh rs2, offset(cs1)
sb rs2, offset(cs1)

Legacy Mode Mnemonics (RV32)

sw rs2, offset(rs1)
sh rs2, offset(rs1)
sb rs2, offset(rs1)

Encoding



Capability Mode Description

Store integer data of the indicated size (byte, halfword, word, double-word) to memory. The effective address of the store is obtained by adding the sign-extended 12-bit offset to the address of **cs1**. The authorising capability for the operation is **cs1**. A copy of **rs2** is written to memory at the location indicated by the effective address and the tag bit of each block of memory naturally aligned to CLEN/8 is cleared.

Legacy Mode Description

Store integer data of the indicated size (byte, halfword, word, double-word) to memory. The effective address of the store is obtained by adding the sign-extended 12-bit offset to **rs1**. The authorising capability for the operation is **ddc**. A copy of **rs2** is written to memory at the location indicated by the effective address and the tag bit of each block of memory naturally aligned to CLEN/8 is cleared.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this

case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode SD

RV64, Zcheri_purecap

Prerequisites for Legacy Mode SD

RV64, Zcheri_legacy

Prerequisites for Capability Mode SW, SH, SB

Zcheri_purecap

Prerequisites for Legacy Mode SW, SH, SB

Zcheri_legacy

Operation

TBD

7.2.18. SRET

See [MRET](#).

7.2.19. MRET

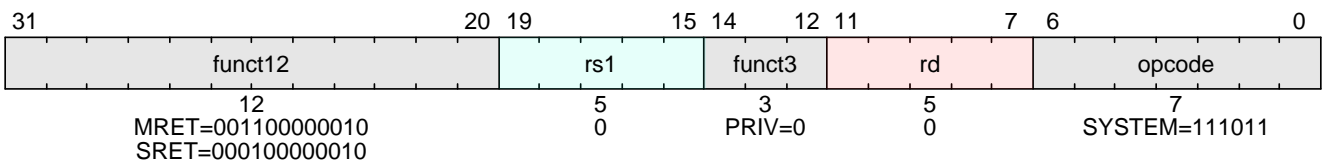
Synopsis

Trap Return (MRET, SRET)

Mnemonics

mret
sret

Encoding



Description

Return from machine mode ([MRET](#)) or supervisor mode ([SRET](#)) trap handler as defined by ([RISC-V, 2023](#)). MRET unseals [mepcc](#) and writes the result into [pcc](#). SRET unseals [sepcc](#) and writes the result into [pcc](#).

Exceptions

CHERI fault exceptions occur when [pcc](#) does not grant [ASR-permission](#) because [MRET](#) and [SRET](#) require access to privileged CSRs. When that exception occurs, *CHERI instruction access fault* is reported in the TYPE field and the Permission Violation codes is reported in the CAUSE field of [mtval](#) or [stval](#).

Operation

TBD

7.2.20. DRET

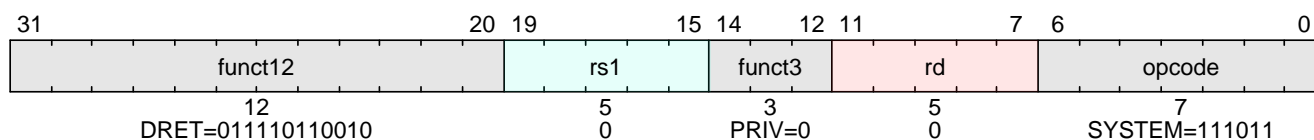
Synopsis

Debug Return (DRET)

Mnemonics

dret

Encoding



Description

[DRET](#) return from debug mode. It unseals [dpcc](#) and writes the result into [pcc](#).



The [DRET](#) instruction is the recommended way to exit debug mode. However, it is a pseudo instruction to return that technically does not execute from the program buffer or memory. It currently does not require the [pcc](#) to grant [ASR-permission](#) so it never excepts.

Prerequisites

Sdext

Operation

TBD

7.3. "A" Standard Extension for Atomic Instructions

7.3.1. AMO<OP>.W

See [AMO<OP>.D](#).

7.3.2. AMO<OP>.D

Synopsis

Atomic Operations (AMO<OP>.W, AMO<OP>.D), 32-bit encodings

Capability Mode Mnemonics (RV64)

`amo<op>.[w|d], offset(cs1)`

Capability Mode Mnemonics (RV32)

`amo<op>.w, offset(cs1)`

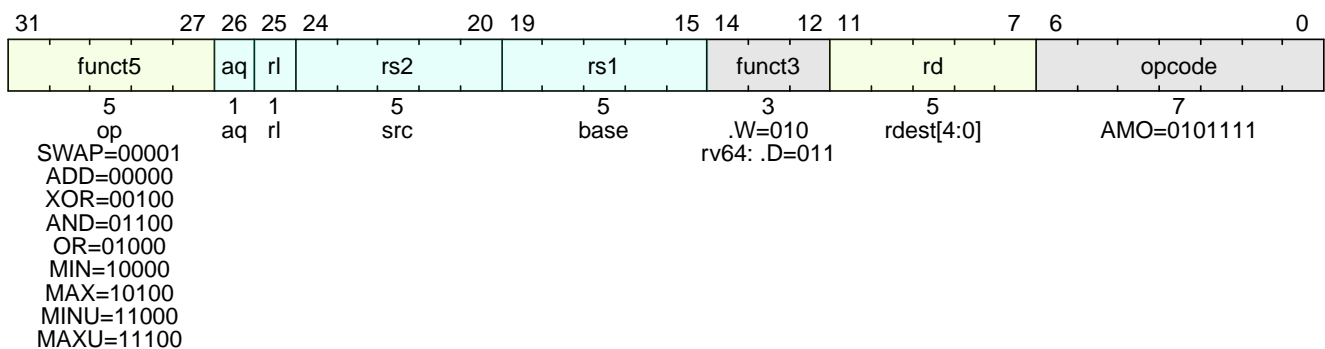
Legacy Mode Mnemonics (RV64)

`amo<op>.[w|d], offset(rs1)`

Legacy Mode Mnemonics (RV32)

`amo<op>.w, offset(rs1)`

Encoding



Capability Mode Description

Standard atomic instructions, authorised by the capability in `cs1`.

Legacy Mode Description

Standard atomic instructions, authorised by the capability in `ddc`.

Permissions

Requires [R-permission](#) and [W-permission](#) in the authorising capability.

Requires all bytes of the access to be in capability bounds.

Exceptions

All misaligned atomics cause a store/AMO address misaligned exception to allow software emulation (if the Zam extension is supported, see ([RISC-V, 2023](#))), otherwise they take a store/AMO access fault exception.

When these instructions cause CHERI exceptions, *CHERI data fault* is reported in the TYPE field and the following codes may be reported in the CAUSE field of [mtval](#) or [stval](#):

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission or W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode AMO<OP>.W, AMO<OP>.D

Zcheri_purecap, and A

Prerequisites for Legacy Mode AMO<OP>.W, AMO<OP>.D

Zcheri_legacy, and A

Capability Mode Operation

TBD

Legacy Mode Operation

TODO

7.3.3. AMOSWAP.C



The RV64 encoding is intended to also allocate the encoding for AMOSWAP.Q for RV128.

Synopsis

Atomic Operation (AMOSWAP.C), 32-bit encoding



These instructions have different encodings for RV64 and RV32.

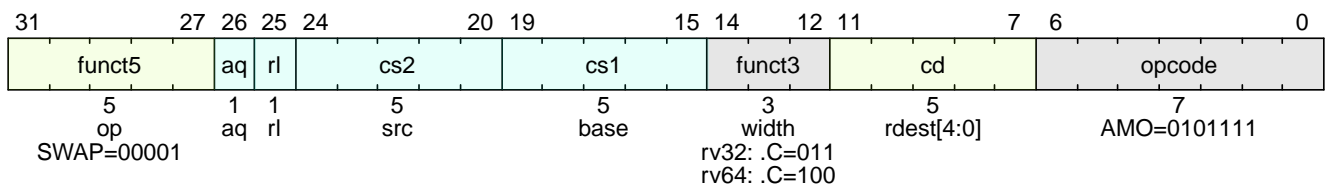
Capability Mode Mnemonics

`amoswap.c, offset(cs1)`

Legacy Mode Mnemonics

`amoswap.c, offset(rs1)`

Encoding



Capability Mode Description

Atomic swap of capability type, authorised by the capability in `cs1`.

Legacy Mode Description

Atomic swap of capability type, authorised by the capability in `ddc`.

Permissions

Requires the authorising capability to be tagged and not sealed.

Requires [R-permission](#) and [W-permission](#) in the authorising capability.

If [C-permission](#) is not granted then store the memory tag as zero, and load `cd.tag` as zero.

(This tag clearing behaviour may become a data dependent exception in future.)

Requires all bytes of the access to be in capability bounds.

Exceptions

All misaligned atomics cause a store/AMO address misaligned exception to allow software emulation (if the Zam extension is supported, see ([RISC-V, 2023](#))), otherwise they take a store/AMO access fault exception.

When these instructions cause CHERI exceptions, *CHERI data fault* is reported in the TYPE field and the following codes may be reported in the CAUSE field of `mtval` or `stval`:

CAUSE	Reason
Tag violation	Authority capability tag set to 0

CAUSE	Reason
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission or W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode AMOSWAP.C

Zcheri_purecap, and A

Prerequisites for Legacy Mode AMOSWAP.C

Zcheri_legacy, and A

Operation

TODO

7.3.4. LR.D

See [LR.B](#).

7.3.5. LR.W

See [LR.B](#).

7.3.6. LR.H

See [LR.B](#).

7.3.7. LR.B

Synopsis

Load Reserved (LR.D, LR.W, LR.H, LR.B), 32-bit encodings

Capability Mode Mnemonics (RV64)

`lr.[d|w|h|b] rd, 0(cs1)`

Capability Mode Mnemonics (RV32)

`lr.[w|h|b] rd, 0(cs1)`

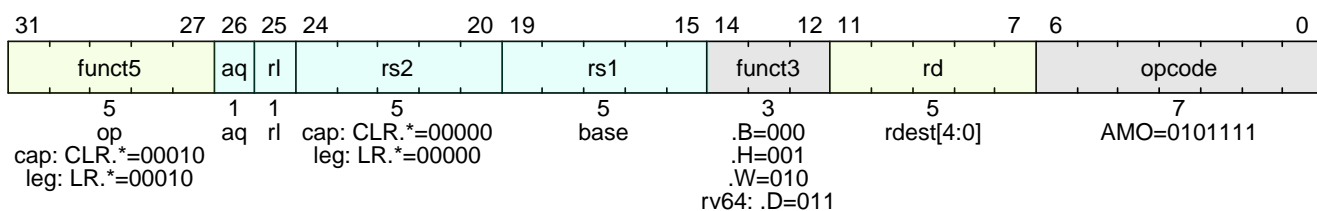
Legacy Mode Mnemonics (RV64)

`lr.[d|w|h|b] rd, 0(rs1)`

Legacy Mode Mnemonics (RV32)

`lr.[w|h|b] rd, 0(rs1)`

Encoding



Capability Mode Description

Load reserved instructions, authorised by the capability in `cs1`.

Legacy Mode Description

Load reserved instructions, authorised by the capability in `ddc`.

Exceptions

All misaligned load reservations cause a load address misaligned exception to allow software emulation (if the Zam extension is supported, see (RISC-V, 2023)), otherwise they take a load access fault exception.

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode LR.D

RV64, Zcheri_purecap, and A

Prerequisites for Capability Mode LR.W

Zcheri_purecap, and A

Prerequisites for Capability Mode LR.H, LR.B

Zabhlrsc, and Zcheri_purecap

Prerequisites for LR.D

RV64, Zcheri_legacy, and A

Prerequisites for LR.W

Zcheri_legacy, and A

Prerequisites for LR.H, LR.B

Zabhlrsc, Zcheri_legacy

Operation

TBD

7.3.8. LR.C



The RV64 encoding is intended to also allocate the encoding for LR.Q for RV128.

Synopsis

Load Reserved Capability (LR.C), 32-bit encodings



These instructions have different encodings for RV64 and RV32.

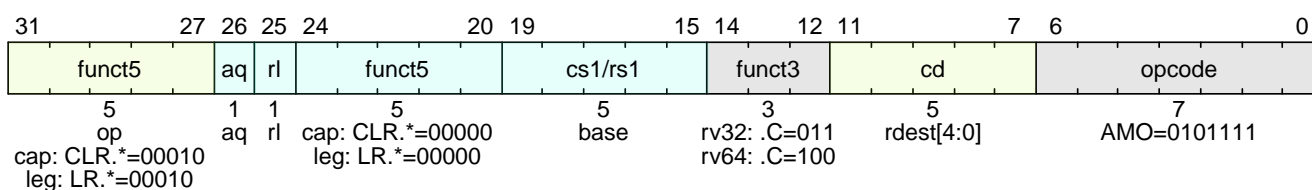
Capability Mode Mnemonics

`lr.c cd, 0(cs1)`

Legacy Mode Mnemonics

`lr.c cd, 0(rs1)`

Encoding



Capability Mode Description

Load reserved instructions, authorised by the capability in `cs1`. All misaligned load reservations cause a load address misaligned exception to allow software emulation (Zam extension, see ([RISC-V, 2023](#))).

Legacy Mode Description

Load reserved instructions, authorised by the capability in `ddc`. All misaligned load reservations cause a load address misaligned exception to allow software emulation (Zam extension, see ([RISC-V, 2023](#))).

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode

Zcheri_purecap, and A

Prerequisites for Legacy Mode

Zcheri_legacy, and A

Operation

TBD

7.3.9. SC.D

See [SC.B](#).

7.3.10. SC.W

See [SC.B](#).

7.3.11. SC.H

See [SC.B](#).

7.3.12. SC.B

Synopsis

Store Conditional (SC.D, SC.W, SC.H, SC.B), 32-bit encodings

Capability Mode Mnemonics (RV64)

`sc.[d|w|h|b] rd, rs2, 0(cs1)`

Capability Mode Mnemonics (RV32)

`sc.[w|h|b] rd, rs2, 0(cs1)`

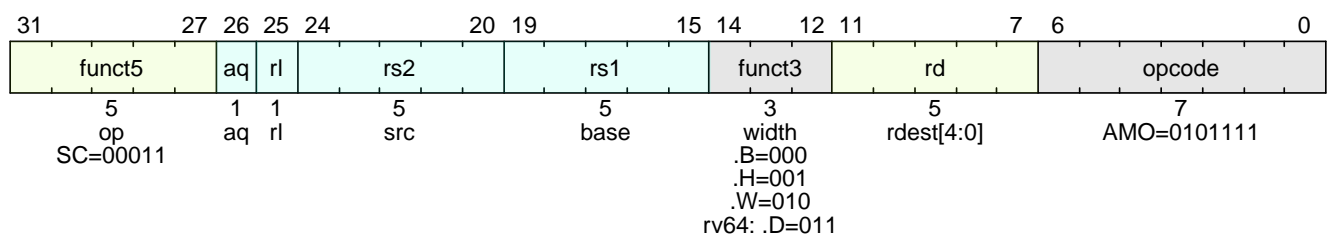
Legacy Mode Mnemonics (RV64)

`sc.[d|w|h|b] rd, rs2, 0(rs1)`

Legacy Mode Mnemonics (RV32)

`sc.[w|h|b] rd, rs2, 0(rs1)`

Encoding



Capability Mode Description

Store conditional instructions, authorised by the capability in `cs1`.

Legacy Mode Description

Store conditional instructions, authorised by the capability in `ddc`.

Exceptions

All misaligned store conditionals cause a store/AMO address misaligned exception to allow software emulation (if the Zam extension is supported, see (RISC-V, 2023)), otherwise they take a store/AMO access fault exception.

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode SC.D

RV64, and Zcheri_purecap, and A

Prerequisites for Legacy Mode SC.D

RV64, and Zcheri_legacy, and A

Prerequisites for Capability Mode SC.W

Zcheri_purecap, and A

Prerequisites for Legacy Mode SC.W

Zcheri_legacy, and A

Prerequisites for Capability Mode SC.H, SC.B

Zcheri_purecap, and Zabhlrsc

Prerequisites for Legacy Mode SC.H, SC.B

Zcheri_legacy, and Zabhlrsc

Operation

TBD

7.3.13. SC.C



The RV64 encoding is intended to also allocate the encoding for SC.Q for RV128.

Synopsis

Store Conditional (SC.C), 32-bit encoding



These instructions have different encodings for RV64 and RV32.

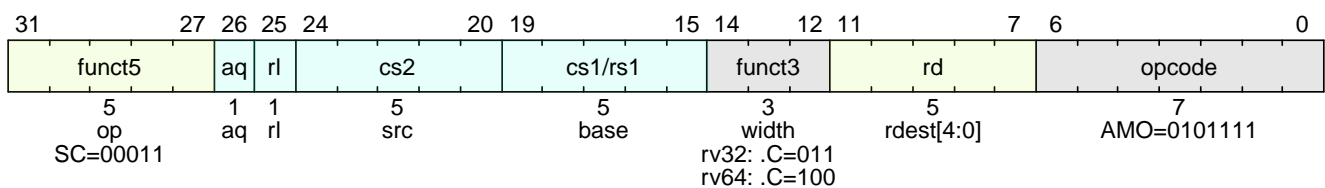
Capability Mode Mnemonics

`sc.c rd, cs2, 0(cs1)`

Legacy Mode Mnemonics

`sc.c rd, cs2, 0(rs1)`

Encoding



Capability Mode Description

Store conditional instructions, authorised by the capability in `cs1`. All misaligned store conditionals cause a store/AMO address misaligned exception to allow software emulation (Zam extension, see (RISC-V, 2023)).

Legacy Mode Description

Store conditional instructions, authorised by the capability in `ddc`. All misaligned store conditionals cause a store/AMO address misaligned exception to allow software emulation (Zam extension, see (RISC-V, 2023)).

Exceptions

All misaligned store conditionals cause a store/AMO address misaligned exception to allow software emulation (if the Zam extension is supported, see (RISC-V, 2023)), otherwise they take a store/AMO access fault exception.

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode

Zcheri_purecap, and A

Prerequisites for Legacy Mode

Zcheri_legacy, and A

Operation

TBD

7.4. "Zicsr", Control and Status Register (CSR) Instructions

7.4.1. CSRRW



CHERI v9 Note: *CSpecialRW* is removed and this functionality replaces it

Synopsis

CSR access (CSRRW) 32-bit encodings

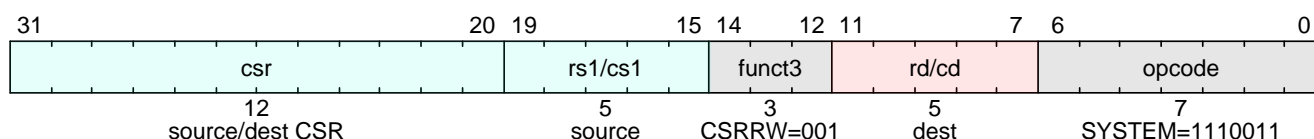
Mnemonic for accessing capability CSRs in Capability Mode

`csrrw cd, csr, cs1`

Mnemonic for accessing XLEN-wide CSRs or extended CSRs in Legacy Mode

`csrrw rd, csr, rs1`

Encoding



Description

This is a standard RISC-V CSR instructions with extended functionality for accessing CLEN-wide CSRs, such as [mtvec/mtvecc](#).

See [Table 35](#) for a list of CLEN-wide CSRs and [Table 36](#) for the action taken on writing each one.

CSRRW writes `cs1` to extended CSRs in Capability Mode, and reads a full capability into `cd`.

CSRRW writes `rs1` to extended CSRs in Legacy Mode, and reads the address field into `rd`.

If `cd` is `c0` (or `rd` is `x0`), then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read.

The assembler pseudoinstruction to write a capability CSR in Capability Mode, `csrw csr, cs1`, is encoded as `csrrw c0, csr, cs1`.

Access to XLEN-wide CSRs from other extensions is as specified by RISC-V.

Permissions

Accessing privileged CSRs require [ASR-permission](#), including existing RISC-V CSRs, as described in [Section 3.5.1](#). The list of privileged and unprivileged CSRs is shown in ([RISC-V, 2023](#)).

Prerequisites for Capability Mode

Zcheri_purecap

Prerequisites for Legacy Mode

Zcheri_legacy

Operation

TBD

7.4.2. CSRRWI

See [CSRRCI](#).

7.4.3. CSRRS

See [CSRRCI](#).

7.4.4. CSRRSI

See [CSRRCI](#).

7.4.5. CSRRC

See [CSRRCI](#).

7.4.6. CSRRCI



CHERI v9 Note: *CSpecialRW* is removed and this functionality replaces it

Synopsis

CSR access (CSRRWI, CSRRS, CSRRSI, CSRRC, CSRRCI) 32-bit encodings

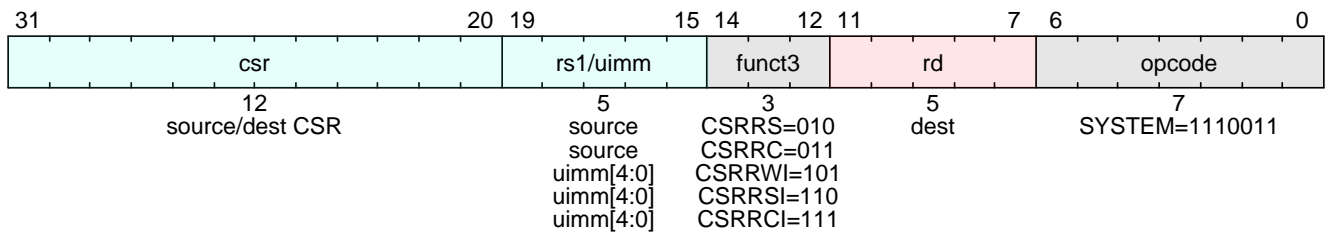
Mnemonics for accessing capability CSRs in Capability Mode

```
csrrs cd, csr, rs1
csrrc cd, csr, rs1
csrrwi cd, csr, imm
csrrsi cd, csr, imm
csrrci cd, csr, imm
```

Mnemonics for accessing XLEN-wide CSRs or extended CSRs in Legacy Mode

```
csrrs rd, csr, rs1
csrrc rd, csr, rs1
csrrwi rd, csr, imm
csrrsi rd, csr, imm
csrrci rd, csr, imm
```

Encoding



Description

These are standard RISC-V CSR instructions with extended functionality for accessing capability CSRs, such as [mtvec/mtvecc](#).

Unlike [CSRRW](#), these instructions only update the address field and the tag as defined in [Table 36](#) when writing capability CSRs regardless of the execution mode. The final address to write to the capability CSR is determined as defined by RISC-V for these instructions.

See [Table 35](#) for a list of capability CSRs and [Table 36](#) for the action taken on writing an XLEN-wide value to each one.

If **cd** is **c0** (or **rd** is **x0**), then [CSRRWI](#) shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read. If **rs1** is **x0** for [CSRRS](#) and [CSRRC](#), or **imm** is 0 for [CSRRSI](#) and [CSRRCI](#), then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write.

The assembler pseudoinstruction to read a capability CSR in Capability Mode, **csrr rd, csr**, is encoded as **csrrs cd, csr, c0**.

Access to XLEN-wide CSRs is as specified by RISC-V.



If the CSR accessed is a capability, and **rs1** is **x0** for [CSRRS](#) and [CSRRC](#), or **imm** is 0 for [CSRRSI](#) and [CSRRCI](#), then the CSR is not written so no representability check is needed in

this case.

Permissions

Accessing privileged CSRs requires [ASR-permission](#), including existing RISC-V CSRs, as described in [Section 3.5.1](#). The list of privileged and unprivileged CSRs is shown in ([RISC-V, 2023](#)).

Prerequisites for Capability Mode

Zcheri_purecap

Prerequisites for Legacy Mode

Zcheri_legacy

Operation

TBD

7.5. "Zfh", "Zfhmin", "F" and "D" Standard Extension for Floating-Point

7.5.1. FLD

See [FLH](#).

7.5.2. FLW

See [FLH](#).

7.5.3. FLH

Synopsis

Floating point loads (FLD, FLW, FLH), 32-bit encodings

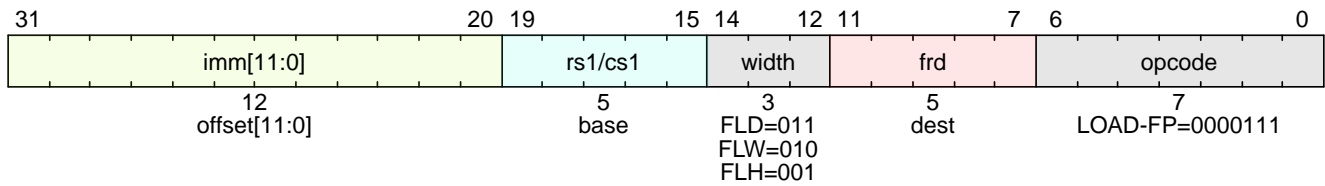
Capability Mode Mnemonics

`fld/flw/flh frd, offset(cs1)`

Legacy Mode Mnemonics

`fld/flw/flh rd, offset(rs1)`

Encoding



Capability Mode Description

Standard floating point load instructions, authorised by the capability in `cs1`.

Legacy Mode Description

Standard floating point load instructions, authorised by the capability in `ddc`.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode FLD

`Zcheri_purecap`, and `D`

Prerequisites for Legacy Mode FLD

`Zcheri_legacy`, and `D`

Prerequisites for Capability Mode FLW

`Zcheri_purecap`, and `F`

Prerequisites for Legacy Mode FLW

`Zcheri_legacy`, and `F`

Prerequisites for Capability Mode FLH

`Zcheri_purecap`, and `Zfhmin` or `Zfh`

Prerequisites for Legacy Mode FLH

Zcheri_legacy, and Zfhmin or Zfh

Operation

TODO

7.5.4. FSD

See [FSH](#).

7.5.5. FSW

See [FSH](#).

7.5.6. FSH

Synopsis

Floating point stores (FSD, FSW, FSH), 32-bit encodings

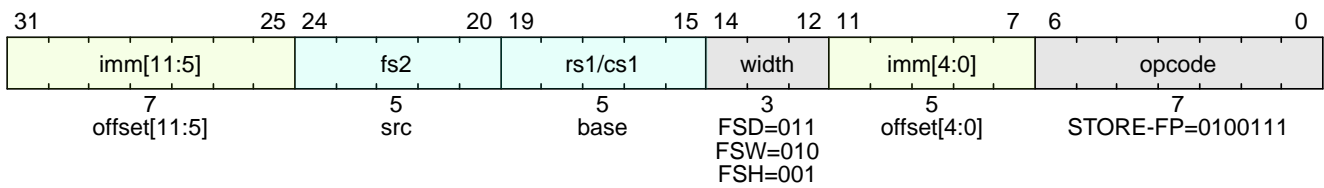
Capability Mode Mnemonics

`fsd/fsw/fsh fs2, offset(cs1)`

Legacy Mode Mnemonics

`fsd/fsw/fsh fs2, offset(rs1)`

Encoding



Capability Mode Description

Standard floating point store instructions, authorised by the capability in `cs1`.

Legacy Mode Description

Standard floating point store instructions, authorised by the capability in `ddc`.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode FSD

Zcheri_purecap, and D

Prerequisites for Legacy Mode FSD

Zcheri_legacy, and D

Prerequisites for Capability Mode FSW

Zcheri_purecap, and F

Prerequisites for Legacy Mode FSW

Zcheri_legacy, and F

Prerequisites for Capability Mode FSH

Zcheri_purecap, and Zfh or Zfhmin

Prerequisites for Legacy Mode FSH

Zcheri_legacy, and Zfh or Zfhmin

Operation

TBD

7.6. "C" Standard Extension for Compressed Instructions

7.6.1. C.BEQZ, C.BNEZ

Synopsis

Conditional branches (C.BEQZ, C.BNEZ), 16-bit encodings

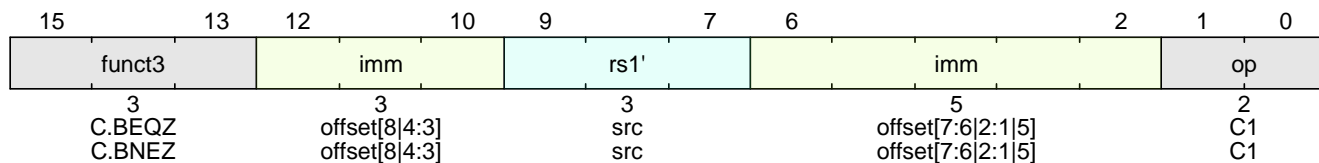
Mnemonics

c.beqz/c.bnez rs1', offset

Expansions

beq/bne rs1', x0, offset

Encoding



Exceptions

When the target address is not within the [pcc](#)'s bounds, and the branch is taken, a *CHERI jump or branch fault* is reported in the TYPE field and Length Violation is reported in the CAUSE field of [mtval](#) or [stval](#):



The instructions on this page are either PC relative or may update the [pcc](#). Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the [pcc](#) in debug mode is UNSPECIFIED by this document.

Prerequisites

C or Zca

Operation (after expansion to 32-bit encodings)

See [Conditional branches \(BEQ, BNE, BLT\[U\], BGE\[U\]\)](#)

7.6.2. C.MV

Synopsis

Capability move (C.MV), 16-bit encoding

Capability Mode Mnemonic

c.mv cd, cs2

Capability Mode Expansion

cmv cd, cs2

Suggested assembly syntax

mv rd, rs2
mv cd, cs2



the suggested assembly syntax distinguishes from integer mv by operand type.

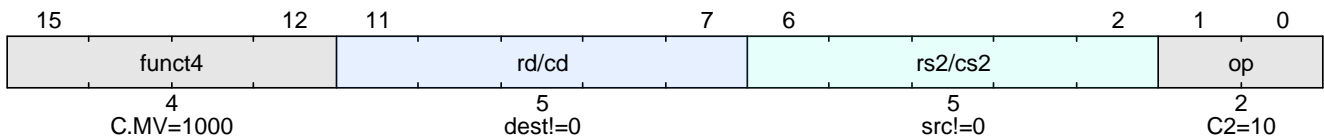
Legacy Mode Mnemonic

c.mv rd, rs2

Legacy Mode Expansion

add rd, x0, rs2

Encoding



Capability Mode Description

Capability register **cd** is replaced with the contents of **cs2**.

Legacy Mode Description

Standard RISC-V [C.MV](#) instruction.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy

Capability Mode Operation (after expansion to 32-bit encodings)

See [CMV](#)

7.6.3. C.ADDI16SP

Synopsis

Stack pointer increment in blocks of 16 (C.ADDI16SP), 16-bit encodings

Capability Mode Mnemonic

c.addi16sp imm

Capability Mode Expansion

cadd csp, csp, imm

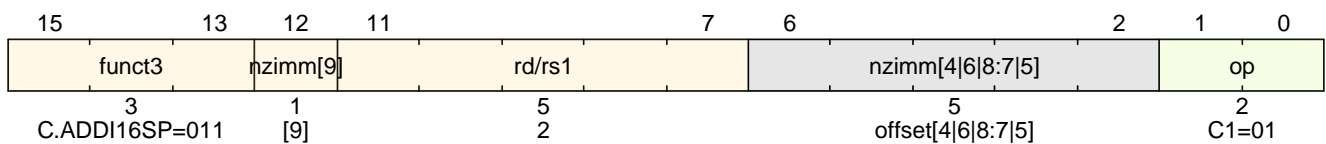
Legacy Mode Mnemonic

c.addi16sp imm

Legacy Mode Expansion

add sp, sp, imm

Encoding



Capability Mode Description

Add the non-zero sign-extended 6-bit immediate to the value in the stack pointer (**csp=c2**), where the immediate is scaled to represent multiples of 16 in the range (-512,496). Clear the tag if the resulting capability is unrepresentable or **csp** is sealed.

Legacy Mode Description

Add the non-zero sign-extended 6-bit immediate to the value in the stack pointer (**sp=x2**), where the immediate is scaled to represent multiples of 16 in the range (-512,496).

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy

Capability Mode Operation

TODO

7.6.4. C.ADDI4SPN

See [C.ADDI4SPN](#).

Synopsis

Stack pointer increment in blocks of 4 (C.ADDI4SPN), 16-bit encoding

Capability Mode Mnemonic

`c.addi4spn cd', uimm`

Capability Mode Expansion

`cadd cd', csp, uimm`

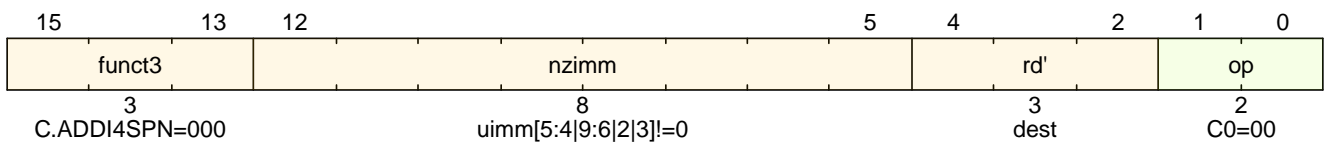
Legacy Mode Mnemonic

`c.addi4spn rd', uimm`

Legacy Mode Expansion

`add rd', sp, uimm`

Encoding



Capability Mode Description

Add a zero-extended non-zero immediate, scaled by 4, to the stack pointer, `csp`, and writes the result to `cd'`. This instruction is used to generate pointers to stack-allocated variables. Clear the tag if the resulting capability is unrepresentable or `csp` is sealed.

Legacy Mode Description

Add a zero-extended non-zero immediate, scaled by 4, to the stack pointer, `sp`, and writes the result to `rd'`. This instruction is used to generate pointers to stack-allocated variables.

Prerequisites for C.ADDI4SPN

C or Zca, Zcheri_purecap

Prerequisites for C.ADDI4SPN

C or Zca, Zcheri_legacy

Capability Mode Operation

TODO

7.6.5. C.MODESW



CHERI v9 Note: This instruction is new.

Synopsis

Capability/Legacy Mode switching (C.MODESW), 16-bit encoding

Mnemonics

`c.modesw`

Expansions

`modesw`

Encoding

15	13	12	10	9	7	6	5	4	2	1	0
1	0	0	1	1	1	0	0	0	1	1	1
3			3			3			2		3
FUNCT3			FUNCT3			FUNCT3			FUNCT2		C.MODESW
											C1=1

Description

Toggle the hart's current CHERI execution mode in [pcc](#). If the current mode in [pcc](#) is Legacy, then the mode bit (M) in [pcc](#) is set to Capability. If the current mode is Capability, then the mode bit (M) in [pcc](#) is set to Legacy.

In debug mode MODESW can still be used to change the operating mode, and the current mode is shown in the M bit of [dinfo](#).

Exceptions

None

Prerequisites

C or Zca, Zcheri_legacy

Operation (after expansion to 32-bit encodings)

See [MODESW](#)

7.6.6. CJALR

Synopsis

Register based jumps with link, 16-bit encodings

Capability Mode Mnemonic

`c.jalr c1, cs1`

Capability Mode Expansion

`jalr c1, 0(cs1)`

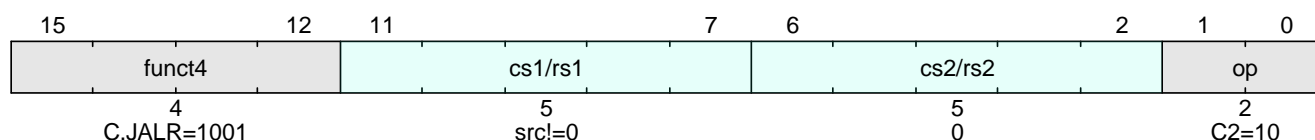
Legacy Mode Mnemonic

`c.jalr x1, rs1`

Legacy Mode Expansion

`jalr x1, 0(rs1)`

Encoding



Capability Mode Description

Link the next linear [pcc](#) to `cd` and seal. Jump to `cs1.address+offset`. [pcc](#) metadata is copied from `cs1`, and is unsealed if necessary. Note that execution has several exception checks.

Legacy Mode Description

Set the next PC and link to `rd` according to the standard [JALR](#) definition. Check a minimum length instruction is in [pcc](#) bounds at the target PC, take a CHERI Length Violation exception on error.

Exceptions

See [JALR](#)



The instructions on this page are either PC relative or may update the [pcc](#). Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the [pcc](#) in debug mode is UNSPECIFIED by this document.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy

Operation (after expansion to 32-bit encodings)

See [JALR](#)

7.6.7. CJR

Synopsis

Register based jumps without link, 16-bit encodings

Capability Mode Mnemonic

`c.jr cs1`

Capability Mode Expansion

`jalr c0, 0(cs1)`

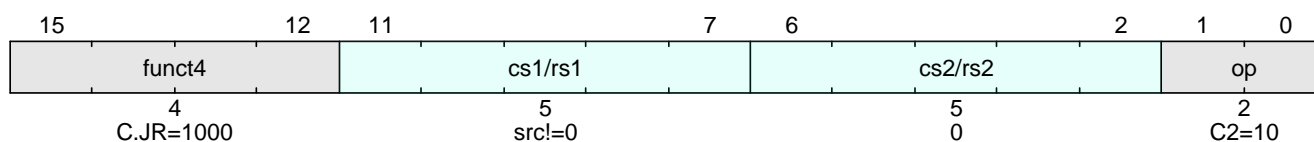
Legacy Mode Mnemonic

`c.jr rs1`

Legacy Mode Expansion

`jalr x0, 0(rs1)`

Encoding



Capability Mode Description

Jump to **cs1.address+offset**. [pcc](#) metadata is copied from **cs1**, and is unsealed if necessary. Note that execution has several exception checks.

Legacy Mode Description

Set the next PC according to the standard **jalr** definition. Check a minimum length instruction is in [pcc](#) bounds at the target PC, take a CHERI Length Violation exception on error.

Exceptions

See [JALR](#)



The instructions on this page are either PC relative or may update the [pcc](#). Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the [pcc](#) in debug mode is UNSPECIFIED by this document.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy

Operation (after expansion to 32-bit encodings)

See [JALR](#)

7.6.8. C.JAL

Synopsis

Register based jumps with link, 16-bit encodings

Capability Mode Mnemonic (RV32)

`c.jal c1, offset`

Capability Mode Expansion (RV32)

`jal c1, offset`

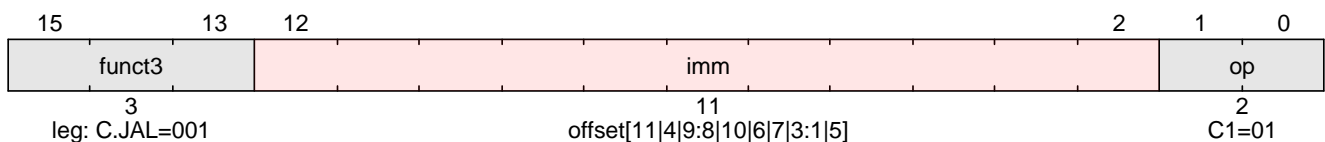
Legacy Mode Mnemonic (RV32)

`c.jal x1, offset`

Legacy Mode Expansion (RV32)

`jal x1, offset`

Encoding (RV32)



Capability Mode Description

Link the next linear [pcc](#) to **cd** and seal. Jump to [pcc](#).address+offset. Check a minimum length instruction is in [pcc](#) bounds at the target PC, take a CHERI Length Violation exception on error.

Legacy Mode Description

Set the next PC and link to **rd** according to the standard [JAL](#) definition. Check a minimum length instruction is in [pcc](#) bounds at the target PC, take a CHERI Length Violation exception on error.

Exceptions

See [JAL](#)



The instructions on this page are either PC relative or may update the [pcc](#). Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the [pcc](#) in debug mode is UNSPECIFIED by this document.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy

Operation (after expansion to 32-bit encodings)

See [JAL](#)

7.6.9. C.J

Synopsis

Register based jumps without link, 16-bit encodings

Mnemonic

`c.j offset`

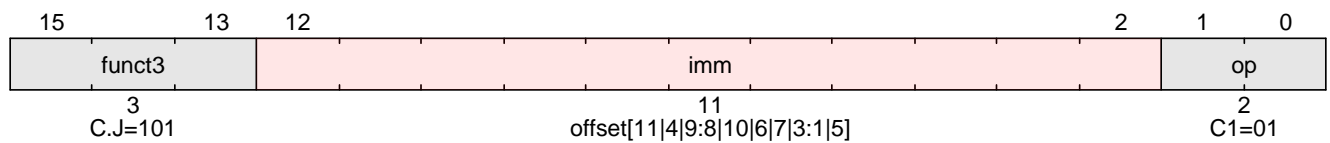
Capability Mode Expansion

`jal c0, offset`

Legacy Mode Expansion

`jal x0, offset`

Encoding



Description

Set the next PC following the standard `jal` definition. Check a minimum length instruction is in `pcc` bounds at the target PC, take a CHERI Length Violation exception on error. There is no difference in Capability Mode or Legacy Mode execution for this instruction.

Exceptions

See [JAL](#)



The instructions on this page are either PC relative or may update the `pcc`. Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the `pcc` in debug mode is UNSPECIFIED by this document.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy

Operation (after expansion to 32-bit encodings)

See [JAL](#)

7.6.10. C.LD

See [C.LW](#).

7.6.11. C.LW

Synopsis

Load (C.LD, C.LW), 16-bit encodings

Capability Mode Mnemonics (RV64)

`c.ld/c.lw rd', offset(cs1')`

Capability Mode Expansions (RV64)

`ld/lw rd', offset(cs1')`

Legacy Mode Mnemonics (RV64)

`c.ld/c.lw rd', offset(rs1')`

Legacy Mode Expansions (RV64)

`ld/lw rd', offset(rs1')`

Capability Mode Mnemonics (RV32)

`c.lw rd', offset(cs1')`

Capability Mode Expansions (RV32)

`lw rd', offset(cs1')`

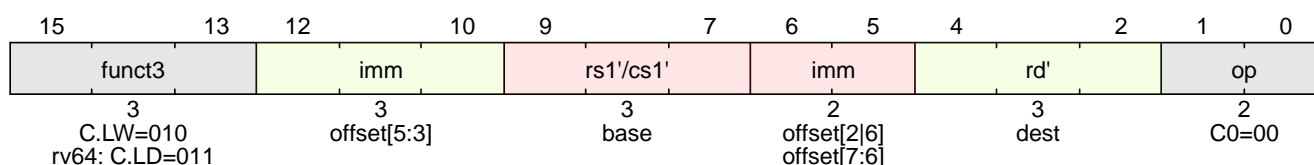
Legacy Mode Mnemonics (RV32)

`c.lw rd', offset(rs1')`

Legacy Mode Expansions (RV32)

`lw rd', offset(rs1')`

Encoding



Capability Mode Description

Standard load instructions, authorised by the capability in `cs1`.

Legacy Mode Description

Standard load instructions, authorised by the capability in `ddc`.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed

CAUSE	Reason
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode C.LD

RV64, and C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode C.LD

RV64, C or Zca, Zcheri_legacy

Prerequisites Capability Mode C.LW

C or Zca, Zcheri_purecap

Prerequisites Legacy Mode C.LW

C or Zca, Zcheri_legacy

Operation (after expansion to 32-bit encodings)

See [LD](#), [LW](#)

7.6.12. C.LWSP

See [C.LDSP](#).

7.6.13. C.LDSP

Synopsis

Load (C.LWSP, C.LDSP), 16-bit encodings

Capability Mode Mnemonics (RV64)

`c.ld/c.lw rd, offset(csp)`

Capability Mode Expansions (RV64)

`ld/lw rd, offset(csp)`

Legacy Mode Mnemonics (RV64)

`c.ld/c.lw rd, offset(sp)`

Legacy Mode Expansions (RV64)

`ld/lw rd, offset(sp)`

Capability Mode Mnemonics (RV32)

`c.lw rd, offset(csp)`

Capability Mode Expansions (RV32)

`lw rd, offset(csp)`

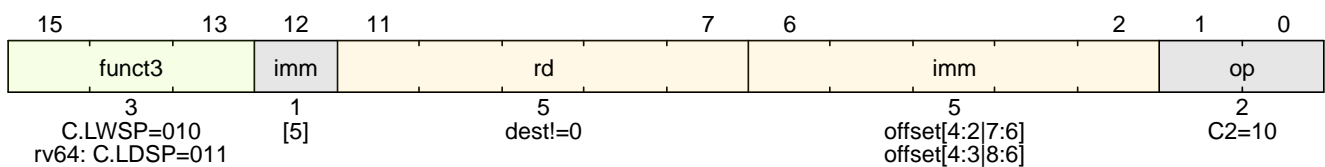
Legacy Mode Mnemonics (RV32)

`c.lw rd, offset(sp)`

Legacy Mode Expansions (RV32)

`lw rd, offset(sp)`

Encoding



Capability Mode Description

Standard stack pointer relative load instructions, authorised by the capability in `csp`.

Legacy Mode Description

Standard stack pointer relative load instructions, authorised by the capability in `ddc`.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed

CAUSE	Reason
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode C.LDSP

RV64, and C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode C.LDSP

RV64, and C or Zca, Zcheri_legacy

Prerequisites for Capability Mode C.LWSP

C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode C.LWSP

C or Zca, Zcheri_legacy

Operation (after expansion to 32-bit encodings)

See [LW](#), [LD](#)

7.6.14. C.FLW

See [C.FLWSP](#).

7.6.15. C.FLWSP

Synopsis

Floating point load (C.FLW, C.FLWSP), 16-bit encodings

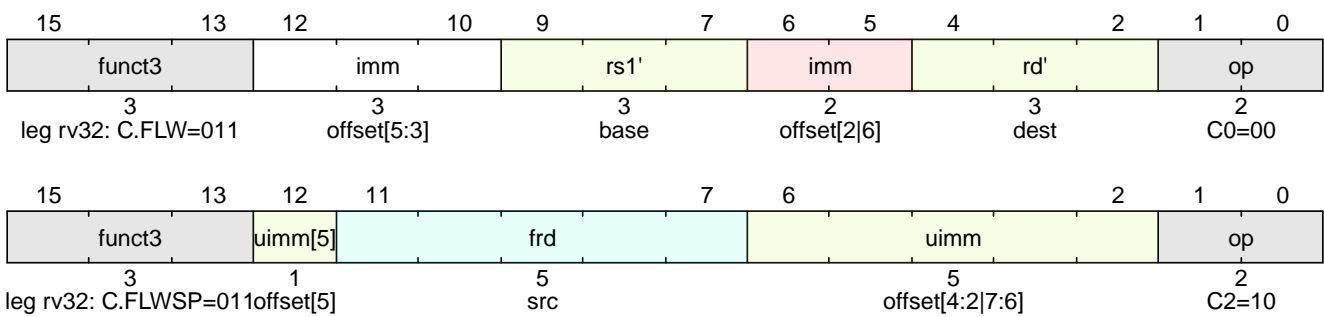
Legacy Mode Mnemonics (RV32)

`c.flw rd', offset(rs1'/sp)`

Legacy Mode Expansions (RV32)

`flw rd', offset(rs1'/sp)`

Encoding (RV32)



Legacy Mode Description

Standard floating point load instructions, authorised by the capability in [ddc](#). Note that these instructions are not available in Capability Mode, as they have been remapped to [C.LC](#), [C.LCSP](#).

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, and Zcf or F

Operation (after expansion to 32-bit encodings)

See [FLW](#)

7.6.16. C.FLD

7.6.17. C.FLDSP

Synopsis

Double precision floating point loads (C.FLD, C.FLDSP), 16-bit encodings

Capability Mode Mnemonics (RV32)

`c.fld frd', offset(cs1'/csp)`

Capability Mode Expansions (RV32)

`fld frd', offset(csp)`

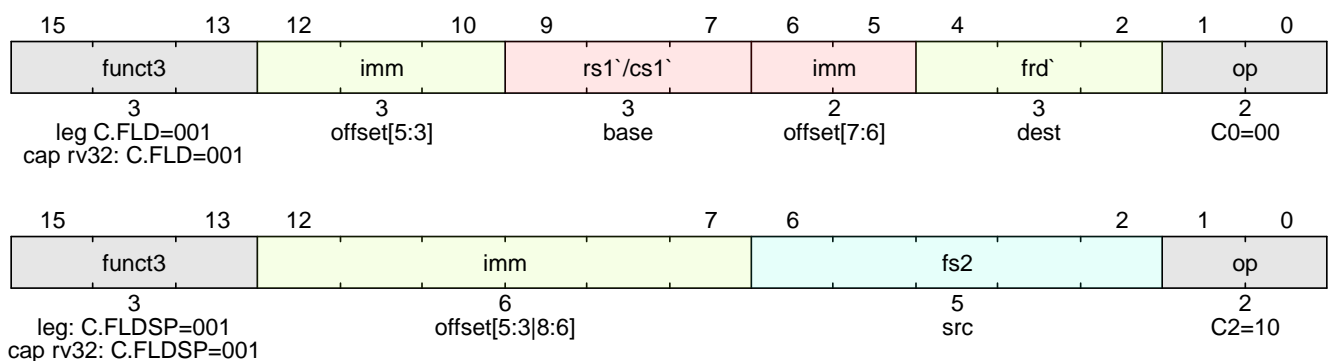
Legacy Mode Mnemonics

`c.fld fs2, offset(rs1'/sp)`

Legacy Mode Expansions

`fld fs2, offset(rs1'/sp)`

Encoding



Legacy Mode Description

Standard floating point stack pointer relative load instructions, authorised by the capability in [ddc](#). Note that these instructions are not available in Capability Mode, as they have been remapped to [C.LC](#), [C.LCSP](#).

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode (RV32 only)

Zcheri_purecap, C and D; or

Zcheri_purecap, Zca and Zcd

Prerequisites for Legacy Mode

Zcheri_legacy, C and D; or
Zcheri_legacy, Zca and Zcd

Operation (after expansion to 32-bit encodings)

See [FLD](#)

7.6.18. C.LC

see [C.LCSP](#).

7.6.19. C.LCSP

Synopsis

Capability loads (C.LC, C.LCSP), 16-bit encodings

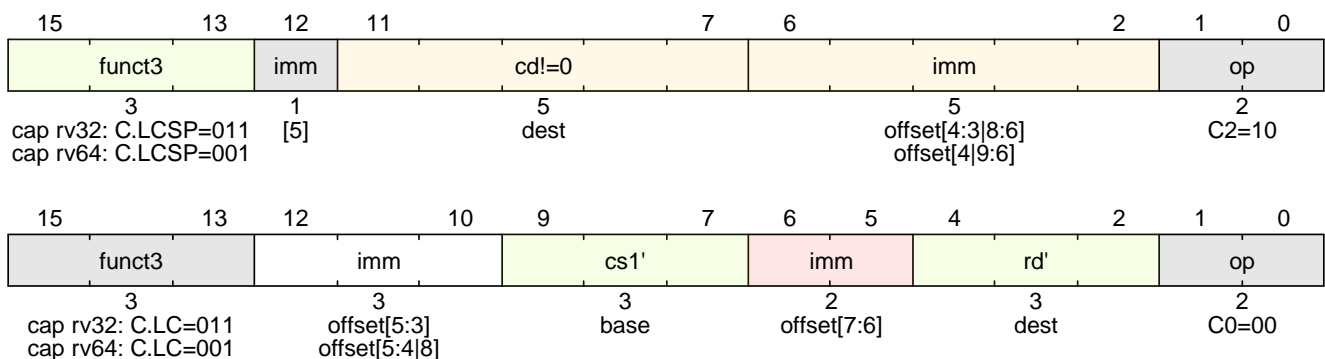
Capability Mode Mnemonics

`c.lc cd', offset(cs1'/csp)`

Capability Mode Expansions

`lc cd', offset(cs1'/csp)`

Encoding



Capability Mode Description

Load capability instruction, authorised by the capability in `cs1`. Take a load address misaligned exception if not naturally aligned.

Legacy Mode Description

These mnemonics do not exist in Legacy Mode. The RV32 encodings map to [C.FLW/C.FLWSP](#) and the RV64 encodings map to [C.FLD/C.FLDSP](#).

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites

C or Zca, Zcheri_purecap

Operation (after expansion to 32-bit encodings)

See [LC](#)

7.6.20. C.SD

See [C.SW](#).

7.6.21. C.SW

Synopsis

Stores (C.SD, C.SW), 16-bit encodings

Capability Mode Mnemonics (RV64)

c.sd/c.sw rs2', offset(cs1')

Capability Mode Expansions (RV64)

sd/sw rs2', offset(cs1')

Legacy Mode Mnemonics (RV64)

c.sd/c.sw rs2', offset(rs1')

Legacy Mode Expansions (RV64)

sd/sw rs2', offset(rs1')

Capability Mode Mnemonics (RV32)

c.sw rs2', offset(cs1')

Capability Mode Expansion (RV32)

sw rs2', offset(cs1')

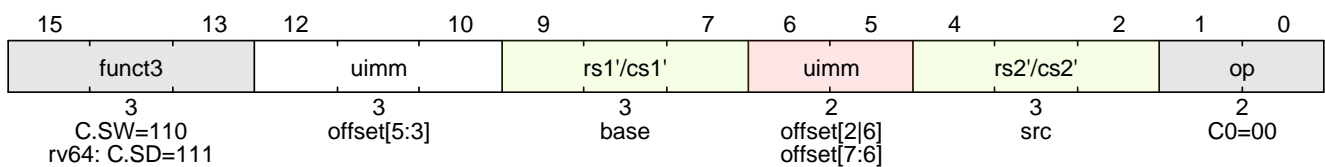
Legacy Mode Mnemonics (RV32)

c.sw rs2', offset(rs1')

Legacy Mode Expansion (RV32)

sw rs2', offset(rs1')

Encoding



Capability Mode Description

Standard store instructions, authorised by the capability in **cs1**.

Legacy Mode Description

Standard store instructions, authorised by the capability in **ddc**.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the **mtval** or **stval** TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed

CAUSE	Reason
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode C.SD

RV64, and C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode C.SD

RV64, and C or Zca, Zcheri_legacy

Prerequisites for Capability Mode C.SW

C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode C.SW

C or Zca, Zcheri_legacy

Operation (after expansion to 32-bit encodings)

See [SD](#), [SW](#)

7.6.22. C.SWSP

See [C.SDSP](#).

7.6.23. C.SDSP

Synopsis

Stack pointer relative stores (C.SWSP, C.SDSP), 16-bit encodings

Capability Mode Mnemonics (RV64)

`c.sd/c.sw rs2, offset(csp)`

Capability Mode Expansions (RV64)

`sd/csw rs2, offset(csp)`

Legacy Mode Mnemonics (RV64)

`c.sd/c.sw rs2, offset(sp)`

Legacy Mode Expansions (RV64)

`sd/sw rs2, offset(sp)`

Capability Mode Mnemonics (RV32)

`c.sw rs2, offset(csp)`

Capability Mode Expansion (RV32)

`sw rs2, offset(csp)`

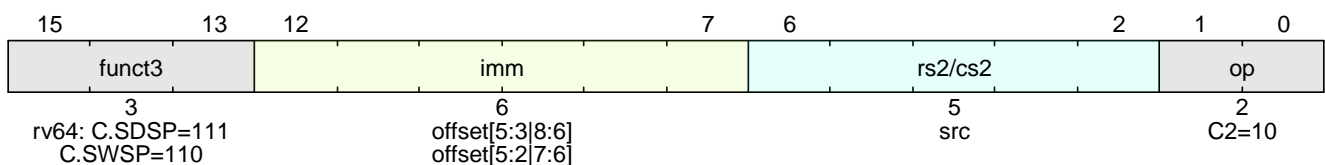
Legacy Mode Mnemonics (RV32)

`c.sw rs2, offset(sp)`

Legacy Mode Expansion (RV32)

`sw rs2, offset(sp)`

Encoding



Capability Mode Description

Standard stack pointer relative store instructions, authorised by the capability in `csp`.

Legacy Mode Description

Standard stack pointer relative store instructions, authorised by the capability in `ddc`.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed

CAUSE	Reason
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode C.SDSP

RV64, and C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode C.SDSP

RV64, and C or Zca, Zcheri_legacy

Prerequisites for Capability Mode C.SWSP

C or Zca, Zcheri_purecap

Prerequisites for Legacy Mode C.SWSP

C or Zca, Zcheri_legacy

Operation (after expansion to 32-bit encodings)

See [SD](#), [SW](#)

7.6.24. C.FSW

See [C.FSWSP](#).

7.6.25. C.FSWSP

Synopsis

Floating point stores (C.FSW, C.FSWSP), 16-bit encodings

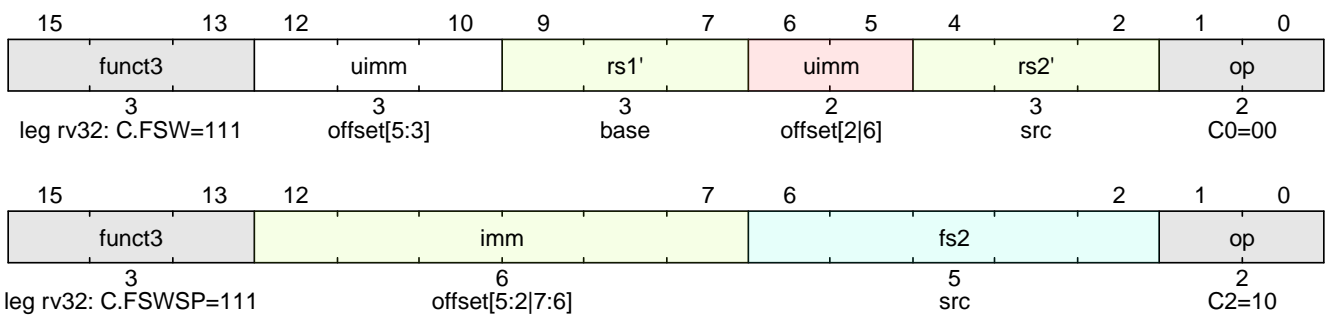
Legacy Mode Mnemonics (RV32)

`c.fsw rs2', offset(rs1'/sp)`

Legacy Mode Expansions (RV32)

`fsw rs2', offset(rs1'/sp)`

Encoding (RV32)



Legacy Mode Description

Standard floating point store instructions, authorised by the capability in [ddc](#).



these instructions are not available in Capability Mode, as they have been remapped to [C.SC](#), [C.SCSP](#).

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites

C or Zca, Zcheri_legacy, Zcf or F

Operation (after expansion to 32-bit encodings)

See [FSW](#)

7.6.26. C.FSD

See [C.FSDSP](#).

7.6.27. C.FSDSP

Synopsis

Double precision floating point stores (C.FSD, C.FSDSP), 16-bit encodings

Capability Mode Mnemonics (RV32CD/RV32D_Zca)

`c.fsd fs2, offset(cs1'/csp)`

Capability Mode Expansions (RV32)

`fsd fs2, offset(csp)`

Legacy Mode Mnemonics (RV32CD/RV32D_Zca)

`c.fsd fs2, offset(rs1'/sp)`

Legacy Mode Expansions (RV32)

`fsd fs2, offset(rs1'/sp)`

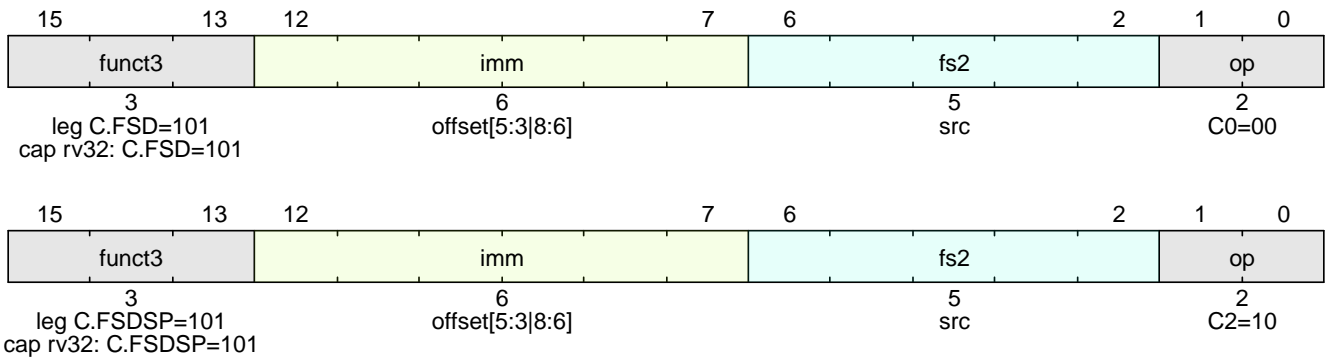
Legacy Mode Mnemonics (RV64CD/RV64D_Zca)

`c.fsd fs2, offset(rs1'/sp)`

Legacy Mode Expansion (RV64)

`fsd fs2, offset(rs1'/sp)`

Encoding



Capability Mode Description

Standard floating point stack pointer relative store instructions, authorised by the capability in `cs1` or `csp`.

Legacy Mode Description

Standard floating point stack pointer relative store instructions, authorised by the capability in [ddc](#).

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode C.FSD, C.FSDSP (RV32 only)

Zcheri_purecap, C and D; or
Zcheri_purecap, Zca and Zcd

Prerequisites for Legacy Mode C.FSD, C.FSDSP

Zcheri_legacy, C and D; or
Zcheri_legacy, Zca and Zcd

Operation (after expansion to 32-bit encodings)

See [FSD](#)

7.6.28. C.SC

see [C.SCSP](#).

7.6.29. C.SCSP

Synopsis

Stores (C.SC, C.SCSP), 16-bit encodings

 These instructions have different encodings for RV64 and RV32.

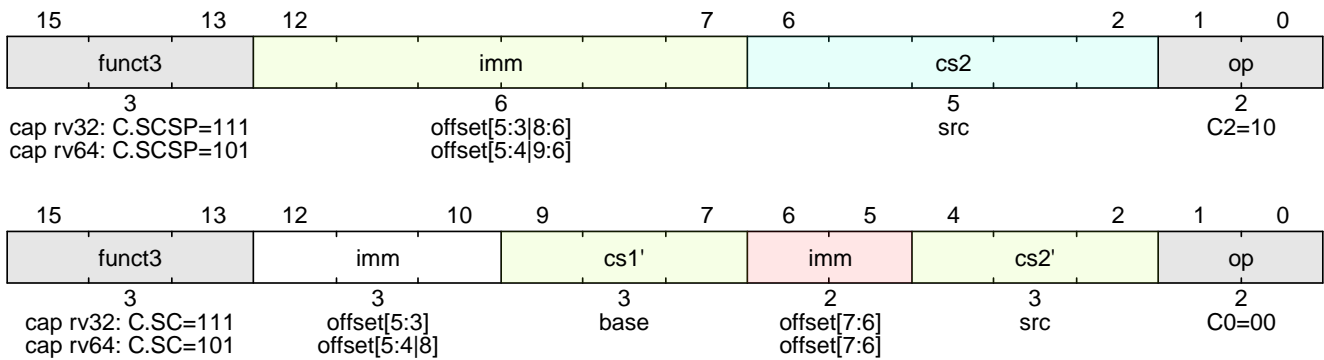
Capability Mode Mnemonics

c.sc cs2', offset(cs1'/csp)

Capability Mode Expansions

sc cs2', offset(cs1'/csp)

Encoding



Capability Mode Description

Store capability instruction, authorised by the capability in **cs1**. Take a store/AMO address misaligned exception if not naturally aligned.

Legacy Mode Description

These mnemonics do not exist in Legacy Mode. The RV32 encodings map to [C.FSW/C.FSWSP](#) and the RV64 encodings map to [C.FSD/C.FSDSP](#).

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites

C or Zca, Zcheri_purecap

Operation (after expansion to 32-bit encodings)

See [SC](#)

7.7. "Zicbom", "Zicbop", "Zicboz" Standard Extensions for Base Cache Management Operations

7.7.1. CBO.CLEAN

Synopsis

Perform a clean operation on a cache block

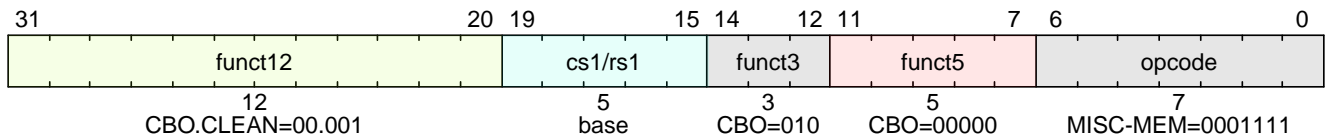
Capability Mode Mnemonic

`cbo.clean 0(cs1)`

Legacy Mode Mnemonic

`cbo.clean 0(rs1)`

Encoding



Capability Mode Description

A CBO.CLEAN instruction performs a clean operation on the cache block whose effective address is the base address specified in **cs1**. The authorising capability for this operation is **cs1**.

Legacy Mode Description

A CBO.CLEAN instruction performs a clean operation on the cache block whose effective address is the base address specified in **rs1**. The authorising capability for this operation is [ddc](#).

Exceptions

CHERI fault exceptions when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	The tag set to 0
Seal violation	It is sealed
Permission violation	It does not grant W-permission and R-permission
Length violation	At least one byte accessed is within the bounds

Prerequisites for Capability Mode

Zicbom, Zcheri_purecap

Prerequisites for Legacy Mode

Zicbom, Zcheri_legacy

Operation

TBD

7.7.2. CBO.FLUSH

Synopsis

Perform a flush operation on a cache block

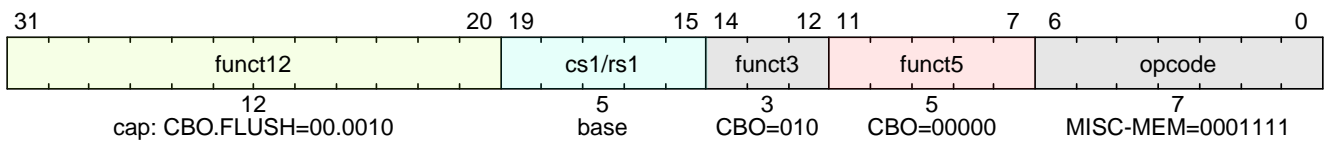
Capability Mode Mnemonic

`cbo.flush 0(cs1)`

Legacy Mode Mnemonic

`cbo.flush 0(rs1)`

Encoding



Capability Mode Description

A CBO.FLUSH instruction performs a flush operation on the cache block whose effective address is the base address specified in `cs1`. The authorising capability for this operation is `cs1`.

Legacy Mode Description

A CBO.FLUSH instruction performs a flush operation on the cache block whose effective address is the base address specified in `rs1`. The authorising capability for this operation is [ddc](#).

Exceptions

CHERI fault exceptions when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	The tag set to 0
Seal violation	It is sealed
Permission violation	It does not grant W-permission and R-permission
Length violation	At least one byte accessed is within the bounds

Prerequisites for Capability Mode

Zicbom, Zcheri_purecap

Prerequisites for Legacy Mode

Zicbom, Zcheri_legacy

Operation

TBD

7.7.3. CBO.INVALID

Synopsis

Perform an invalidate operation on a cache block

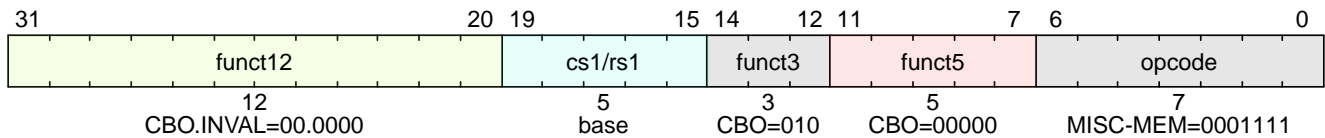
Capability Mode Mnemonic

cbo.inval 0(cs1)

Legacy Mode Mnemonic

cbo.inval 0(rs1)

Encoding



Capability Mode Description

A CBO.INVALID instruction performs an invalidate operation on the cache block whose effective address is the base address specified in **cs1**. The authorising capability for this operation is **cs1**.

Legacy Mode description

A CBO.INVALID instruction performs an invalidate operation on the cache block whose effective address is the base address specified in **rs1**. The authorising capability for this operation is [ddc](#).

Exceptions

CHERI fault exceptions when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the [mtval](#) or [stval](#) TYPE field and the corresponding code is written to CAUSE.

The CBIE bit in [menvcfg](#) and [senvcfg](#) indicates whether CBO.INVALID performs cache block flushes instead of invalidations for less privileged modes. The instruction checks shown in the table below remain unchanged regardless of the value of CBIE and the privilege mode.

CAUSE	Reason
Tag violation	The tag set to 0
Seal violation	It is sealed
Permission violation	It does not grant W-permission , R-permission or ASR-permission
Length violation	At least one byte accessed is outside the bounds

Prerequisites for Capability Mode

Zicbom, Zcheri_purecap

Prerequisites for Legacy Mode

Zicbom, Zcheri_legacy

Operation

TBD

7.7.4. CBO.ZERO

Synopsis

Store zeros to the full set of bytes corresponding to a cache block

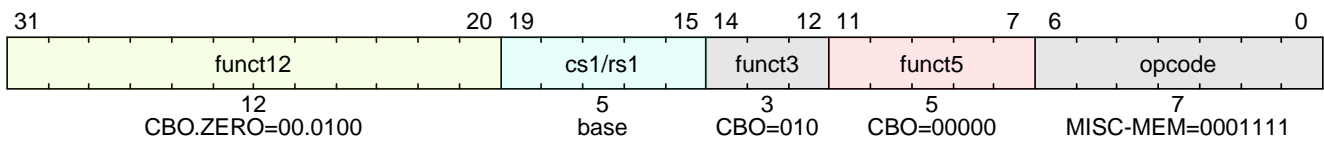
Capability Mode Mnemonic

`cbo.zero 0(cs1)`

Legacy Mode Mnemonic

`cbo.zero 0(rs1)`

Encoding



Capability Mode Description

A `cbo.zero` instruction performs stores of zeros to the full set of bytes corresponding to the cache block whose effective address is the base address specified in `cs1`. An implementation may or may not update the entire set of bytes atomically although each individual write must atomically clear the tag bit of the corresponding aligned CLEN-bit location. The authorising capability for this operation is `cs1`.

Legacy Mode Description

A `cbo.zero` instruction performs stores of zeros to the full set of bytes corresponding to the cache block whose effective address is the base address specified in `cs1`. An implementation may or may not update the entire set of bytes atomically although each individual write must atomically clear the tag bit of the corresponding aligned CLEN-bit location. The authorising capability for this operation is `ddc`.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode

Zicboz, Zcheri_purecap

Prerequisites for Legacy Mode

Zicboz, Zcheri_legacy

Operation

TBD

7.7.5. PREFETCH.I

Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by an instruction fetch in the near future

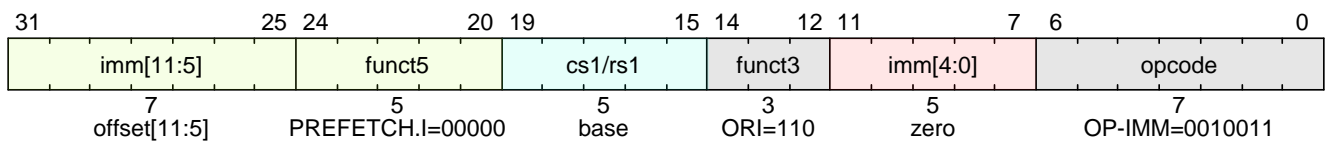
Capability Mode Mnemonic

`prefetch.i offset(cs1)`

Legacy Mode Mnemonic

`prefetch.i offset(rs1)`

Encoding



Capability Mode Description

A PREFETCH.I instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in **cs1** and the sign-extended offset encoded in `imm[11:0]`, where `imm[4:0]` equals `0b000000`, is likely to be accessed by an instruction fetch in the near future. The encoding is only valid if `imm[4:0]=0`. The authorising capability for this operation is **cs1**. This instruction does not throw any exceptions. However, following [CHERI Exceptions and speculative execution](#), this instruction does not perform a prefetch if it is not authorized by **cs1**. This instruction does not perform a memory access if one or more of the following conditions of **cs1** are met:

- The tag is not set
- The sealed bit is set
- No bytes of the cache line requested is in bounds
- The [X-permission](#) is not set

Legacy Mode Description

A PREFETCH.I instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in **rs1** and the sign-extended offset encoded in `imm[11:0]`, where `imm[4:0]` equals `0b000000`, is likely to be accessed by an instruction fetch in the near future. The encoding is only valid if `imm[4:0]=0`. The authorising capability for this operation is [ddc](#).

Prerequisites for Capability Mode

Zicbop, Zcheri_purecap

Prerequisites for Legacy Mode

Zicbop, Zcheri_legacy

Operation

TODO

7.7.6. PREFETCH.R

Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by a data read in the near future

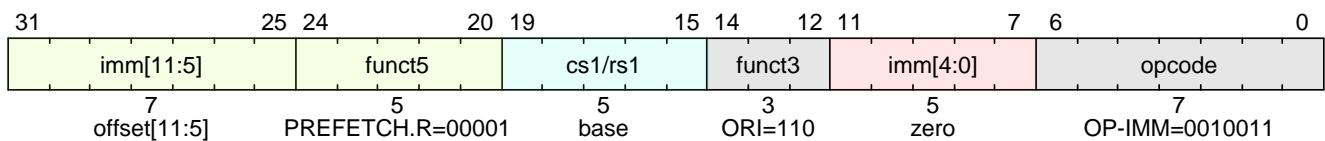
Capability Mode Mnemonic

`prefetch.r offset(cs1)`

Legacy Mode Mnemonic

`prefetch.r offset(rs1)`

Encoding



Capability Mode Description

A PREFETCH.R instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in **cs1** and the sign-extended offset encoded in `imm[11:0]`, where `imm[4:0]` equals `0b000000`, is likely to be accessed by a data read (i.e. load) in the near future. The encoding is only valid if `imm[4:0]=0`. The authorising capability for this operation is **cs1**. This instruction does not throw any exceptions. However, in following [CHERI Exceptions and speculative execution](#), this instruction does not perform a prefetch if it is not authorized by **cs1**. This instruction does not perform a memory access if one or more of the following conditions of **cs1** are met:

- The tag is not set
- The sealed bit is set
- No bytes of the cache line requested is in bounds
- The [R-permission](#) is not set

Legacy Mode Description

A PREFETCH.R instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in **rs1** and the sign-extended offset encoded in `imm[11:0]`, where `imm[4:0]` equals `0b000000`, is likely to be accessed by a data read (i.e. load) in the near future. The encoding is only valid if `imm[4:0]=0`. The authorising capability for this operation is [ddc](#).

Prerequisites for Capability Mode

Zicbop, Zcheri_purecap

Prerequisites for Legacy Mode

Zicbop, Zcheri_legacy

Operation

TODO

7.7.7. PREFETCH.W

Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by a data write in the near future

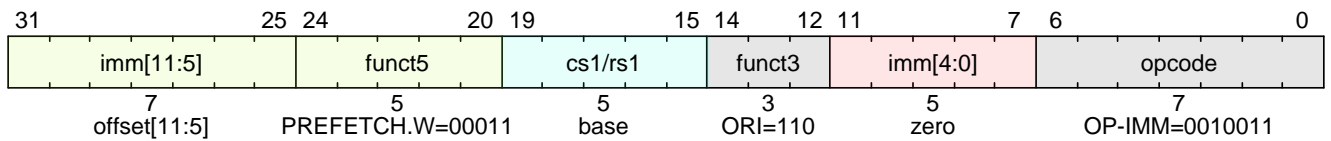
Capability Mode Mnemonic

`prefetch.w offset(cs1)`

Legacy Mode Mnemonic

`prefetch.w offset(rs1)`

Encoding



Capability Mode Description

A PREFETCH.W instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in **cs1** and the sign-extended offset encoded in `imm[11:0]`, where `imm[4:0]` equals `0b000000`, is likely to be accessed by a data write (i.e. store) in the near future. The encoding is only valid if `imm[4:0]=0`. The authorising capability for this operation is **cs1**. This instruction does not throw any exceptions. However, following [CHERI Exceptions and speculative execution](#), this instruction does not perform a prefetch if it is not authorized by **cs1**. This instruction does not perform a memory access if one or more of the following conditions of **cs1** are met:

- The tag is not set
- The sealed bit is set
- No bytes of the cache line requested is in bounds
- The [W-permission](#) is not set

Legacy Mode Description

A PREFETCH.W instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in **rs1** and the sign-extended offset encoded in `imm[11:0]`, where `imm[4:0]` equals `0b000000`, is likely to be accessed by a data write (i.e. store) in the near future. The encoding is only valid if `imm[4:0]=0`. The authorising capability for this operation is [ddc](#).

Prerequisites for Capability Mode

Zicbop, Zcheri_purecap

Prerequisites for Legacy Mode

Zicbop, Zcheri_legacy

Operation

TODO

7.8. "Zba" Extension for Bit Manipulation Instructions

7.8.1. ADD.UW

Synopsis

Add unsigned word for address generation

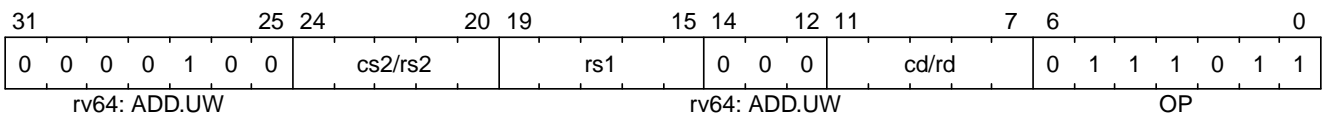
Capability Mode Mnemonic (RV64)

add.uw cd, rs1, cs2

Legacy Mode Mnemonics (RV64)

add.uw rd, rs1, rs2

Encoding



Capability Mode Description

Increment the address field of **cs2** by the unsigned word in **rs1**. Clear the tag if the resulting capability is unrepresentable or **cs2** is sealed.

Legacy Mode Description

Increment **rs2** by the unsigned word in **rs1**.

Prerequisites for Capability Mode

RV64, Zcheri_purecap, Zba

Prerequisites for Legacy Mode

RV64, Zcheri_legacy, Zba

Capability Mode Operation

TBD

Legacy Mode Operation

TODO

7.8.2. SH1ADD

See [SH3ADD](#).

7.8.3. SH2ADD

See [SH3ADD](#).

7.8.4. SH3ADD

Synopsis

Shift by *n* and add for address generation (SH1ADD, SH2ADD, SH3ADD)

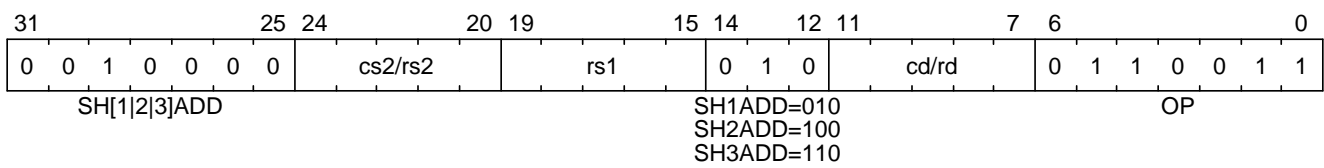
Capability Mode Mnemonics

sh[1|2|3]add cd, rs1, cs2

Legacy Mode Mnemonics

sh[1|2|3]add rd, rs1, rs2

Encoding



Capability Mode Description

Increment the address field of **cs2** by **rs1** shifted left by *n* bit positions. Clear the tag if the resulting capability is unrepresentable or **cs2** is sealed.

Legacy Mode Description

Increment **rs2** by **rs1** shifted left by *n* bit positions.

Prerequisites for Capability Mode

Zcheri_purecap, Zba

Prerequisites for Legacy Mode

Zcheri_legacy, Zba

Capability Mode Operation

TBD

Legacy Mode Operation

TODO

7.8.5. SH1ADD.UW

See [SH3ADD.UW](#).

7.8.6. SH2ADD.UW

See [SH3ADD.UW](#).

7.8.7. SH3ADD.UW

Synopsis

Shift by *n* and add unsigned word for address generation (SH1ADD.UW, SH2ADD.UW, SH3ADD.UW)

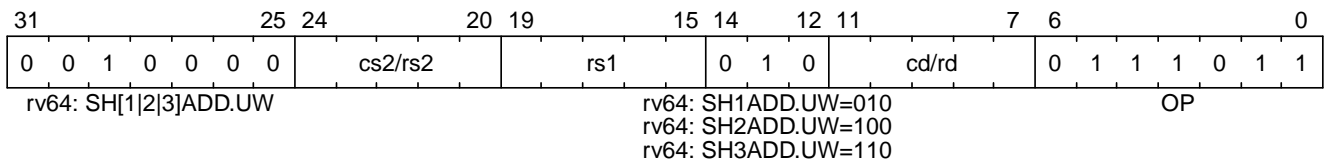
Capability Mode Mnemonic (RV64)

sh[1|2|3]add.uw cd, rs1, cs2

Legacy Mode Mnemonics (RV64)

sh[1|2|3]add.uw rd, rs1, rs2

Encoding



Capability Mode Description

Increment the address field of **cs2** by the unsigned word in **rs1** shifted left by *n* bit positions. Clear the tag if the resulting capability is unrepresentable or **cs2** is sealed.

Legacy Mode Description

Increment **rs2** by the unsigned word in **rs1** shifted left by *n* bit positions.

Prerequisites for Capability Mode

RV64, Zcheri_purecap, Zba

Prerequisites for Legacy Mode

RV64, Zcheri_legacy, Zba

Capability Mode Operation

TBD

Legacy Mode Operation

TODO

7.8.8. SH4ADD



CHERI v9 Note: This instruction is new.

Synopsis

Shift by 4 and add for address generation (SH4ADD)

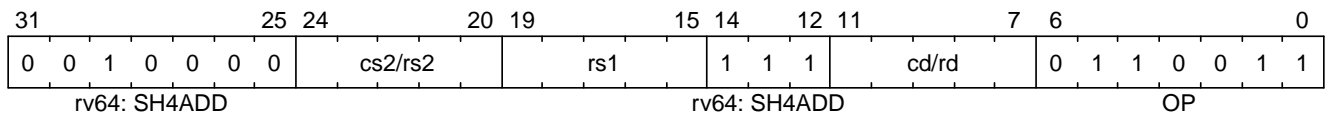
Capability Mode Mnemonics (RV64)

sh4add cd, rs1, cs2

Legacy Mode Mnemonics (RV64)

sh4add rd, rs1, rs2

Encoding



Capability Mode Description

Increment the address field of **cs2** by **rs1** shifted left by 4 bit positions. Clear the tag if the resulting capability is unrepresentable or **cs2** is sealed.

Legacy Mode Description

Increment **rs2** by **rs1** shifted left by 4 bit positions.

Prerequisites for Capability Mode

RV64, Zcheri_purecap, Zba

Prerequisites for Legacy Mode

RV64, Zcheri_legacy, Zba

Capability Mode Operation

TBD

Legacy Mode Operation

TBD

7.8.9. SH4ADD.UW

Synopsis

Shift by 4 and add unsigned words for address generation (SH4ADD.UW)

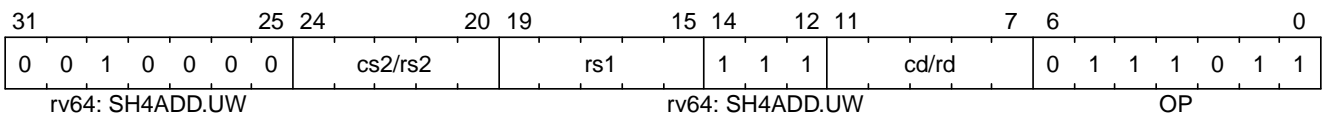
Capability Mode Mnemonics (RV64)

sh4add.uw cd, rs1, cs2

Legacy Mode Mnemonics (RV64)

sh4add.uw rd, rs1, rs2

Encoding



Capability Mode Description

Increment the address field of **cs2** by the unsigned word in **rs1** shifted left by 4 bit positions. Clear the tag if the resulting capability is unrepresentable or **cs2** is sealed.

Legacy Mode Description

Increment **rs2** by the unsigned word in **rs1** shifted left by 4 bit positions.

Prerequisites for Capability Mode

RV64, Zcheri_purecap, Zba

Prerequisites for Legacy Mode

RV64, Zcheri_legacy, Zba

Capability Mode Operation

TBD

Legacy Mode Operation

TBD

7.9. "Zcb" Standard Extension For Code-Size Reduction

7.9.1. C.LH

See [C.LBU](#).

7.9.2. C.LHU

See [C.LBU](#).

7.9.3. C.LBU

Synopsis

Load (C.LH, C.LHU, C.LBU), 16-bit encodings

Capability Mode Mnemonics

`c.lh/c.lhu/c.lbu rd', offset(cs1')`

Capability Mode Expansions

`lh/lhu/lbu rd, offset(cs1)`

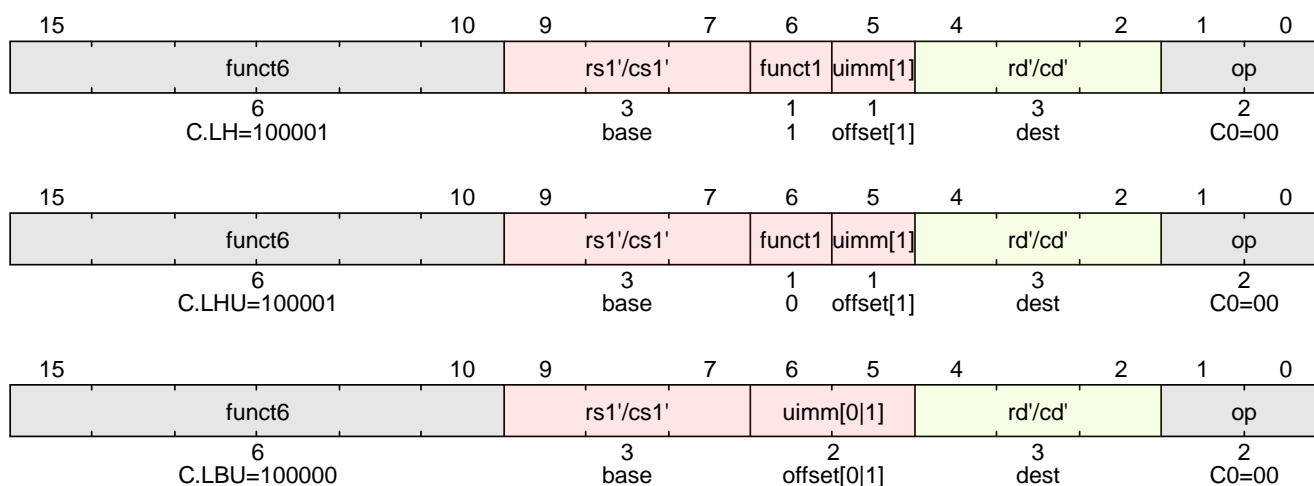
Legacy Mode Mnemonics

`c.lh/c.lhu/c.lbu rd', offset(rs1')`

Legacy Mode Expansions

`lh/lhu/lbu rd, offset(rs1)`

Encoding



Capability Mode Description

Subword load instructions, authorised by the capability in `cs1`.

Legacy Mode Description

Subword load instructions, authorised by the capability in `ddc`.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the `mtval` or `stval` TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap, and Zcb

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, and Zcb

Operation (after expansion to 32-bit encodings)

See [LHU](#), [LH](#), [LBU](#)

7.9.4. C.SH

See [C.SB](#).

7.9.5. C.SB

Synopsis

Stores (C.SH, C.SB), 16-bit encodings

Capability Mode Mnemonics

c.sh/c.sb rs2', offset(cs1')

Capability Mode Expansions

sh/sb rs2', offset(cs1')

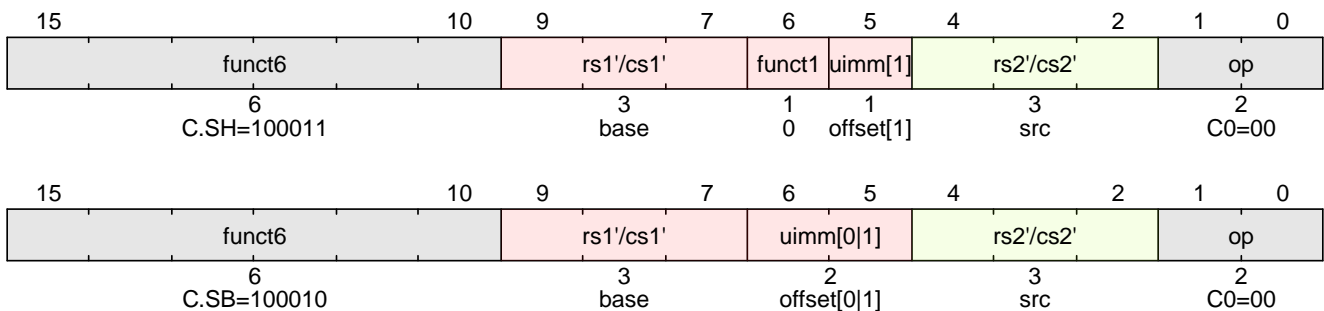
Legacy Mode Mnemonics

c.sh/c.sb rs2', offset(rs1')

Legacy Mode Expansions

sh/sb rs2', offset(rs1')

Encoding



Capability Mode Description

Subword store instructions, authorised by the capability in cs1.

Legacy Mode Description

Subword store instructions, authorised by the capability in ddc.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval or stval TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap, and Zcb

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, and Zcb

Operation (after expansion to 32-bit encodings)

See [SH](#), [SB](#)

7.10. "Zcmp" Standard Extension For Code-Size Reduction

The push ([CM.PUSH](#)) and pop ([CM.POP](#), [CM.POPRET](#), [CM.POPRETZ](#)) instructions are redefined in Capability Mode to save/restore full capabilities.

The double move instructions ([CM.MVSAO1](#), [CM.MVAO1S](#)) are redefined in Capability Mode to move full capabilities between registers. The saved register mapping is as shown in

Table 28. saved register mapping for Zcmp

saved register specifier	xreg	integer ABI	CHERI ABI
0	x8	s0	cs0
1	x9	s1	cs1
2	x18	s2	cs2
3	x19	s3	cs3
4	x20	s4	cs4
5	x21	s5	cs5
6	x22	s6	cs6
7	x23	s7	cs7

All instructions are defined in ([RISC-V, 2023](#)).

7.10.1. CM.PUSH

Synopsis

Create stack frame (CM.PUSH): store the return address register and 0 to 12 saved registers to the stack frame, optionally allocate additional stack space. 16-bit encodings.

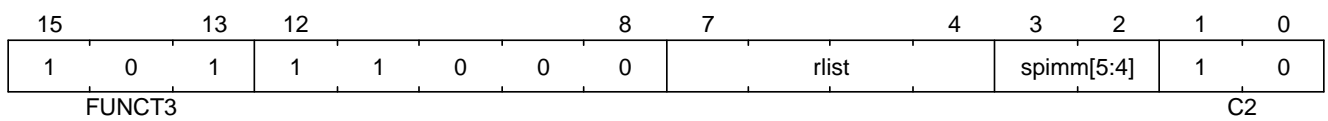
Capability Mode Mnemonic

cm.push {creg_list}, -stack_adj

Legacy Mode Mnemonics

cm.push {reg_list}, -stack_adj

Encoding



rlist values 0 to 3 are reserved for a future EABI variant

Capability Mode Description

Create stack frame, store capability registers as specified in *creg_list*. Optionally allocate additional multiples of 16-byte stack space. All accesses are checked against **csp**.

Legacy Mode Description

Create stack frame, store integer registers as specified in *reg_list*. Optionally allocate additional multiples of 16-byte stack space. All accesses are checked against **ddc**.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the **mtval** or **stval** TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap, Zcmp

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, Zcmp

Operation

TBD

7.10.2. CM.POP

Synopsis

Destroy stack frame (CM.POP): load the return address register and 0 to 12 saved registers from the stack frame, deallocate the stack frame. 16-bit encodings.

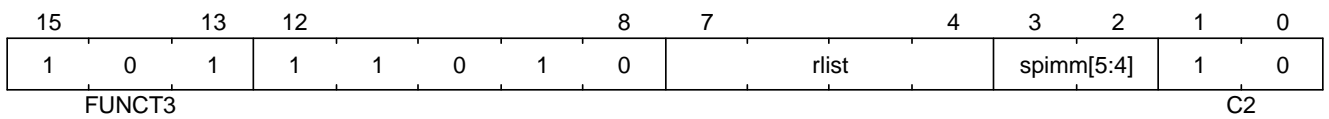
Capability Mode Mnemonic

cm.pop {creg_list}, -stack_adj

Legacy Mode Mnemonics

cm.pop {reg_list}, -stack_adj

Encoding



rlist values 0 to 3 are reserved for a future EABI variant

Capability Mode Description

Load capability registers as specified in *creg_list*. Deallocate stack frame. All accesses are checked against **csp**.

Legacy Mode Description

Load integer registers as specified in *reg_list*. Deallocate stack frame. All accesses are checked against **ddc**.

Exceptions

CHERI fault exception when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the **mtval** or **stval** TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission
Length violation	At least one byte accessed is outside the authority capability bounds

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap, Zcmp

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, Zcmp

Operation

TBD

7.10.3. CM.POPRET

Synopsis

Destroy stack frame (CM.POPRET): load the return address register and 0 to 12 saved registers from the stack frame, deallocate the stack frame. Return through the return address register. 16-bit encodings.

Capability Mode Mnemonic

`cm.popret {creg_list}, -stack_adj`

Legacy Mode Mnemonics

`cm.popret {reg_list}, -stack_adj`

Encoding

15	13	12					8	7			4	3	2	1	0
1	0	1	1	1	1	1	0		rlist			spimm[5:4]		1	0
FUNCT3														C2	



rlist values 0 to 3 are reserved for a future EABI variant

Capability Mode Description

Load capability registers as specified in `creg_list`. Deallocate stack frame. Return by calling [JALR](#) to `cra`. All data accesses are checked against `csp`. The return destination is checked against `cra`.

Legacy Mode Description

Load integer registers as specified in `reg_list`. Deallocate stack frame. Return by calling [JALR](#) to `ra`. All data accesses are checked against `ddc`. The return destination is checked against `pcc`.

Permissions

Loads are checked as for [LC](#) in both Capability and Legacy Modes.

The return is checked as for [JALR](#) in both Capability and Legacy Modes.

Exceptions

When these instructions cause CHERI exceptions, *CHERI data fault* is reported in the TYPE field if a load causes an exception, or *CHERI instruction access fault* if the return causes an exception. The following codes may be reported in the CAUSE field of [mtval](#) or [stval](#):

CAUSE	
Tag violation	✓
Seal violation	✓
Permission violation	✓
Length violation	✓



The instructions on this page are either PC relative or may update the [pcc](#). Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the [pcc](#) in debug mode is UNSPECIFIED by this document.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap, Zcmp

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, Zcmp

Operation

TBD

7.10.4. CM.POPRETZ

Synopsis

Destroy stack frame (CM.POPRETZ): load the return address register and register 0 to 12 saved registers from the stack frame, deallocate the stack frame. Move zero into argument register zero. Return through the return address register. 16-bit encodings.

Capability Mode Mnemonic

`cm.popretz {creg_list}, -stack_adj`

Legacy Mode Mnemonics

`cm.popretz {reg_list}, -stack_adj`

Encoding

15	13	12				8	7			4	3	2	1	0
1	0	1	1	1	1	0	0	rlist			spimm[5:4]		1	0
FUNCT3										C2				



rlist values 0 to 3 are reserved for a future EABI variant

Capability Mode Description

Load capability registers as specified in `creg_list`. Deallocate stack frame. Move zero into `ca0`. Return by calling `JALR` to `cra`. All data accesses are checked against `csp`. The return destination is checked against `cra`.

Legacy Mode Description

Load integer registers as specified in `reg_list`. Deallocate stack frame. Move zero into `a0`. Return by calling `JALR` to `ra`. All data accesses are checked against `ddc`. The return destination is checked against `pcc`.

Permissions

Loads are checked as for `LC` in both Capability and Legacy Modes.

The return is checked as for `JALR` in both Capability and Legacy Modes.

Exceptions

When these instructions cause CHERI exceptions, *CHERI data fault* is reported in the TYPE field if a load causes an exception, or *CHERI instruction access fault* if the return causes an exception. The following codes may be reported in the CAUSE field of `mtval` or `stval`:

CAUSE	
Tag violation	✓
Seal violation	✓
Permission violation	✓
Length violation	✓



The instructions on this page are either PC relative or may update the `pcc`. Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the `pcc` in debug mode is UNSPECIFIED by this document.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap, Zcmp

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, Zcmp

Operation

TBD

7.10.5. CM.MVSA01

Synopsis

CM.MVSA01: Move argument registers 0 and 1 into two saved registers.

Capability Mode Mnemonic

`cm.mvsa01 cr1s', cr2s'`

Legacy Mode Mnemonics

`cm.mvsa01 r1s', r2s'`

Encoding

15	13	12	10	9	7	6	5	4	2	1	0		
1	0	1	0	1	1	r1s'		0	1	r2s'		1	0
FUNCT3											C2		



The encoding uses sreg number specifiers instead of xreg number specifiers to save encoding space. The saved register encoding is shown in [Table 28](#).

Capability Mode Description

Atomically move two saved capability registers **cs0-cs7** into **ca0** and **ca1**.

Legacy Mode Description

Atomically move two saved integer registers **s0-s7** into **a0** and **a1**.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap, Zcmp

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, Zcmp

Operation

TBD

7.10.6. CM.MVA01S

Synopsis

Move two saved registers into argument registers 0 and 1.

Capability Mode Mnemonic

cm.mva01s cr1s', cr2s'

Legacy Mode Mnemonics

cm.mva01s r1s', r2s'

Encoding

15	13	12	10	9	7	6	5	4	2	1	0		
1	0	1	0	1	1	r1s'		1	1	r2s'		1	0
FUNCT3						C2							



The encoding uses sreg number specifiers instead of xreg number specifiers to save encoding space. The saved register encoding is shown in [Table 28](#).

Capability Mode Description

Atomically move two capability registers **ca0** and **ca1** into **cs0-cs7**.

Legacy Mode Description

Atomically move two integer registers **a0** and **a1** into **s0-s7**.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap, Zcmp

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, Zcmp

Operation

TBD

7.11. "Zcmt" Standard Extension For Code-Size Reduction

The table jump instructions ([CM.JT](#), [CM.JALT](#)) defined in ([RISC-V, 2023](#)) are *not* redefined in Capability Mode to have capabilities in the jump table. This is to prevent the code-size growth caused by doubling the size of the jump table.

In the future, new jump table modes or new encodings can be added to have capabilities in the jump table.

The jump vector table CSR [jvt](#) has a capability alias [jvtc](#) so that it can only be configured to point to accessible memory. All accesses to the jump table are checked against [jvtc](#) in Capability Mode, and against [pcc](#) bounds in Legacy Mode. This allows the jump table to be accessed when the [pcc](#) bounds are set narrowly to the local function only in Capability Mode.



Zcmt defines that the fetch from the jump table is from instruction memory. The overall instruction executed is effectively 48-bit, with 16-bits from [CM.JALT/CM.JT](#), the other 32-bits (for RV32) from the table. Therefore [pcc](#) is used to authorise the fetch in Legacy Mode, as the fetch is designated to be from instruction memory in ([RISC-V, 2023](#)).



In Capability Mode the implementation doesn't need to expand and bounds check against [jvtc](#) on every access, it is sufficient to decode the valid accessible range of entries after every write to [jvtc](#), and then check that the accessed entry is in that range.

7.11.1. Jump Vector Table CSR ([jvt](#))

The JVT CSR is exactly as defined by ([RISC-V, 2023](#)). It is renamed to [jvtc](#).

7.11.2. Jump Vector Table CSR ([jvtc](#))

[jvtc](#) extends [jvt](#) to be a capability width CSR, as shown in [Table 16](#).

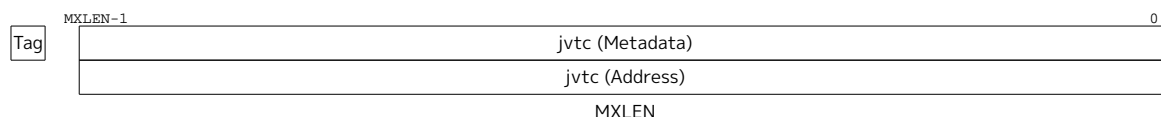


Figure 43. Jump Vector Table Capability register

All instruction fetches from the jump vector table are checked against [jvtc](#) in Capability Mode. In Legacy Mode the address field gives the base address of the table, and the access is checked against [pcc](#) bounds.

See [CM.JALT](#), [CM.JT](#).

If the access to the jump table succeeds, then the instructions execute as follows:

- [CM.JT](#) executes as [J](#) or [AUIPC+JR](#)
- [CM.JALT](#) executes as [JAL](#) or [AUIPC+JALR](#)

As a result the capability metadata is retained in [pcc](#) during execution.

7.11.3. CM.JALT

Synopsis

Jump via table with link (CM.JALT), 16-bit encodings

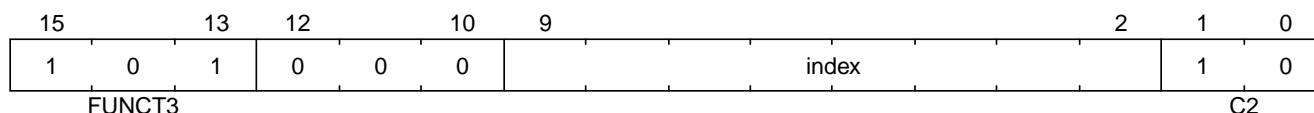
Capability Mode Mnemonic

cm.jalt index

Legacy Mode Mnemonics

cm.jalt index

Encoding



For this encoding to decode as <CM.JALT, index>=32, otherwise it decodes as CM.JT.

Capability Mode Description

Redirect instruction fetch via the jump table defined by the indexing via **jvtc.address+index*XLEN/8**, checking every byte of the jump table access against **jvtc** bounds (not against **pcc**) and requiring **X-permission**. Link to **cra**.

Legacy Mode Description

Redirect instruction fetch via the jump table defined by the indexing via **jvtc.address+index*XLEN/8**, checking every byte of the jump table access against **pcc** bounds and requiring **X-permission**. Link to **ra**.

Capability Mode Permissions

Requires **jvtc** to be tagged, not sealed, have **X-permission** and for the full XLEN-wide access to be in **jvtc** bounds.

Capability Mode Exceptions

When these instructions cause CHERI exceptions, *CHERI instruction access fault* is reported in the TYPE field and the following codes may be reported in the CAUSE field of **mtval** or **stval**:

CAUSE	
Tag violation	✓
Seal violation	✓
Permission violation	✓
Length violation	✓



The instructions on this page are either PC relative or may update the **pcc**. Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the **pcc** in debug mode is UNSPECIFIED by this document.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap, Zcmt

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, Zcmt

Operation

TBD

7.11.4. CM.JT

Synopsis

Jump via table with link (CM.JT), 16-bit encodings

Capability Mode Mnemonic

cm.jt index

Legacy Mode Mnemonics

cm.jt index

Encoding

15	13	12	10	9		2	1	0
1	0	1	0	0	0	index	1	0
FUNCT3							C2	



For this encoding to decode as [CM.JT](#), $\text{index} < 32$, otherwise it decodes as [CM.JALT](#).

Capability Mode Description

Redirect instruction fetch via the jump table defined by the indexing via **jvtc.address+index*XLEN/8**, checking every byte of the jump table access against **jvtc** bounds (not against **pcc**) and requiring [X-permission](#).

Legacy Mode Description

Redirect instruction fetch via the jump table defined by the indexing via **jvtc.address+index*XLEN/8**, checking every byte of the jump table access against **pcc** bounds and requiring [X-permission](#).

Capability Mode Permissions

Requires **jvtc** to be tagged, not sealed, have [X-permission](#) and for the full XLEN-wide access to be in **jvtc** bounds.

Capability Mode Exceptions

When these instructions cause CHERI exceptions, *CHERI instruction access fault* is reported in the TYPE field and the following codes may be reported in the CAUSE field of **mtval** or **stval**:

CAUSE	
Tag violation	✓
Seal violation	✓
Permission violation	✓
Length violation	✓



The instructions on this page are either PC relative or may update the **pcc**. Therefore an implementation may make them illegal in debug mode. If they are supported then the value of the **pcc** in debug mode is *UNSPECIFIED* by this document.

Prerequisites for Capability Mode

C or Zca, Zcheri_purecap, Zcmt

Prerequisites for Legacy Mode

C or Zca, Zcheri_legacy, Zcmt

Operation

TBD

Chapter 8. Extension summary

8.1. Zabhlrsc

Zabhlrsc is a separate extension independent of CHERI, but is required for CHERI software.

These instructions are not controlled by the CRE bits in [mseccfg](#), [menvcfg](#) or [senvcfg](#).

Table 29. Zabhlrsc instruction extension

Mnemonic	Zabhlrsc	Function
LR.H	✓	Load reserved half
LR.B	✓	Load reserved byte
SC.H	✓	Store conditional half
SC.B	✓	Store conditional byte

8.2. Zcheri_purecap

Zcheri_purecap defines the set of instructions used by a purecap core.

Some instructions depend on the presence of other extensions, as listed in [Table 30](#)

Table 30. Zcheri_purecap instruction extension - Pure Capability Mode instructions

Mnemonic	RV 32	RV 64	A	Za bhl rsc	Zic bo[mp z]	C or Zca	Zb a	Zc b	Zc mp	Zc mt	Zfh	F	D	V	Function
LC	✓	✓													Load cap via capability register
SC	✓	✓													Store cap via capability register
C.LCSP	✓	✓				✓									Load cap via cap, SP relative
C.SCSP	✓	✓				✓									Store cap via cap, SP relative
C.LC	✓	✓				✓									Load cap via cap
C.SC	✓	✓				✓									Store cap via cap
C.LWSP	✓	✓				✓									Load word via cap, SP relative
C.SWSP	✓	✓				✓									Store word via cap, SP relative
C.LW	✓	✓				✓									Load word via cap
C.SW	✓	✓				✓									Store word via cap
C.LD		✓				✓									Load word via cap

Mnemonic	RV 32	RV 64	A	Za bhl rsc	Zic bo[mp z]	C or Zca	Zb a	Zc b	Zc mp	Zc mt	Zfh	F	D	V	Function
C.SD		✓				✓									Store word via cap
C.LDSP		✓				✓									Load word via cap
C.SDSP		✓				✓									Store word via cap
LB	✓	✓													Load signed byte
LH	✓	✓													Load signed half
C.LH	✓	✓						✓							Load signed half
LW	✓	✓													Load signed word
LBU	✓	✓													Load unsigned byte
C.LBU	✓	✓						✓							Load unsigned byte
LHU	✓	✓													Load unsigned half
C.LHU	✓	✓						✓							Load unsigned half
LWU		✓													Load unsigned word
LD		✓													Load double
SB	✓	✓													Store byte
C.SB	✓	✓						✓							Store byte
SH	✓	✓													Store half
C.SH	✓	✓						✓							Store half
SW	✓	✓													Store word
SD		✓													Store double
AUIPC	✓	✓													Add immediate to PCC address
CADD	✓	✓													Increment cap address by register, representability check
CADDI	✓	✓													Increment cap address by immediate, representability check

Mnemonic	RV 32	RV 64	A	Za bhl rsc	Zic bo[mp z]	C or Zca	Zb a	Zc b	Zc mp	Zc mt	Zfh	F	D	V	Function
SCADDR	✓	✓													Replace capability address, representability check
GCTAG	✓	✓													Get tag field
GCPERM	✓	✓													Get hperm and uperm fields as 1-bit per permission, packed together
CMV	✓	✓													Move capability register
ACPERM	✓	✓													AND capability permissions (expand to 1-bit per permission before ANDing)
GCHI	✓	✓													Get metadata
SCHI	✓	✓													Set metadata and clear tag
SCEQ	✓	✓													Full capability bitwise compare, set result true if both are fully equal
SENTRY	✓	✓													Seal capability
SCSS	✓	✓													Set result true if cs1 and cs1 tags match and cs2 bounds and permissions are a subset of cs1
CBLD	✓	✓													Set cd to cs2 with its tag set after checking that cs2 is a subset of cs1
SCBNDS	✓	✓													Set register bounds on capability with rounding, clear tag if rounding is required

Mnemonic	RV 32	RV 64	A	Za bhl rsc	Zic bo[mp z]	C or Zca	Zb a	Zc b	Zc mp	Zc mt	Zfh	F	D	V	Function
SCBNDSI	✓	✓													Set immediate bounds on capability with rounding, clear tag if rounding is required
SCBNDSR	✓	✓													Set bounds on capability with rounding up as required
GRAM	✓	✓													Representable Alignment Mask: Return mask to apply to address to get the requested bounds
GCBASE	✓	✓													Get capability base
GCLen	✓	✓													Get capability length
C.ADDI16SP	✓	✓				✓									ADD immediate to stack pointer, CADD in Capability Mode
C.ADDI4SPN	✓	✓				✓									ADD immediate to stack pointer, CADDI in Capability Mode
C.MV	✓	✓				✓									Register Move, cap reg move in Capability Mode
C.J	✓	✓				✓									Jump to PC+offset, bounds check minimum size target instruction
C.JAL	✓					✓									Jump to PC+offset, bounds check minimum size target instruction, link to cd

Mnemonic	RV 32	RV 64	A	Za bhl rsc	Zic bo[mp z]	C or Zca	Zb a	Zc b	Zc mp	Zc mt	Zfh	F	D	V	Function
JAL	✓	✓				✓									Jump to PC+offset, bounds check minimum size target instruction, link to cd
JALR	✓	✓													Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap
C.JALR	✓	✓				✓									Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap
C.JR	✓	✓				✓									Indirect cap jump, bounds check minimum size target instruction, unseal target cap
CBO.INVALID	✓	✓			✓										Cache block invalidate (implemented as clean)
CBO.CLEAN	✓	✓			✓										Cache block clean
CBO.FLUSH	✓	✓			✓										Cache block flush
CBO.ZERO	✓	✓			✓										Cache block zero
PREFETCH. R	✓	✓			✓										Prefetch instruction cache line, always valid
PREFETCH. W	✓	✓			✓										Prefetch read-only data cache line
PREFETCH.I	✓	✓			✓										Prefetch writeable data cache line
LR.C	✓	✓	✓												Load reserved capability

Mnemonic	RV 32	RV 64	A	Za bhl rsc	Zic bo[mp z]	C or Zca	Zb a	Zc b	Zc mp	Zc mt	Zfh	F	D	V	Function
LR.D			✓												Load reserved double
LR.W			✓												Load reserved word
LR.H	✓	✓		✓											Load reserved half
LR.B	✓	✓		✓											Load reserved byte
SC.C	✓	✓	✓												Store conditional capability
SC.D			✓												Store conditional double
SC.W			✓												Store conditional word
SC.H	✓	✓		✓											Store conditional half
SC.B	✓	✓		✓											Store conditional byte
AMOSWAP.C	✓	✓	✓												Atomic swap of cap
AMO<OP>.W	✓	✓	✓												Atomic op of word
AMO<OP>.D		✓	✓												Atomic op of double
C.FLD	✓												✓		Load floating point double
C.FLDSP	✓												✓		Load floating point double, sp relative
C.FSD	✓												✓		Store floating point double
C.FSDSP	✓												✓		Store floating point double, sp relative
FLH	✓	✓									✓				Load floating point half via cap
FSH	✓	✓									✓				Store floating point half via cap
FLW	✓	✓										✓			Load floating point word via cap

Mnemonic	RV 32	RV 64	A	Za bhl rsc	Zic bo[mp z]	C or Zca	Zb a	Zc b	Zc mp	Zc mt	Zfh	F	D	V	Function
FSW	✓	✓										✓			Store floating point word via cap
FLD	✓	✓											✓		Load floating point double via cap
FSD	✓	✓											✓		Store floating point double via cap
CM.PUSH	✓	✓							✓						Push integer stack frame
CM.POP	✓	✓							✓						Pop integer stack frame
CM.POPRET	✓	✓							✓						Pop integer stack frame and return
CM.POPRET Z	✓	✓							✓						Pop integer stack frame and return zero
CM.MVSA01	✓	✓							✓						Move two integer registers
CM.MVA01S	✓	✓							✓						Move two integer registers
CM.JALT	✓	✓								✓					Table jump and link
CM.JT	✓	✓								✓					Table jump
ADD.UW		✓					✓								add unsigned words, representability check in cap mode
SH1ADD	✓	✓					✓								shift and add, representability check in cap mode
SH1ADD.UW		✓					✓								shift and add unsigned words, representability check in cap mode

Mnemonic	RV 32	RV 64	A	Za bhl rsc	Zic bo[mp z]	C or Zca	Zb a	Zc b	Zc mp	Zc mt	Zfh	F	D	V	Function
SH2ADD	✓	✓					✓								shift and add, representability check in cap mode
SH2ADD.UW		✓					✓								shift and add unsigned words, representability check in cap mode
SH3ADD	✓	✓					✓								shift and add, representability check in cap mode
SH3ADD.UW		✓					✓								shift and add unsigned words, representability check in cap mode
SH4ADD		✓													shift and add, representability check in cap mode
SH4ADD.UW		✓													shift and add unsigned words, representability check in cap mode

8.3. Zcheri_legacy

Zcheri_legacy defines the set of instructions added by the Legacy Mode, in addition to Zcheri_purecap.



Zcheri_legacy implies Zcheri_purecap

Table 31. Zcheri_legacy instruction extension - Legacy Mode instructions

Mnemonic	RV 32	RV 64	A	Za bhl rsc	Zic bo[mp z]	C or Zca	Zb a	Zc b	Zc mp	Zc mt	Zfh	F	D	V	Function
SCMODE	✓	✓													Set the mode bit of a capability, no permissions required
MODESW	✓	✓													Directly switch mode (legacy / capability)
C.MODESW	✓	✓													Directly switch mode (legacy / capability)
C.FLW	✓											✓			Load floating point word via cap
C.FLWSP	✓											✓			Load floating point word, sp relative
C.FSW	✓											✓			Store floating point word via cap
C.FSWSP	✓											✓			Store floating point word, sp relative
C.FLD		✓											✓		Load floating point double
C.FLDSP		✓											✓		Load floating point double, sp relative
C.FSD		✓											✓		Store floating point double
C.FSDSP		✓											✓		Store floating point double, sp relative

8.4. Instruction Modes

The tables summarise which operating modes each instruction may be executed in.

Table 32. Instructions valid for execution in Capability Mode only

Mnemonic	Zcheri_leg acy	Zcheri_pu recap	Function
C.LCSP		✓	Load cap via cap, SP relative
C.SCSP		✓	Store cap via cap, SP relative

Mnemonic	Zcheri_leg acy	Zcheri_pu recap	Function
C.LC		✓	Load cap via cap
C.SC		✓	Store cap via cap

Table 33. Instructions valid for execution in Legacy Mode only

Mnemonic	Zcheri_legacy	Zcheri_pu recap	Function
C.FLW	✓		Load floating point word via cap
C.FLWSP	✓		Load floating point word, sp relative
C.FSW	✓		Store floating point word via cap
C.FSWSP	✓		Store floating point word, sp relative
C.FLD	✓		Load floating point double
C.FLDSP	✓		Load floating point double, sp relative
C.FSD	✓		Store floating point double
C.FSDSP	✓		Store floating point double, sp relative

Table 34. Instructions valid for execution in both capability and Legacy Modes

Mnemonic	Zcheri_legacy	Zcheri_pu recap	Function
LC	✓	✓	Load cap via int pointer
SC	✓	✓	Store cap via int pointer
C.LWSP	✓	✓	Load word via cap, SP relative
C.SWSP	✓	✓	Store word via cap, SP relative
C.LW	✓	✓	Load word via cap
C.SW	✓	✓	Store word via cap
C.LD	✓	✓	Load word via cap
C.SD	✓	✓	Store word via cap
C.LDSP	✓	✓	Load word via cap
C.SDSP	✓	✓	Store word via cap
LB	✓	✓	Load signed byte
LH	✓	✓	Load signed half
C.LH	✓	✓	Load signed half
LW	✓	✓	Load signed word
LBU	✓	✓	Load unsigned byte
C.LBU	✓	✓	Load unsigned byte
LHU	✓	✓	Load unsigned half
C.LHU	✓	✓	Load unsigned half
LWU	✓	✓	Load unsigned word
LD	✓	✓	Load double
SB	✓	✓	Store byte

Mnemonic	Zcheri_leg acy	Zcheri_pu recap	Function
C.SB	✓	✓	Store byte
SH	✓	✓	Store half
C.SH	✓	✓	Store half
SW	✓	✓	Store word
SD	✓	✓	Store double
AUIPC	✓	✓	Add immediate to PCC address
CADD	✓	✓	Increment cap address by register, representability check
CADDI	✓	✓	Increment cap address by immediate, representability check
SCADDR	✓	✓	Replace capability address, representability check
GCTAG	✓	✓	Get tag field
GCPERM	✓	✓	Get hperm and uperm fields as 1-bit per permission, packed together
CMV	✓	✓	Move capability register
ACPERM	✓	✓	AND capability permissions (expand to 1-bit per permission before ANDing)
GCHI	✓	✓	Get metadata
SCHI	✓	✓	Set metadata and clear tag
SCEQ	✓	✓	Full capability bitwise compare, set result true if both are fully equal
SENTRY	✓	✓	Seal capability
SCSS	✓	✓	Set result true if cs1 and cs1 tags match and cs2 bounds and permissions are a subset of cs1
CBLD	✓	✓	Set cd to cs2 with its tag set after checking that cs2 is a subset of cs1
SCBNDS	✓	✓	Set register bounds on capability with rounding, clear tag if rounding is required
SCBNDSI	✓	✓	Set immediate bounds on capability with rounding, clear tag if rounding is required
SCBNDSR	✓	✓	Set bounds on capability with rounding up as required

Mnemonic	Zcheri_legacy	Zcheri_purify	Function
GRAM	✓	✓	Representable Alignment Mask: Return mask to apply to address to get the requested bounds
GCBASE	✓	✓	Get capability base
GCLLEN	✓	✓	Get capability length
SCMODE	✓		Set the mode bit of a capability, no permissions required
MODESW	✓		Directly switch mode (legacy / capability)
C.MODESW	✓		Directly switch mode (legacy / capability)
C.ADDI16SP	✓	✓	ADD immediate to stack pointer, CADD in Capability Mode
C.ADDI4SPN	✓	✓	ADD immediate to stack pointer, CADDI in Capability Mode
C.MV	✓	✓	Register Move, cap reg move in Capability Mode
C.J	✓	✓	Jump to PC+offset, bounds check minimum size target instruction
C.JAL	✓	✓	Jump to PC+offset, bounds check minimum size target instruction, link to cd
JAL	✓	✓	Jump to PC+offset, bounds check minimum size target instruction, link to cd
JALR	✓	✓	Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap
C.JALR	✓	✓	Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap
C.JR	✓	✓	Indirect cap jump, bounds check minimum size target instruction, unseal target cap
DRET			Return from debug mode, sets ddc from dddc and pcc from dpcc
MRET			Return from machine mode handler, sets PCC from MTVECC, needs ASR permission

Mnemonic	Zcheri_leg acy	Zcheri_pu recap	Function
SRET			Return from supervisor mode handler, sets PCC from STVECC, needs ASR permission
CSRRW			CSR write - can also read/write a full capability through an address alias
CSRRS			CSR set - can also read/write a full capability through an address alias
CSRRC			CSR clear - can also read/write a full capability through an address alias
CSRRWI			CSR write - can also read/write a full capability through an address alias
CSRRSI			CSR set - can also read/write a full capability through an address alias
CSRRCI			CSR clear - can also read/write a full capability through an address alias
CBO.INVALID	✓	✓	Cache block invalidate (implemented as clean)
CBO.CLEAN	✓	✓	Cache block clean
CBO.FLUSH	✓	✓	Cache block flush
CBO.ZERO	✓	✓	Cache block zero
PREFETCH.R	✓	✓	Prefetch instruction cache line, always valid
PREFETCH.W	✓	✓	Prefetch read-only data cache line
PREFETCH.I	✓	✓	Prefetch writeable data cache line
LR.C	✓	✓	Load reserved capability
LR.D	✓	✓	Load reserved double
LR.W	✓	✓	Load reserved word
LR.H	✓	✓	Load reserved half
LR.B	✓	✓	Load reserved byte
SC.C	✓	✓	Store conditional capability
SC.D	✓	✓	Store conditional double
SC.W	✓	✓	Store conditional word
SC.H	✓	✓	Store conditional half
SC.B	✓	✓	Store conditional byte
AMOSWAP.C	✓	✓	Atomic swap of cap
AMO<OP>.W	✓	✓	Atomic op of word
AMO<OP>.D	✓	✓	Atomic op of double

Mnemonic	Zcheri_leg acy	Zcheri_pu recap	Function
C.FLD	✓	✓	Load floating point double
C.FLDSP	✓	✓	Load floating point double, sp relative
C.FSD	✓	✓	Store floating point double
C.FSDSP	✓	✓	Store floating point double, sp relative
FLH	✓	✓	Load floating point half via cap
FSH	✓	✓	Store floating point half via cap
FLW	✓	✓	Load floating point word via cap
FSW	✓	✓	Store floating point word via cap
FLD	✓	✓	Load floating point double via cap
FSD	✓	✓	Store floating point double via cap
CM.PUSH	✓	✓	Push integer stack frame
CM.POP	✓	✓	Pop integer stack frame
CM.POPRET	✓	✓	Pop integer stack frame and return
CM.POPRETZ	✓	✓	Pop integer stack frame and return zero
CM.MVSAO1	✓	✓	Move two integer registers
CM.MVAO1S	✓	✓	Move two integer registers
CM.JALT	✓	✓	Table jump and link
CM.JT	✓	✓	Table jump
ADD.UW	✓	✓	add unsigned words, representability check in cap mode
SH1ADD	✓	✓	shift and add, representability check in cap mode
SH1ADD.UW	✓	✓	shift and add unsigned words, representability check in cap mode
SH2ADD	✓	✓	shift and add, representability check in cap mode
SH2ADD.UW	✓	✓	shift and add unsigned words, representability check in cap mode
SH3ADD	✓	✓	shift and add, representability check in cap mode
SH3ADD.UW	✓	✓	shift and add unsigned words, representability check in cap mode
SH4ADD	✓	✓	shift and add, representability check in cap mode

Mnemonic	Zcheri_leg acy	Zcheri_pu recap	Function
SH4ADD.UW	✓	✓	shift and add unsigned words, representability check in cap mode

Chapter 9. Capability Width CSR Summary

Table 35. CSRs renamed and extended to capability width

CLEN CSR	Alias	Prerequisites
dpcc	dpc	Sdext
dscratch0c	dscratch0	Sdext
dscratch1c	dscratch1	Sdext
mtvecc	mtvec	M-mode
mscratchc	mscratch	M-mode
mepcc	mepc	M-mode
stvecc	stvec	S-mode
sscratchc	sscratch	S-mode
sepc	sepc	S-mode
jvte	jvt	Zcmt

Table 36. Action taken on writing to extended CSRs.

CLEN CSR	Action on XLEN write	Action on CLEN write
dpcc	Apply Invalid address conversion . Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change
dscratch0c	Update the CSR using SCADDR .	direct write
dscratch1c	Update the CSR using SCADDR .	direct write
mtvecc	Apply Invalid address conversion . Always update the CSR with SCADDR even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.	Apply Invalid address conversion . Always update the CSR with SCADDR even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.
mscratchc	Update the CSR using SCADDR .	direct write
mepcc	Apply Invalid address conversion . Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change

CLEN CSR	Action on XLEN write	Action on CLEN write
stvecc	Apply Invalid address conversion . Always update the CSR with SCADDR even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.	Apply Invalid address conversion . Always update the CSR with SCADDR even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.
sscratchc	Update the CSR using SCADDR .	direct write
sepcc	Apply Invalid address conversion . Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change
jvte	Apply Invalid address conversion . Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change

* The vector range check is to ensure that vectored entry to the handler is within bounds of the capability written to [Xtvecc](#). The check on writing must include the lowest (0 offset) and highest possible offset (e.g. $64 * MXLEN$ bits where $HICAUSE=16$).



XLEN writing is only available if Zcheri_legacy is implemented.



Implementations which allow [misa.C](#) to be writable need to legalise [Xepcc](#) on reading if the [misa.C](#) value has changed since the value was written as this can cause the read value of bit [1] to change state.



[CSRRW](#) make an XLEN-wide access to the XLEN-wide CSR aliases or a CLEN-wide access to the CLEN-wide aliases for all extended CSRs. [CSRRWI](#), [CSRRS](#), [CSRRSI](#), [CSRRC](#) and [CSRRCI](#) only make XLEN-wide accesses even if the CLEN-wide alias is specified.

Table 37. CLEN-wide CSRs storing executable vectors or data pointers

CLEN CSR	Executable Vector	Data Pointer	Unseal On Execution
dpcc	✓		✓
mtvecc	✓		
mepcc	✓		✓
stvecc	✓		
sepcc	✓		✓
jvte	✓		
dddc		✓	
ddc		✓	

Some CSRs store executable vectors as shown in [Table 37](#). These CSRs do not need to store the full width address on RV64. If they store fewer address bits then writes are subject to the invalid address check in [Invalid address conversion](#).

Table 38. CLEN-wide CSRs which store all CLEN+1 bits

CLEN CSR	Store full metadata
dscratch0c	✓
dscratch1c	✓
mscratchc	✓
sscratchc	✓
dinfcc	✓

Table 38 shows which CLEN-wide CSRs store all CLEN+1 bits. No other CLEN-wide CSRs store any reserved bits. All CLEN-wide CSRs store *all* non-reserved metadata fields.

Table 39. All CLEN-wide CSRs

CLEN CSR	Zcher i_legacy	Zcher i_pur ecap	Prere quisit es	Addr ess	Permissions	Reset Value	Description
dpcc	✓	✓	Sdext	0x7b1	DRW	tag=0, otherwise undefined	Debug Program Counter Capability
dscratch0c	✓	✓	Sdext	0x7b2	DRW	tag=0, otherwise undefined	Debug Scratch Capability 0
dscratch1c	✓	✓	Sdext	0x7b3	DRW	tag=0, otherwise undefined	Debug Scratch Capability 1
mtvecc	✓	✓	M- mode	0x305	MRW, ASR- permission	Infinite	Machine Trap-Vector Base- Address Capability
mscratchc	✓	✓	M- mode	0x340	MRW, ASR- permission	tag=0, otherwise undefined	Machine Scratch Capability
mepcc	✓	✓	M- mode	0x341	MRW, ASR- permission	Infinite	Machine Exception Program Counter Capability
stvecc	✓	✓	S- mode	0x105	SRW, ASR- permission	Infinite	Supervisor Trap-Vector Base-Address Capability
sscratchc	✓	✓	S- mode	0x140	SRW, ASR- permission	tag=0, otherwise undefined	Supervisor Scratch Capability
sepcc	✓	✓	S- mode	0x141	SRW, ASR- permission	Infinite	Supervisor Exception Program Counter Capability
jvtc	✓	✓	Zcmt	0x017	URW	tag=0, otherwise undefined	Jump Vector Table Capability

CLEN CSR	Zcher i_legacy	Zcher i_pur ecap	Prere quisit es	Addr ess	Permissions	Reset Value	Description
dddc	✓		Sdext	0x7bc	DRW	tag=0, otherwise undefined	Debug Default Data Capability (saved/restored on debug mode entry/exit)
mtdc	✓		M- mode	0x74c	MRW, ASR- permission	tag=0, otherwise undefined	Machine Trap Data Capability (scratch register)
stdc	✓		S- mode	0x163	SRW, ASR- permission	tag=0, otherwise undefined	Supervisor Trap Data Capability (scratch register)
ddc	✓		none	0x416	URW	Infinite	User Default Data Capability
dinf	✓	✓	Sdext	0x7bd	DRW	Infinite	Source of Infinite capability in debug mode, writes are ignored

9.1. Other tables

Table 40. Mnemonics with the same encoding but mapped to different instructions in Legacy and Capability Mode

Mnemonic	Legacy mnemonic RV32	Legacy mnemonic RV64
C.LCSP	C.FLWSP	C.FLDSP
C.SCSP	C.FSWSP	C.FSDSP
C.LC	C.FLW	C.FLD
C.SC	C.FSW	C.FSD

Table 41. Instruction encodings which vary depending on the current XLEN

Mnemonic	Function
LC	Load cap via int pointer
SC	Store cap via int pointer
C.LCSP	Load cap via cap, SP relative
C.SCSP	Store cap via cap, SP relative
C.LC	Load cap via cap
C.SC	Store cap via cap
LR.C	Load reserved capability
SC.C	Store conditional capability
AMOSWAP.C	Atomic swap of cap



[MODESW](#) and [SCMODE](#) only exist in Capability Mode if Legacy Mode is also present. A purecap core does not implement the mode bit in the capability.

Table 42. Illegal instruction detect for CHERI instructions

Mnemonic	illegal insn if (1)	OR illegal insn if (2)	OR illegal insn if (3)
MODESW	mode==D (optional)		
C.MODESW	mode==D (optional)		
C.J	mode==D (optional)		
C.JAL	mode==D (optional)		
JAL	mode==D (optional)		
JALR	mode==D (optional)		
C.JALR	mode==D (optional)		
C.JR	mode==D (optional)		
DRET	MODE<D		
MRET	MODE<M	PCC.ASR==0	
SRET	MODE<S	PCC.ASR==0	mstatus.TSR==1 AND MODE==S
CSRRW	CSR permission fault		
CSRRS	CSR permission fault		
CSRRC	CSR permission fault		
CSRRWI	CSR permission fault		
CSRRSI	CSR permission fault		
CSRRCI	CSR permission fault		
CBO.INVALID	MODE<M AND menvcfg.CBIE[0]==0	MODE<S AND senvcfg.CBIE[0]==0	
CBO.CLEAN	MODE<M AND menvcfg.CBIE[0]==0	MODE<S AND senvcfg.CBIE[0]==0	
CBO.FLUSH	MODE<M AND menvcfg.CBIE[0]==0	MODE<S AND senvcfg.CBIE[0]==0	
CBO.ZERO	MODE<M AND menvcfg.CBIE[0]==0	MODE<S AND senvcfg.CBIE[0]==0	
C.FLW	Xstatus.fs==0		
C.FLWSP	Xstatus.fs==0		
C.FSW	Xstatus.fs==0		
C.FSWSP	Xstatus.fs==0		
C.FLD	Xstatus.fs==0		
C.FLDSP	Xstatus.fs==0		
C.FLD	Xstatus.fs==0		
C.FLDSP	Xstatus.fs==0		
C.FSD	Xstatus.fs==0		
C.FSDSP	Xstatus.fs==0		

Mnemonic	illegal insn if (1)	OR illegal insn if (2)	OR illegal insn if (3)
C.FSD	Xstatus.fs==0		
C.FSDSP	Xstatus.fs==0		
FLH	Xstatus.fs==0		
FSH	Xstatus.fs==0		
FLW	Xstatus.fs==0		
FSW	Xstatus.fs==0		
FLD	Xstatus.fs==0		
FSD	Xstatus.fs==0		

Bibliography

RISC-V. (2022). *RISC-V Debug Specification*. github.com/riscv/riscv-debug-spec/raw/c93823ef349286dc71a00928bddb7254e46bc3b5/riscv-debug-stable.pdf

RISC-V. (2023). *RISC-V Privileged Specification*. github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-056b6ff-2023-10-02/priv-isa-asciidoc.pdf

RISC-V. (2023). *RISC-V Unprivileged Specification*. github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-056b6ff-2023-10-02/unpriv-isa-asciidoc.pdf

RISC-V. (2023). *RISC-V Code-size Reduction Specification*. github.com/riscv/riscv-code-size-reduction/releases/download/v1.0.4-3/Zc-1.0.4-3.pdf

Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Barnes, G., Chisnall, D., Clarke, J., Davis, B., Eisen, L., Filardo, N. W., Fuchs, F. A., Grisenthwaite, R., Joannou, A., Laurie, B., Markettos, A. T., Moore, S. W., ... Xia, H. (2023). *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)* (UCAM-CL-TR-987; Issue UCAM-CL-TR-987). University of Cambridge, Computer Laboratory. doi.org/10.48456/tr-987

Woodruff, J., Joannou, A., Xia, H., Fox, A., Norton, R. M., Chisnall, D., Davis, B., Gudka, K., Filardo, N. W., Markettos, A. T., & others. (2019). Cheri concentrate: Practical compressed capabilities. *IEEE Transactions on Computers*, 68(10), 1455–1469. doi.org/10.1109/TC.2019.2914037