# 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 Markettos, 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 v0.0.1-prerelease, 2024-01-20: Draft

# Table of Contents

# 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

# 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 special and general-purpose 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

**Zcheri_mode**
: 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

*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, Zcheri_mode and Zcheri_pte are optional extensions in addition to Zcheri_purecap. Zcheri_mode requires supporting both Zcheri_purecap and Zcheri_legacy.

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, Zcheri_legacy and Zcheri_mode.

See Chapter 8 for compatibility with other RISC-V extensions.

## 1.3. Risks and Known Uncertainty

- All extensions could be divided up differently in 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
  - The instruction mnemonics could be the same regardless of CHERI mode
  - 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 XLENMAX=32 and XLENMAX=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!*

### 1.3.3. Suggested Mnemonic Renaming

Table 1 lists the currently proposed renames. Please update the table when new renames are proposed or confirmed.

| Current Name | Suggestion |
|---|---|
| CMOVE | CMV |
| CINCOFFSET | CADD |
| CINCOFFSETIMM | CADDI |

| Current Name | Suggestion |
|---|---|
| C.CINCOFFSET16CSP | C.CADDI16SP |
| C.CINCOFFSET4CSPN | C.CADDI4SPN |
| CLC/LC | CLCAP/LCAP |
| CSC/SC | CSCAP/SCAP |

*Table 1. Suggested instruction names*

*Renaming SC is not a choice. The store capability instruction must be renamed because it conflicts with store conditional from the RISC-V A extension.*

Further to the new proposed mnemonics in Table 1, the following general proposals have been discussed:

- Do not use the letter 'c' to indicate 'capability' or 'CHERI' because this conflicts with the already ratified RISC-V C extension
  - We previously discussed using other letters like 'p' for 'pointer' or 'f' for 'fat pointer' although 'f' is already used for floating point
- Do not change instruction mnemonics based on the current CHERI execution mode
  - For example, LW is always load word regardless of the CHERI mode, so the mnemonic CLW disappears
  - This facilitates writing the ISA specification as well as code maintenance in systems software like Linux
  - However, it also goes against intentionality and can make assembly code (which occurs very infrequently in real-world code) more difficult to understand without additional context
  - Both options could be supported by using assembler aliases

# 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 XLENMAX to be widest XLEN that the implementation supports.

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

## 2.1. Components of a Capability

Capabilities contain the software accessible fields described in this section.

### 2.1.1. Tag

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.

### 2.1.2. Architectural Permissions (AP)

**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 2.1.4), and bounds checks (see Section 2.1.5). 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 in 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 XLENMAX as shown in Table 2. A 4-bit vector encodes the permissions when XLENMAX=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.

| XLENMAX | Permissions width |
|---|---|
| 32 | 4 |
| 64 | 5 |

*Table 2. Permissions widths depending on XLENMAX*

| Encoding | R | W | C | X | ASR |
|---|---|---|---|---|---|
| 0b0000 | | | | | |
| 0b0001 | reserved | | | | |
| 0b0010 | | ✓ | | | |
| 0b0011 | | ✓ | ✓ | | |
| 0b0100 | ✓ | | | | |
| 0b0101 | ✓ | | ✓ | | |
| 0b0110 | ✓ | ✓ | | | |
| 0b0111 | ✓ | ✓ | ✓ | | |
| 0b1000 | ✓ | | | ✓ | |
| 0b1001 | ✓ | | ✓ | ✓ | |
| 0b1010 | ✓ | ✓ | | ✓ | |
| 0b1011 | ✓ | ✓ | ✓ | ✓ | |
| 0b1100 | ✓ | | | ✓ | ✓ |
| 0b1101 | ✓ | | ✓ | ✓ | ✓ |
| 0b1110 | ✓ | ✓ | | ✓ | ✓ |
| 0b1111 | ✓ | ✓ | ✓ | ✓ | ✓ |

*Table 3. Encoding of architectural permissions for XLENMAX=32*

The encoding in Table 3 is chosen to facilitate hardware implementations. Therefore, it can be worked out if the permissions are granted as follows:

- C-permission: bit 0 is set
- W-permission: bit 1 is set
- X-permission: bit 3 is set
- R-permission: bits 3 or 2 are set

- ASR-permission: bits 3 and 2 are set

A 5-bit vector encodes the permissions when XLENMAX=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.

| Bit | Name |
|---|---|
| 0 | C-permission |
| 1 | W-permission |
| 2 | R-permission |
| 3 | X-permission |
| 4 | ASR-permission |

*Table 4. Encoding of architectural permissions for XLENMAX=64*

*TODO: Confirm that we need a separate permissions format for 32-bit and 64-bit.*

*Valid capabilities must not have the permissions field set to a reserved value according to Table 3 when XLENMAX=32.*

## 2.1.3. 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.*

| XLENMAX | SDP width |
|---|---|
| 32 | 2 |
| 64 | 4 |

*Table 5. SDP widths depending on XLENMAX*

## 2.1.4. Sealed (S) Bit

**CHERI v9:** *The sealing bit is new (1-bit otype) and the old CHERI v9 otype no longer exists.*

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

In Zcheri_purecap, the sealing bit is used to implement immutable capabilities that describe function entry points. A program may jump to a sealed 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 CJALR instruction also seals the return address capability (if any) since it is the entry point to the caller function.

### 2.1.5. Bounds

**CHERI v9 Note:** *The bounds mantissa width is different in XLENMAX=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 base ≤ A < 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 2.2.

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, so it is 'embedded'

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

| XLENMAX | MW |
|---|---|
| 32 | 10 |
| 64 | 14 |

*Table 6. Mantissa width (MW) values depending on XLENMAX*

The exponent E indicates the position of T and B within the capability's address as described in Section 2.2. The bit width of the exponent (EW) is set depending on the value of XLENMAX. The maximum value of the exponent is calculated as follows:

```
CAP_MAX_E = XLENMAX - MW + 2
```

The possible values for EW and CAP_MAX_E are shown in Table 7.

| XLENMAX | EW | CAP_MAX_E |
|---|---|---|
| 32 | 5 | 24 |
| 64 | 6 | 52 |

*Table 7. Exponent widths and CAP_MAX_E depending on XLENMAX*

*The address and bounds must be representable in valid capabilities i.e. when the tag is set* (see Section 2.5).

### 2.1.6. Address

XLENMAX integer value that encodes the byte-address of a memory location.

| XLENMAX | Address width |
|---|---|
| 32 | 32 |
| 64 | 64 |

*Table 8. Address widths depending on XLENMAX*

### 2.1.7. Reserved Bits

Reserved bits available for future extensions to Zcheri_purecap.

*Reserved bits must be 0 in valid capabilities.*

## 2.2. Capability Encoding

**CHERI v9 Note:** *The encoding changes eliminate the concept of the in-memory format, and also increase precision for RV32. When EF=0, T and B are now shifted right rather than left within the address. Also, the bounds decoding for XLENMAX=32 uses a trick (see bit T8) to save one bit when encoding the exponent.*

The components of a capability are encoded as shown in Figure 1 and Figure 2 when XLENMAX=32 and XLENMAX=64 respectively.

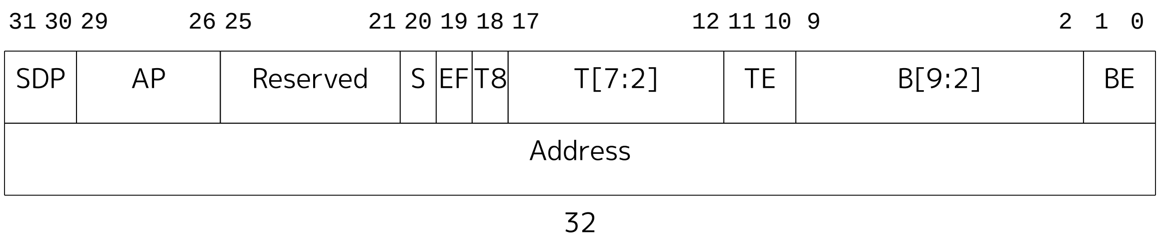


*Figure 1. Capability encoding when XLENMAX=32*

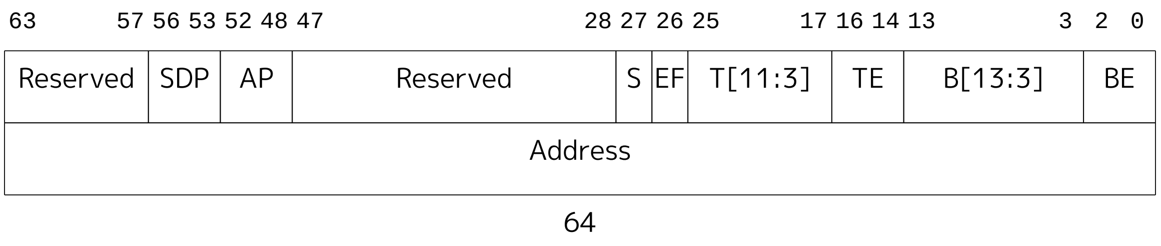


*Figure 2. Capability encoding when XLENMAX=64*

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.

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.*

```
EW        = (XLENMAX == 32) ? 5 : 6
CAP_MAX_E = XLENMAX - 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            = (XLENMAX == 32) ? T8 : 0
else:
    E               = CAP_MAX_E - ( (XLENMAX == 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[XLENMAX - 1:E + MW] + ct, T[MW - 1:0]    , {E{1'b0}} }
base:   b = { a[XLENMAX - 1:E + MW] + cb, B[MW - 1:0]    , {E{1'b0}} }
```

The corrections $c_t$ and $c_b$ are calculated as as shown below using the definitions in Table 9 and Table 10.

```
Ac = a[E + MW - 1:E + MW - 3]
Bc = B[MW - 1:MW - 3]
Tc = T[MW - 1:MW - 3]
R  = Bc - 1
```

| $A_c < R$ | $T_c < R$ | $c_t$ |
|---|---|---|
| false | false | 0 |
| false | true | +1 |
| true | false | -1 |
| true | true | 0 |

*Table 9. Calculation of top address correction*

| $A_c < R$ | $B_c < R$ | $c_b$ |
|---|---|---|
| false | false | 0 |
| false | true | +1 |
| true | false | -1 |
| true | true | 0 |

*Table 10. Calculation of base address correction*

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 XLENMAX=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 relies on using the 'spare' encodings where `T < B` to define a space boundary R, relative to the base, calculated by subtracting 1 from the top three bits of 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 $\text{space}_U$ in Figure 3 and the corrections $c_t$ and $c_b$ are computed accordingly. The overall effect is that at least $2^{E+MW}/8$ bytes below the base address and $2^{E+MW}/4$ bytes above the top address can roam out-of-bounds while still allowing the bounds to be correctly decoded.

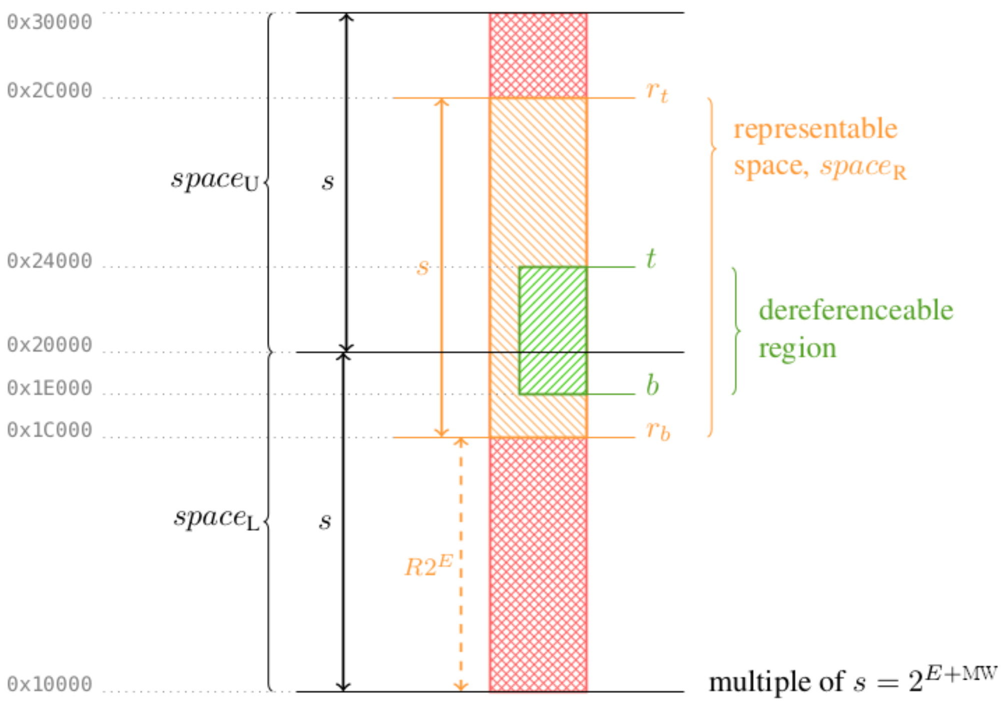


*Figure 3. Memory address bounds encoded within a capability*

A capability whose bounds cover the entire address space has 0 base and top equals $2^{XLENMAX}$, i.e. *t* is a XLENMAX + 1 bit value. However, *b* is a XLENMAX bit value and the size mismatch introduces additional complications when decoding, so the following condition is required to correct *t* for capabilities whose representable region wraps the edge of the address space:

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

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

## 2.3. NULL and Infinite Capabilities

**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 NULL has no permissions and its exponent E is CAP_MAX_E e.g. 52 when XLENMAX=64, so its bounds cover the entire address space such that the expanded base is 0 and top is $2^{XLENMAX}$. In contrast, the Infinity capability grants all permissions while its bounds also cover the whole address space.

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

| Field | Value | Comment |
|---|---|---|
| SDP | zeros | Grants no permissions |
| AP | zeros | Grants no permissions |
| S | zero | Unsealed |
| EF | zero | Internal exponent format |
| $T_8$ | zeros | Top address bit (XLENMAX=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 |

*Table 11. Field values of the NULL capability*

| Field | Value | Comment |
|---|---|---|
| SDP | ones | Grants all permissions |
| AP | ones | Grants all permissions |
| S | zero | Unsealed |
| EF | zero | Internal exponent format |
| $T_8$ | zeros | Top address bit (XLENMAX=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 |

*Table 12. Field values of the Infinite capability*

## 2.4. Representable Limit Check

Pointer arithmetic on capabilities must be checked to ensure that the new address is within the capability's representable region described in Section 2.2. The new address, after pointer arithmetic, is within the representable region 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 region if `b == b' && t == t'`.

Changing a capability's address to a value outside the representable region unconditionally clears the capability's tag.

*The encoding of the bounds depends upon the leading 1 of the address which is used to determine the exponent. If the leading 1 of the address moves then the bounds will need to be recalculated. Instructions like CINCOFFSET and CSETADDR update the address field but do not recalculate the bounds. Therefore, if the leading 1 moves relative to when the bounds were calculated then the tag is cleared on the result as the encoding has been invalidated.*

## 2.5. 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[MW - 1:MW - 2] != 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.

# 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.*

## 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 $2^{XLEN}$ - 1 is adjacent to the byte at address zero. A capability's representable region described in Section 2.2 is also circular, so address 0 is within the representable region of a capability where address $2^{XLENMAX}$ - 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. Therefore, each **x** 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.

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.

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 that the hardware automatically increments as instructions are executed. The pcc's metadata and tag are reset to the Infinity capability metadata and tag.

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

## 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.

- CGETTAG: inspects the tag of the input capability. The output is 1 if the tag is set and 0 otherwise
- CGETPERM: outputs the architectural (AP) and software-defined (SDP) permission fields of the input capability
- CGETBASE: outputs the expanded base address of the input capability
- CGETLEN: outputs the length of the input capability. Length is defined as `top - base`. The output is $2^{XLEN}$-1 when the capability's length is $2^{XLENMAX}$
- CRAM: outputs the nearest bounds alignment that a valid capability can represent
- CGETHIGH: outputs the compressed capability metadata
- CSETEQUALEXACT: compares two capabilities including tag, metadata and address
- CTESTSUBSET: tests whether the bounds and permissions of a capability are a subset of those from another capability

*CGETBASE and CGETLEN output 0 when a capability with malformed bounds is provided as an input (see Section 2.5).*

### 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.

- CSETADDR: set the address of a capability to an arbitrary address
- CINCOFFSET, CINCOFFSETIMM: increment the address of the input capability by an arbitrary offset
- CSETHIGH: replace a capability's metadata with an arbitrary value. The output tag is always 0
- CANDPERM: bitwise AND of a mask value with a bit map representation of the architectural (AP)

and software-defined (SDP) permissions fields

- CSETBOUNDS: set the base and length of a capability. The tag is cleared, if the encoding cannot represents the bounds exactly
- CSETBOUNDSINEXACT: 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
- CSEAL: seal capability
- CBUILDCAP: replace the base, top, address, permissions and mode fields of a capability with the fields from another capability
- CMOVE: move a capability from a **c** register to another **c** register

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

**CHERI v9 Note:** *CSETBOUNDS and CSETBOUNDSIMM perform the role of the old CSETBOUNDSEXACT while the new CSETBOUNDSINEXACT is the old CSETBOUNDS.*

### 3.3.3. Capability Load and Store Instructions

A load capability instruction, CLC, 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, CSC, 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.

CLC and CSC 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.

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*

### 3.3.4. Unconditional Integer Address Jumps

The indirect jump and link pcc (JALR.PCC) instruction allows unconditional jumps to a target address. The target address is provided in an **x** register; the new address is installed in the address field of the pcc. The address of the instruction following the jump (**pc** + 4) is written to an **x** register. JALR.PCC causes an exceptions when a minimum sized instruction at the target address is not within the bounds of the pcc or the target address is misaligned.

**CHERI v9 Note:** *This instruction is now modal and shares the same encoding with JALR.CAP when both Zcheri_purecap and Zcheri_legacy are supported.*

## 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 (AUIPCC) replaces the add upper immediate to **pc** instruction (AUIPC) at the same encoding. AUIPCC is used to build pcc-relative capabilities. AUIPCC 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 region. 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

The capability jump and link (CJAL) instruction replaces jump and link (JAL) at the same encoding. CJAL 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.

The capability jump and link register (CJALR) instruction replaces the jump and link register (JALR) instruction at the same encoding. This instruction 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, then setting the least significant bit of the result to zero. The capability with the address of the instruction following the jump (pcc + 4) is 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.

CJALR causes a CHERI exception when:

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

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

#### 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. For clarity, the mnemonics of these instructions are prefixed with the letter 'C' (e.g. LD becomes CLD, SW becomes CSW, etc) to distinguish them from the standard RISC-V instructions that do not have a **c** register operand.

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 removed in Zcheri_purecap and analogous CLEN-bit versions of those CSRs are added to the ISA as described in Section 3.6.

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 region 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 10.

### 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 39.

CSRRWI, CSRRS, CSRRSI, CSRRC and CSRRCI specify **x** registers and so only access the address field of the capability when specifying a capability CSR such as mtvecc. They calculate the final address using the standard RISC-V behaviour (set bits, clear bits etc.) and that final address is updated in the capability. The update typically uses the semantics of a CSETADDR instruction which clears the tag if the capability is sealed, or if the updated address is not representable. Table 39 shows the exact action taken for each capability width CSR.

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 removes 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 removed because they are designated to hold addresses, but are only XLEN bits wide. The removed registers are replaced with CLEN+1 bits wide registers. The new CSRs are analogous to the original, removed RISC-V CSRs although at different CSR numbers as shown in Table 17, Table 18, Table 19 and Table 20. Therefore, the specification of the address field for the new capability CSRs remains the same as the corresponding, removed CSR which is described in (RISC-V, 2023) and the specifications of relevant RISC-V extensions.

| Replaced CSR | Address | Prerequisites | Permissions | Description |
|---|---|---|---|---|
| dpc | 0x7b1 | Sdext | DRW, ASR-permission | Debug Program Counter Capability |
| dscratch0 | 0x7b2 | Sdext | DRW, ASR-permission | Debug Scratch Capability 0 |
| dscratch1 | 0x7b3 | Sdext | DRW, ASR-permission | Debug Scratch Capability 1 |

*Table 13. Debug-mode CSRs removed in Zcheri_purecap*

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

*Table 14. Machine-mode CSRs removed in Zcheri_purecap*

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

*Table 15. Supervisor-mode CSRs removed in Zcheri_purecap*

| Replaced CSR | Address | Prerequisites | Permissions | Description |
|---|---|---|---|---|
| jvt | 0x017 | Zcmt | URW | Jump Vector Table Capability |

*Table 16. User-mode CSRs removed in Zcheri_purecap*

| Zcheri_purecap CSR | Address | Replaced CSR | Prerequisites | Permissions | Description |
|---|---|---|---|---|---|
| dpcc | 0x7b9 | dpc | Sdext | DRW, ASR-permission | Debug Program Counter Capability |
| dscratch0c | 0x7ba | dscratch0 | Sdext | DRW, ASR-permission | Debug Scratch Capability 0 |
| dscratch1c | 0x7bb | dscratch1 | Sdext | DRW, ASR-permission | Debug Scratch Capability 1 |

*Table 17. New debug-mode CSRs in Zcheri_purecap replacing RISC-V CSRs*

| Zcheri_purecap CSR | Address | Replaced CSR | Prerequisites | Permissions | Description |
|---|---|---|---|---|---|
| mtvecc | 0x765 | mtvec | M-mode | MRW, ASR-permission | Machine Trap-Vector Base-Address Capability |
| mscratchc | 0x760 | mscratch | M-mode | MRW, ASR-permission | Machine Scratch Capability |
| mepcc | 0x761 | mepc | M-mode | MRW, ASR-permission | Machine Exception Program Counter Capability |

*Table 18. New machine-mode CSRs in Zcheri_purecap replacing RISC-V CSRs*

| Zcheri_purecap CSR | Address | Replaced CSR | Prerequisites | Permissions | Description |
|---|---|---|---|---|---|
| stvecc | 0x505 | stvec | S-mode | SRW, ASR-permission | Supervisor Trap-Vector Base-Address Capability |
| sscratchc | 0x540 | sscratch | S-mode | SRW, ASR-permission | Supervisor Scratch Capability |
| sepcc | 0x541 | sepc | S-mode | SRW, ASR-permission | Supervisor Exception Program Counter Capability |

*Table 19. New supervisor-mode CSRs in Zcheri_purecap replacing RISC-V CSRs*

| Zcheri_purecap CSR | Address | Replaced CSR | Prerequisites | Permissions | Description |
|---|---|---|---|---|---|
| jvtc | 0x417 | jvt | Zcmt | URW | Jump Vector Table Capability |

*Table 20. New user-mode CSRs in Zcheri_purecap replacing RISC-V CSRs*

Zcheri_purecap also introduces the new unprivileged CSRs shown in Table 21.

| Extended CSR | CLEN Address | Prerequisites | Permissions | Description |
|---|---|---|---|---|
| pcc | 0xcb0 | none | URO | User Program Counter Capability (to allow reading in legacy mode) |

*Table 21. User-mode CSRs added in Zcheri_purecap*

## 3.7. Machine-Level CSRs

Zcheri_purecap adds new M-mode capability CSRs and extends some of the existing RISC-V CSRs with new functions. pcc must grant ASR-permission to access M-mode CSRs regardless of the RISC-V privilege mode.

### 3.7.1. Machine ISA Register (misa)

The **misa** register operates as described in (RISC-V, 2023) except for the MXL (Machine XLEN) field. The MXL field encodes the native base integer ISA width as shown in Table 22. Only 1 and 2 are supported values for MXL and the field must be read-only in implementations supporting Zcheri_purecap. The effective XLEN in M-mode, MXLEN, is given by the setting of MXL, or has a fixed value if **misa** is zero.

| MXL | XLEN |
|---|---|
| 1 | 32 |
| 2 | 64 |
| 3 | ~~128~~ |

*Table 22. Encoding of MXL field in* **misa**

*RV128 is not currently supported by any CHERI extension*

*A further CHERI extension, Zcheri_legacy, optionally makes MXL writeable, so implementations that support multiple base ISAs must support both Zcheri_purecap and Zcheri_legacy.*

## 3.7.2. 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.

The encoding of the SXL and UXL fields is the same as the MXL field of **misa**, shown in Table 22. 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.

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

## 3.7.3. 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.

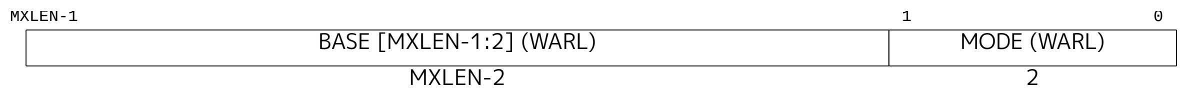


*Figure 4. Machine-mode trap-vector base-address register*

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

The mtvecc register is an extension to mtvec that holds a capability. Its reset value is the Infinity capability. The capability represents an executable vector.

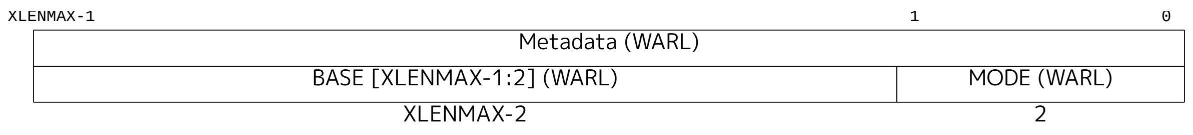


*Figure 5. Machine-mode trap-vector base-capability register*

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 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.*

## 3.7.5. 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.

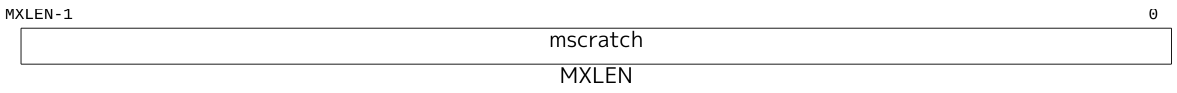


*Figure 6. Machine-mode scratch register*

## 3.7.6. Machine Scratch Register Capability (mscratchc)

The mscratchc register is an extension to mscratch that is able to hold a capability. Its reset value is the NULL capability.

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

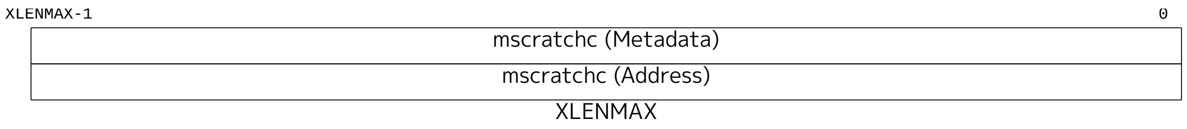


*Figure 7. Machine-mode scratch capability register*

## 3.7.7. Machine Exception Program Counter (mepc)

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

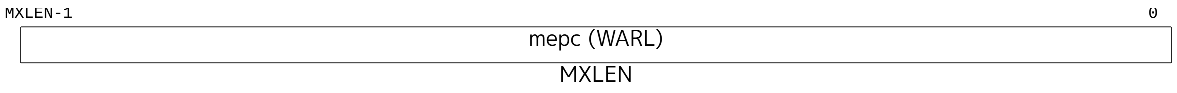


*Figure 8. Machine exception program counter register*

## 3.7.8. Machine Exception Program Counter Capability (mepcc)

The mepcc register is an extension to mepc that is able to hold a capability. Its reset value is the NULL capability.

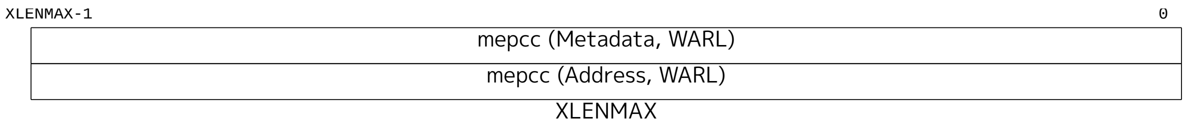


*Figure 9. Machine exception program counter capability register*

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 region.

*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 40, mepcc is an executable vector, so it need not 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.

## 3.7.9. 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 23 and Table 24 respectively. The behavior and usage of mcause otherwise remains as described in (RISC-V, 2023).

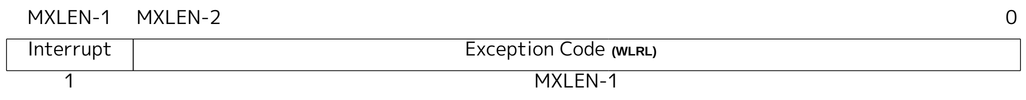


*Figure 10. Machine cause register*

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

| 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 | *Reserved* |
| 0 | 11 | Environment call from M-mode |
| 0 | 12 | Instruction page fault |
| 0 | 13 | Load page fault |
| 0 | 14 | *Reserved* |
| 0 | 15 | Store/AMO page fault |
| 0 | 16-23 | *Reserved* |
| 0 | 24-27 | *Designated for custom use* |
| **0** | **28** | **CHERI fault** |
| 0 | 29-31 | *Designated for custom use* |
| 0 | 32-47 | *Reserved* |
| 0 | 48-63 | *Designated for custom use* |
|  | ≥64 | *Reserved* |

*Table 23. Machine cause register (mcause) values after trap. Entries added in Zcheri_purecap are in* **bold**

| Priority | Exc.Code | Description |
|---|---|---|
| *Highest* | 3 | Instruction address breakpoint |
|  | **28** | **Prior to instruction address translation:**<br>**CHERI fault** |
|  | 12, 1 | During instruction address translation:<br>First encountered page fault or access fault |
|  | 1 | With physical address for instruction:<br>Instruction access fault |
|  | 2<br>0<br>8,9,11<br>3<br>3 | Illegal instruction<br>Instruction address misaligned<br>Environment call<br>Environment break<br>Load/store/AMO address breakpoint |
|  | **28** | **Prior to address translation for an explicit memory access or jump:**<br>**CHERI fault** |
|  | 4,6 | Optionally:<br>Load/store/AMO address misaligned |
|  | 13, 15, 5, 7 | During address translation for an explicit memory access:<br>First encountered page fault or access fault |
|  | 5,7 | With physical address for an explicit memory access:<br>Load/store/AMO access fault |
| *Lowest* | 4,6 | If not higher priority:<br>Load/store/AMO address misaligned |

*Table 24. Synchronous exception priority in decreasing priority order. Entries added in Zcheri_purecap are in* **bold**

### 3.7.10. 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.

### 3.7.11. 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 11 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.

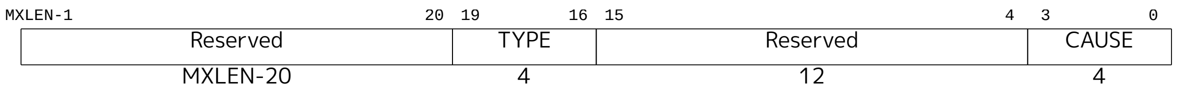


*Figure 11. Machine trap value register*

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 25 and Table 26 respectively.

| 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 25. Encoding of TYPE field*

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

*Table 26. Encoding of CAUSE field*

## 3.8. Supervisor-Level CSRs

Zcheri_purecap adds new S-mode capability CSRs and extends some of the existing RISC-V CSRs with 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.

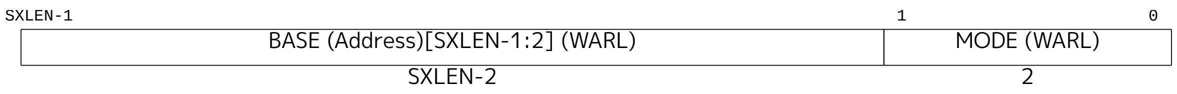


*Figure 12. Supervisor trap-vector base-address register*

### 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 an extension to stvec that is able to hold a capability. Its reset value is the Infinity capability.

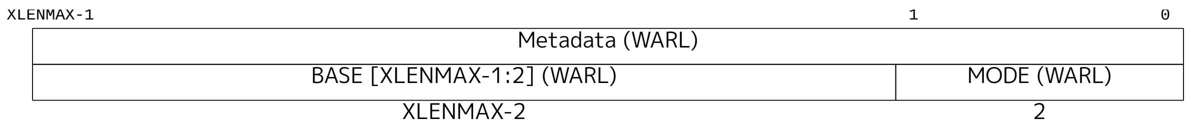


*Figure 13. Supervisor trap-vector base-capability register*

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.

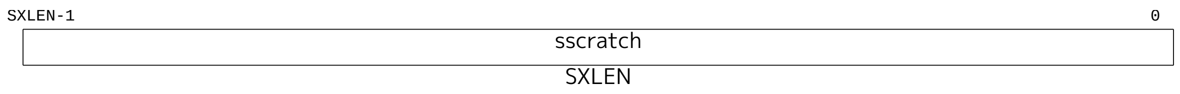


*Figure 14. Supervisor-mode scratch register*

### 3.8.4. Supervisor Scratch Registers (sscratchc)

The sscratchc register is an extension to sscratch that is able to hold a capability. Its reset value is the NULL capability.

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

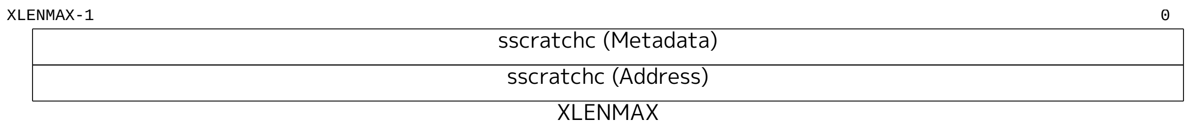


*Figure 15. Supervisor scratch capability register*

### 3.8.5. Supervisor Exception Program Counter (sepc)

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

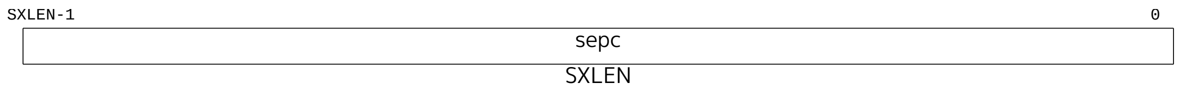


*Figure 16. Supervisor exception program counter register*

### 3.8.6. Supervisor Exception Program Counter Capability (sepcc)

The sepcc register is an extension to sepc that is able to hold a capability. Its reset value is the NULL capability.

As shown in Table 40, 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.

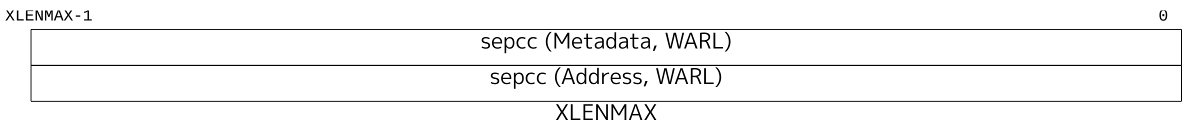


*Figure 17. Supervisor exception program counter capability register*

### 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 27 and Table 24 respectively. The behavior and usage of scause otherwise remains as described in (RISC-V, 2023).

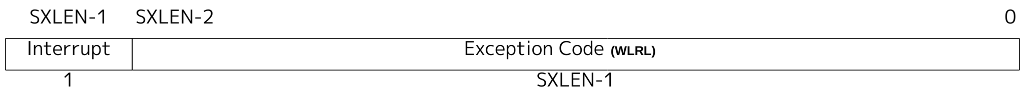


*Figure 18. Supervisor cause register*

| Interrupt | Exception Code | Description |
|---|---|---|
| 1 | 0 | *Reserved* |
| 1 | 1 | Supervisor software interrupt |
| 1 | 2-4 | *Reserved* |
| 1 | 5 | Supervisor timer interrupt |
| 1 | 6-8 | *Reserved* |
| 1 | 9 | Supervisor external interrupt |
| 1 | 10-15 | *Reserved* |
| 1 | ≥16 | *Designated for platform use* |
| 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 | *Reserved* |
| 0 | 12 | Instruction page fault |
| 0 | 13 | Load page fault |
| 0 | 14 | *Reserved* |
| 0 | 15 | Store/AMO page fault |
| 0 | 16-23 | *Reserved* |
| 0 | 24-27 | *Designated for custom use* |
| **0** | **28** | **CHERI fault** |
| 0 | 29-31 | *Designated for custom use* |
| 0 | 32-47 | *Reserved* |
| 0 | 48-63 | *Designated for custom use* |
|  | ≥64 | *Reserved* |

*Table 27. Supervisor cause register (scause) values after trap. Causes added in Zcheri_purecap are in* **bold**

### 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 19 to assist software in handling the trap.

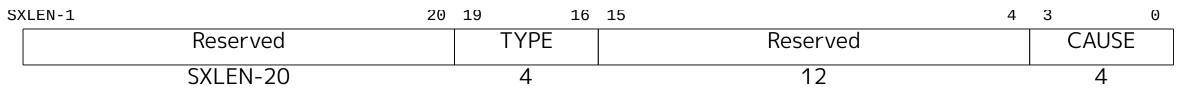


*Figure 19. Supervisor trap value register*

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 25 and Table 26 respectively.

## 3.9. Unprivileged CSRs

Unlike machine and supervisor level CSRs, Zcheri_purecap does not require pcc to grant ASR-permission to access privileged CSRs.

### 3.9.1. Program Counter Capability (pcc)

The pcc is made visible in a CSR. This provides access to an Infinity capability while in debug mode without executing AUIPCC.

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

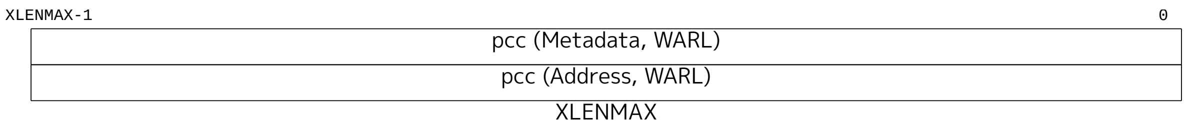


*Figure 20. Program Counter Capability*

As shown in Table 40, pcc is an executable vector, so it need not be able to hold all possible invalid addresses.

## 3.10. CHERI Exception handling

`auth_cap` *is ddc for Legacy mode and* `cs1` *for Capability Mode*

| 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) |
| 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 | | | | | |
| CJAL, JAL, Conditional branches (BEQ, BNE, BLT[U], BGE[U]) | 28 | 2 | 3 | pcc length | any byte of 16-bit instruction at target out of pcc bounds |
| indirect jumps and conditional branches additional exception checks | | | | | |
| indirect jumps and conditional branches | 28 | 2 | 0 | `cs1` tag | not(`cs1.tag`) |
| indirect jumps and conditional branches | 28 | 2 | 1 | `cs1` seal | isCapSealed(`cs1`) |
| indirect jumps and conditional branches | 28 | 2 | 2 | `cs1` permission | not(`cs1`.X-permission) |
| indirect jumps and conditional branches | 28 | 2 | 3 | `cs1` length | any byte of 16-bit instruction at target out of `cs1` bounds |
| Load additional exception checks | | | | | |
| all loads | 28 | 1 | 0 | `auth_cap` tag | not(`auth_cap.tag`) |
| all loads | 28 | 1 | 1 | `auth_cap` seal | isCapSealed(`auth_cap`) |
| all loads | 28 | 1 | 2 | `auth_cap` permission | not(`auth_cap`.R-permission) |

| Instructions | Xcause | Xtval. TYPE | Xtval. CAUSE | Description | Check |
|---|---|---|---|---|---|
| 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, all cbos | 28 | 1 | 2 | `auth_cap` permission | AMO only: not(`auth_cap`.R-permission) |
| all stores, all atomics, all cbos | 28 | 1 | 2 | `auth_cap` permission | not(auto_cap.W-permission) |
| all stores, all atomics | 28 | 1 | 3 | `auth_cap` length | any byte of access[1] out of `auth_cap` bounds |
| capability stores, all atomics | 6 | N/A | N/A | Misaligned store/AMO | Misaligned capability store or AMO |

Table 28. Valid CHERI exception combination description

*Indirect branches are CJALR, JALR, JALR.PCC, JALR.CAP, conditional branches are Conditional branches (BEQ, BNE, BLT[U], BGE[U]).*

*CBO.ZERO.CAP, CBO.ZERO issues as a cache line wide store*

*[1]Other CBOs (CBO.FLUSH.CAP, CBO.FLUSH, CBO.CLEAN.CAP, CBO.CLEAN, CBO.INVAL.CAP, CBO.INVAL) require at least one byte of the access to be in `auth_cap` bounds*

## 3.11. 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.

When the hart attempts to store or load 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

## 3.12. 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.

### 3.12.1. 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 some CSRs, such as mtvec and mepc (see Table 40), need not be able to hold all possible invalid addresses. Prior to writing these CSRs, implementations may convert an invalid address into some other invalid address that the register is capable of holding. However, these registers hold capabilities in Zcheri_purecap and the bounds encoding depends on the address value, so implementations must not convert invalid addresses to other arbitrary invalid address in an unrestricted manner. The following procedure must be used instead when writing a capability A to these CSRs:

1. If A's address cannot be held then convert it to another address that the CSR can hold
2. If conversion *was* required, then A's tag is cleared if A is sealed or if the new address is not representable — this is equivalent to the semantics of CSETADDR
3. Write the final (potentially modified) version of capability A to the CSR e.g. mtvecc, mepcc, etc.

This implies that sealed capabilities will always get their tags cleared when written to these CSRs unless the specification explicitly states that the CSR behaves otherwise (see mepcc and sepcc). Also notes that pcc is available in a read-only CSR. It can be written with CJALR instruction which automatically unseals the capability *before* the invalid address conversion above.

# Chapter 4. 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.

## 4.1. 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.

## 4.2. Core Debug Registers

Zcheri_purecap removes debug CSRs that are designated to hold addresses and replaces them with analogous CSRs able to hold capabilities. The removed debug CSRs are listed in Table 13 and the new CSRs are listed in Table 17.

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).

### 4.2.1. 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.

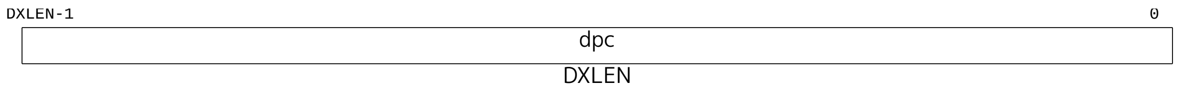


*Figure 21. Debug program counter*

### 4.2.2. Debug Program Counter Capability (dpcc)

The dpcc register is a extension to dpc that is able to hold a capability. Its reset value is the NULL capability.

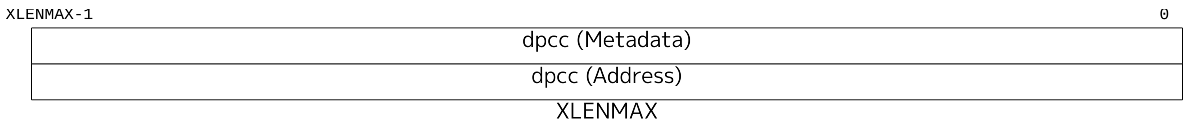


*Figure 22. Debug program counter capability*

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 updates pcc.

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).

Additionally, the pcc is updated as follows:

- All metadata is set to the Infinity capability
  - The pcc may be used as a source of the Infinity capability in debug mode to allow other capabilities to be created and written into memory.

When resuming, the hart's pcc is updated to the capability stored in dpcc. A debugger may write dpcc to change where the hart resumes and its mode, permissions, sealing or bounds.

### 4.2.3. 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. Its reset value is the NULL capability. dscratch0 is extended into dscratch0c.

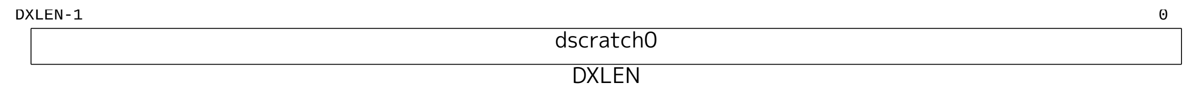


*Figure 23. Debug scratch 0 register*

### 4.2.4. Debug Scratch Register 0 (dscratch0c)

The dscratch0c register is a CLEN-bit plus tag bit extension to dscratch0 that is able to hold a capability. Its reset value is the NULL capability.

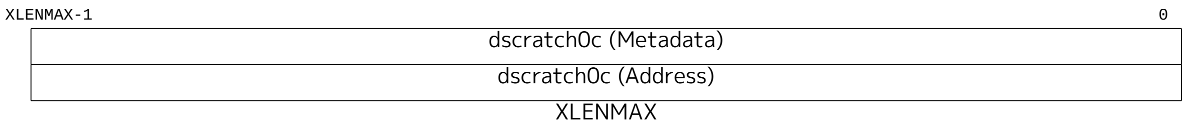


*Figure 24. Debug scratch 0 capability register*

### 4.2.5. 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. Its reset value is the NULL capability. dscratch1 is extended into dscratch1c.

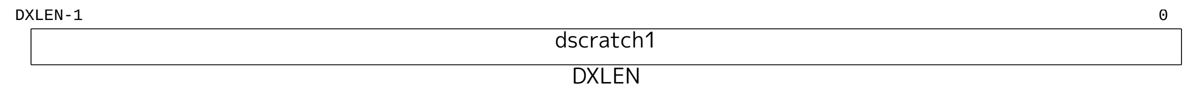


*Figure 25. Debug scratch 0 register*

### 4.2.6. Debug Scratch Register 1 (dscratch1c)

The dscratch1c register is a CLEN-bit plus tag bit extension to dscratch1 that is able to hold a capability. Its reset value is the NULL capability.

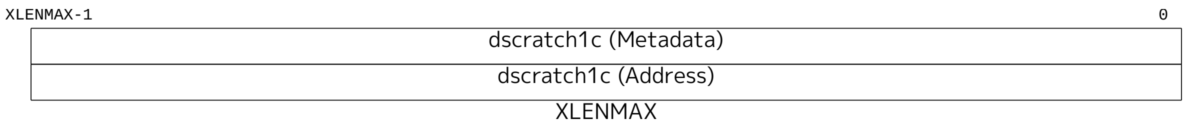


*Figure 26. Debug scratch 1 capability register*

# Chapter 5. "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.

## 5.1. 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 27, Figure 28 and Figure 29 respectively.

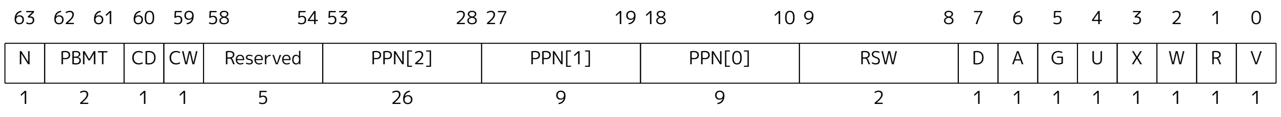


*Figure 27. Sv39 page table entry*

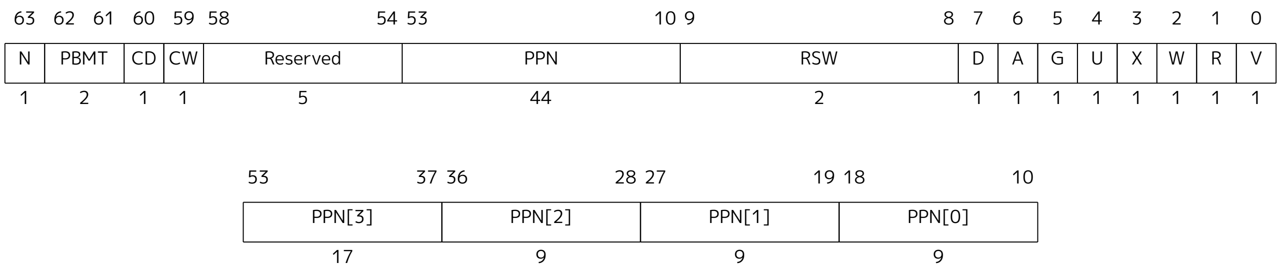


*Figure 28. Sv48 page table entry*

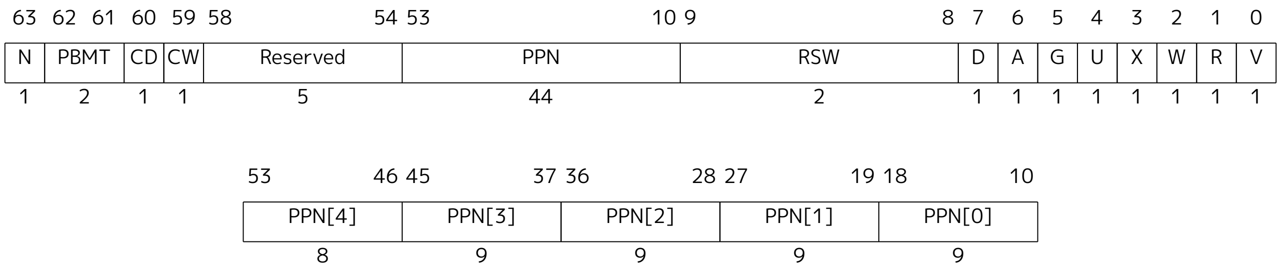


*Figure 29. Sv57 page table entry*

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 pcc grants store capability 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 this feature.*

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 pcc grants store capability 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 execute, the pcc grants store capability 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.*

## 5.2. 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 30 when XLEN=64.

| 63 | 62 | 61 | 60 8 | 7 | 6 | 5 4 | 3 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| STCE | PBMTE | CDE | WPRI | CBZE | CBCFE | CBIE | WPRI | FIOM |
| 1 | 1 | 1 | 55 | 1 | 1 | 1 | 2 | 3 |
| 1 | | | | | | | | |

*Figure 30. 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 6. "Zcheri_legacy" Extension for CHERI Legacy Mode

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

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 disable 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 the RISC-V ISA (RV32I and RV64I).

## 6.1. 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 current privilege level and the value of the CME bit in menvcfg and senvcfg for S-mode and U-mode. M-mode is always in Capability mode.

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 8).

*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), so the instruction is CLW from Zcheri_purecap, or ddc (Legacy mode), so the instruction is simply LW.

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 configured CHERI execution mode is Legacy by setting CME=0 in menvcfg or senvcfg as required, and the Infinity capability is installed in the pcc and ddc such that:

- Tags are set
- Capabilities are unsealed
- All permissions are granted

- The bounds authorise accesses to the entire address space i.e base is 0 and top is $2^{XLENMAX}$

## 6.2. Zcheri_legacy Instructions

Zcheri_legacy does not introduce new instructions to the base RISC-V integer ISA. However, the behavior of some existing instructions changes depending on the current CHERI execution mode.

### 6.2.1. 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 CLC when the CHERI execution mode is Capability; the instruction behaves as described in Section 3.3. That encoding is LC when the mode is Legacy. In this case, 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 CSC when the CHERI execution mode is Capability; the instruction behaves as described in Section 3.3. That encoding is SC when the mode is Legacy. In this case, the capability authorising the memory access is ddc, so the effective address is obtained by adding the **x** register to the sign-extended offset.

### 6.2.2. Unconditional Capability Jumps

The indirect jump and link pcc (JALR.PCC) instruction shares the same encoding with a new indirect jump and link capability (JALR.CAP) instruction. JALR.PCC is a Zcheri_purecap instruction executed when the mode is Capability as described in Section 3.3.4. In Legacy mode, the encoding is executed as JALR.CAP which allows unconditional jumps to a target capability. The target capability is provided in a **c** register and is written to pcc. The pcc of the next instruction following the jump (pcc + 4) is written to a **c** register. JALR.CAP cause CHERI exceptions when:

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

JALR.CAP causes an instruction address misaligned exception when the target address is misaligned.

*JALR.CAP can be used to change the current CHERI execution mode when the implementation supports Zcheri_mode.*

## 6.3. 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.

### 6.3.1. 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 is CJAL when the CHERI execution mode is Capability; the instruction behaves as described in Section 3.4. That encoding is JAL 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 CJALR when the CHERI execution mode is Capability; the instruction behaves as described in Section 3.4. That encoding is JALR 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.

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

## 6.3.2. Conditional Branches

The behaviour is as shown in Section 3.4.2.2.

## 6.3.3. 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.

## 6.3.4. 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 to legacy code (e.g. mtvec) and the full capability is also visible through an alias (e.g. mtvecc). These are referred to as *extended CSRs*.

Extended CSRs are accessible through two addresses, and the address determines the access width.

When the XLEN-bit alias is used by CSRRW:

- 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 39.
- Only XLEN bits are read from the capability address field, which is zero extended to the destination **x** register.

When the CLEN-bit alias is used by CSRRW:

- 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 39).
- The full capability is written to destination **c** register.

When either alias is used by another CSR instruction (CSRRWI, 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 39 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 accessing either alias in this case - the 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.

## 6.4. Integrating Zcheri_legacy with Sdext

A new debug default data capability (dddc) CSR is added at the CSR number shown in Table 29.

## 6.5. Debug Default Data Capability (dddc)

dddc is a register that is able to hold a capability. Its reset value is the NULL capability. The address is shown in Table 29.

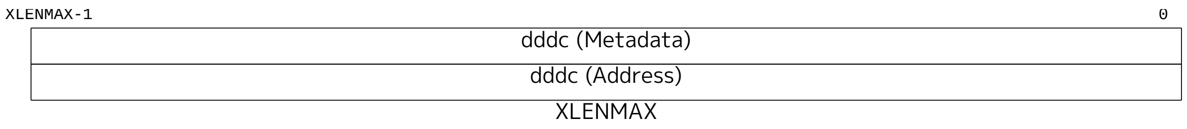


*Figure 31. Debug default data capability*

Upon entry to debug mode, ddc is saved in dddc. ddc's metadata is set to the Infinity 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.

## 6.6. Disabling CHERI Features

**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 most CHERI features. For example, executing in a privilege mode where the effective XLEN is less than XLENMAX. The following occurs when executing code in a privileged that has CHERI disabled:

- The CHERI instructions in Section 3.3 (and Section 9.5 if Zcheri_mode is supported) 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 CME bits in menvcfg and senvcfg have no effect whilst CHERI is disabled.

Security checks continue to be enforced when CHERI is disabled regardless of the reason. The last capability installed in pcc and ddc before disabling CHERI will be used to authorise instruction execution and data memory accesses.

## 6.7. Added CLEN-wide CSRs

Zcheri_legacy adds the CLEN-wide CSRs shown in Table 29.

| Extended CSR | CLEN Address | Prerequisites | Permissions | Description |
|---|---|---|---|---|
| dddc | 0x7bc | Sdext | DRW, ASR-permission | Debug Default Data Capabilty (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 |

*Table 29. CLEN-wide CSRs added in Zcheri_legacy*

### 6.7.1. Machine ISA Register (misa)

Zcheri_legacy eliminates some restrictions for MXL imposed in Zcheri_purecap to allow implementations supporting multiple base ISAs. Namely, the MXL field, that encodes the native base integer ISA width as shown in Table 22, may be writable.

Setting the MXL field to a value that is not XLENMAX disables most CHERI features and instructions as described in Section 6.6.

### 6.7.2. 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.

Zcheri_legacy requires that lower-privilege modes have XLEN settings less than or equal to the next-higher privilege mode. WARL field behaviour restricts programming so that it is not possible to program MXL, SXL or UXL to violate this rule.

Setting the SXL or UXL field to a value that is not XLENMAX disables most CHERI features and instructions, as described in Section 6.6, while in that privilege mode.

Whenever XLEN in any mode is set to a value less than XLENMAX, 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 the entire widest supported XLEN in the destination register. Similarly, **pc** bits above XLEN are ignored, and when the **pc** is written, it is sign-extended to fill XLENMAX. 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.

### 6.7.3. 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. mtdc's reset value is the NULL capability.

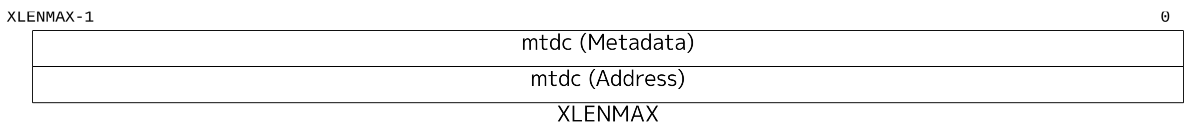


*Figure 32. Machine-mode trap data capability register*

### 6.7.4. Machine Environment Configuration Register (menvcfg)

Zcheri_legacy adds a new enable bit to menvcfg as shown in Figure 33.

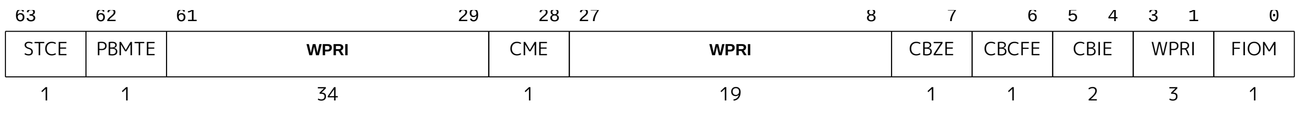


*Figure 33. Machine environment configuration register* (**menvcfg**)

The CHERI Mode Enable (CME) bit controls whether less privileged levels (e.g. S-mode and U-mode) execute in Capability or Legacy mode. When CME=1, the CHERI execution mode is Capability. When CME=0, the mode is Legacy.

### 6.7.5. 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. stdc's reset value is the NULL capability.

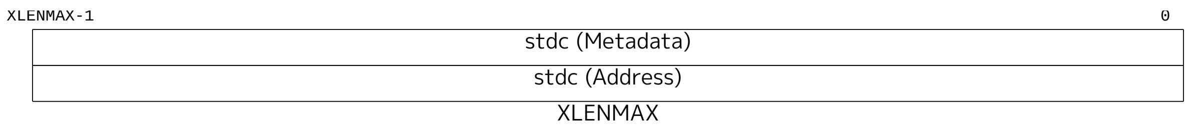


*Figure 34. Supervisor trap data capability register* (**stdc**)

### 6.7.6. Supervisor Environment Configuration Register (senvcfg)

The **senvcfg** register operates as described in the RISC-V Privileged Specification. Zcheri_legacy adds one new enable bit as shown in Figure 35.

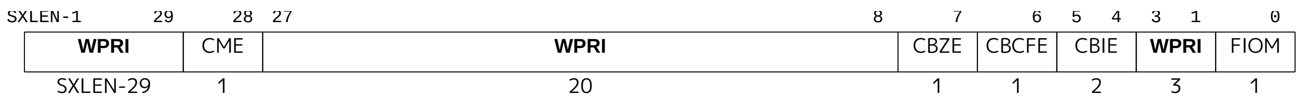


*Figure 35. Supervisor environment configuration register* (**senvcfg**)

The CHERI Mode Enable (CME) bit controls whether U-mode executes in Capability or Legacy mode. When CME=1, the CHERI execution mode is Capability. When CME=0, the mode is Legacy.

### 6.7.7. 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 Infinity capability.

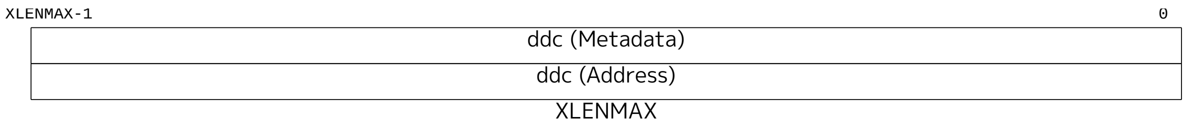


*Figure 36. Unprivileged default data capability register*

# Chapter 7. "Zcheri_mode" Extension for CHERI Execution Mode

Zcheri_mode is an optional extension to Zcheri_legacy. Implementations that support Zcheri_mode allow fine-grained switching between Capability and Legacy modes using indirect jump instructions.

## 7.1. CHERI Execution Mode

Zcheri_mode adds a new CHERI execution mode bit (M) to capabilities. The mode bit is encoded as shown in Figure 37 and Figure 38. The current CHERI execution mode is give by the M bit of the pcc and the CME bits in menvcfg and senvcfg as follows:

- The mode is Capability when the M bit of the pcc is 1 and the effective CME=1 for the current privilege level
- The mode is Legacy when the effective CME=0 for the current privilege level
- The mode is Legacy when the M bit of the pcc is 0 and the effective CME=1 for the current privilege level

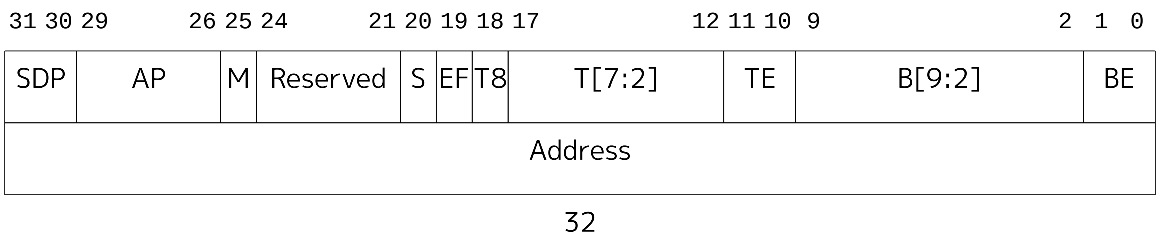


*Figure 37. Capability encoding when XLENMAX=32 and Zcheri_mode is supported*

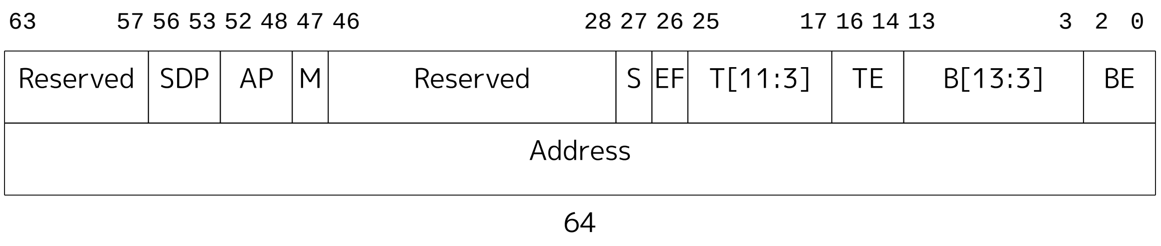


*Figure 38. Capability encoding when XLENMAX=64 and Zcheri_mode is supported*

Zcheri_mode allows the M bit to 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.

## 7.2. Zcheri_mode Instructions

Zcheri_mode introduces new instructions to the base RISC-V integer ISA in addition to the instructions added in Zcheri_purecap. The new instructions in Zcheri_mode allows inspecting the CHERI mode bit in capabilities and changing the current CHERI execution mode.

### 7.2.1. Capability Manipulation Instructions

A new Section 8.1.9 instruction allows setting a capability's CHERI execution mode to the indicated value. The output is written to an unprivileged **c** register, not pcc.

### 7.2.2. Mode Change Instructions

A new CHERI execution mode switch (CMODESWITCH) 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 CMODESWITCH is Capability and vice-versa. This instruction effectively writes the CHERI execution mode bit M of the capability currently installed in the pcc.

### 7.2.3. Unconditional Capability Jumps

Zcheri_mode allows changing the current CHERI execution mode when executing CJALR or JALR.CAP.

## 7.3. Integrating Zcheri_mode with Sdext

**CHERI v9 Note:** *The mode change instruction CMODESWITCH is new and the requirement to optionally support it in debug mode is also new.*

In addition to the changes described in Chapter 4 and Section 6.4, Zcheri_mode allows CMODESWITCH to act as an illegal instruction when it is executed while in debug mode.

# Chapter 8. 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 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. CJALR)
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.*

## 8.1. "Zcheri_purecap", "Zcheri_legacy" and "Zcheri_mode" Extensions for CHERI

### 8.1.1. JALR.PCC

See JALR.CAP.

### 8.1.2. JALR.CAP

**CHERI v9 Note:** *These instructions used to have separate encodings in CHERI v9. The instructions depend on the CHERI execution mode and now they share the same* **new** *encoding.*

#### Synopsis

Indirect jump and link (via integer address or capability)

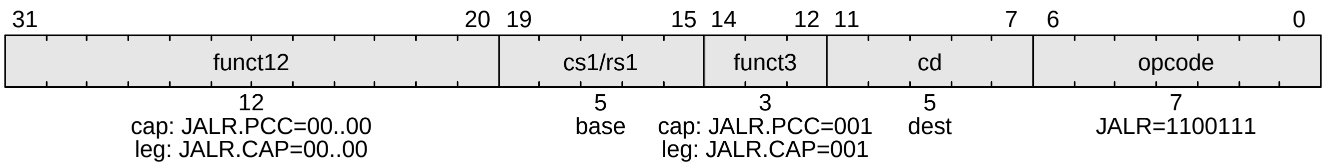
#### Capability Mode Mnemonic

```
jalr.pcc rd, rs1
```

#### Legacy Mode Mnemonic

```
jalr.cap cd, cs1
```

#### Encoding



#### Capability Mode Description

JALR.PCC allows unconditional jumps to a target integer address. The target address in `rs1` is installed in the address field of the pcc. The address of the instruction following the jump (pcc + 4) is written to `rd`. This is identical to the standard JALR instruction, but with zero offset.

#### Legacy Mode Description

JALR.CAP allows unconditional jumps to a target capability. The capability in `cs1` is installed in pcc. The pcc of the next instruction following the jump (pcc + 4) is sealed and written to `cd`. This instruction can be used to change the current CHERI execution mode and is identical to CJALR but with zero offset.

#### Exception

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 | JALR.PCC | JALR.CAP | 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 |

*The instructions on this page are either PC relative or may update the pcc. Therefore an implementation may make them illegal in debug mode.*

**Prerequisites JALR.PCC**

Zcheri_purecap

**Prerequisites JALR.CAP**

Zcheri_legacy

**Operation JALR.PCC**

TODO

**Operation JALR.CAP**

TODO

### 8.1.3. CMOVE

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

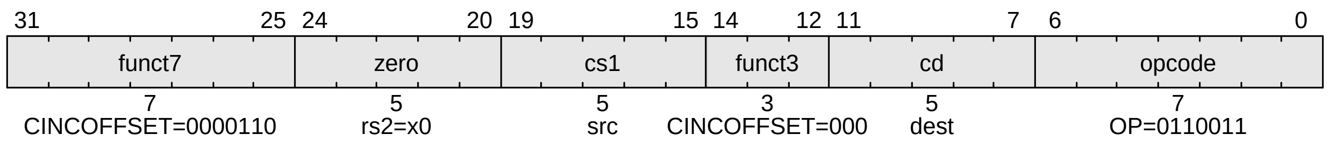
**Synopsis**

Capability move

**Mnemonic**

`cmove cd, cs1`

**Encoding**



*CMOVE is encoded as CINCOFFSET with* `rs2=x0`.

**Description**

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

**Prerequisites**

Zcheri_purecap

**Operation**

TODO

### 8.1.4. CMODESWITCH

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

**Synopsis**

Switch CHERI execution mode

**Mnemonics**

`cmodeswitch`

**Encoding**

| 31–25 | 24–20 | 19–15 | 14–12 | 11–7 | 6–0 |
|---|---|---|---|---|---|
| funct7 | funct5 | funct5 | funct3 | funct5 | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| CMS=0001001 | CMS=00000 | CMS=00000 | CMS=001 | CMS=000 | OP=0110011 |

**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.

*The instructions on this page are either PC relative or may update the pcc. Therefore an implementation may make them illegal in debug mode.*

**Prerequisites**

Zcheri_mode

**CModeSwitch Operation**

```
TODO
```

### 8.1.5. CINCOFFSETIMM

See CINCOFFSET.

### 8.1.6. CINCOFFSET

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

**CHERI v9 Note:** *the immediate format has changed*

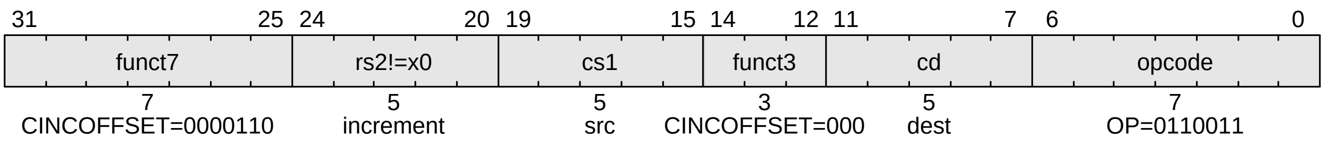
**Synopsis**

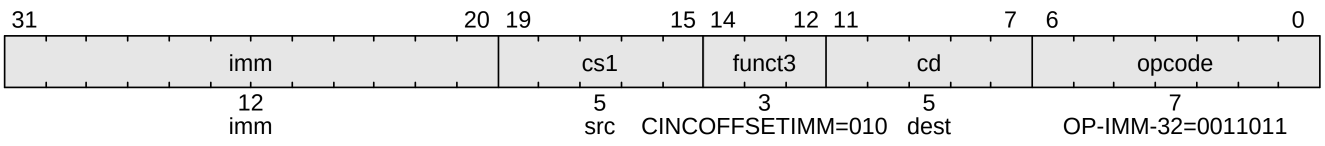
Capability pointer increment

**Mnemonic**

```
cincoffset cd, cs1, rs2
cincoffsetimm cd, cs1, imm
```

**Encoding**





*CINCOFFSET with `rs2=x0` is decoded as CMOVE instead, the key difference being that tagged and sealed capabilities do not have their tag cleared by CMOVE.*

**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 region or `cs1` is sealed.
For CINCOFFSET, the address is incremented by the value in `rs2` .
For CINCOFFSETIMM, the address is incremented by the immediate value `imm`.

**Prerequisites**

Zcheri_purecap

**Operation (CINCOFFSET)**

TODO

**Operation (CINCOFFSETIMM)**

TODO

### 8.1.7. CSETADDR

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

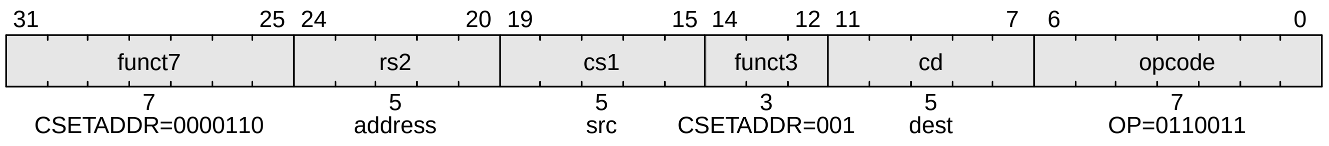
**Synopsis**

Capability set address

**Mnemonic**

`csetaddr 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, `rs1` is outside the representable range of `cs1` or if `cs1` is sealed.

**Prerequisites**

Zcheri_purecap

**Operation**

TODO

### 8.1.8. CANDPERM

**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 page has* **new** *encodings.*

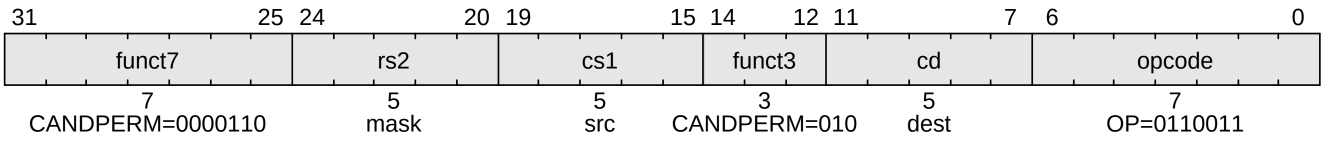
**Synopsis**

Mask capability permissions

**Mnemonics**

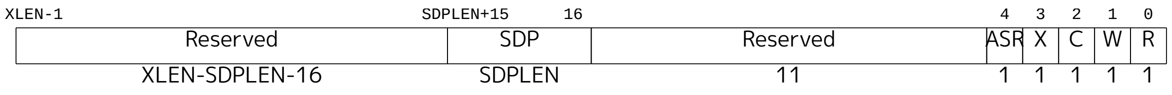
```
candperm cd, cs1, rs2
```

**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 XLENMAX=32. If permissions that cannot be encoded are indicated, CANDPERM outputs a capability with all architectural permissions cleared.*

*TODO: this may not be correct - we should work through the different combinations which are possible for removing a permission for RV32, where it is restricted, and decide what to do in each case*

**Prerequisites**

Zcheri_purecap

**Operation**

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

### 8.1.9. CSETMODE

**CHERI v9 Note:** *This instruction used to be CSetFlags.*

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

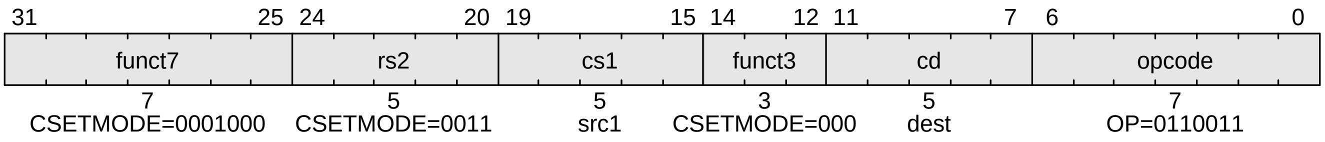
**Synopsis**

Capability set CHERI execution mode

**Mnemonic**

`csetmode 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.

**Prerequisites**

Zcheri_mode

**Operation**

```
TODO
```

### 8.1.10. CSETHIGH

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

**Synopsis**

Capability set metadata

**Mnemonic**

`csethigh cd, cs1, rs2`

**Encoding**

| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | cs1 | funct3 | cd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| CSETHIGH=0000110 | metadata | src | CSETHIGH=011 | dest | OP=0110011 |

**Description**

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

**Prerequisites**

Zcheri_purecap

**Operation**

TODO this is correct but capToMemBits is redundant, as it's now XORed with zero (null-cap)

TODO

### 8.1.11. CSETEQUALEXACT

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

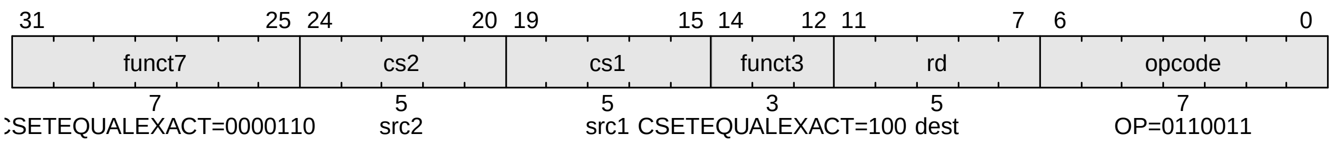
**Synopsis**

Capability equals

**Mnemonics**

`csetequalexact 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

### 8.1.12. CSEAL

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

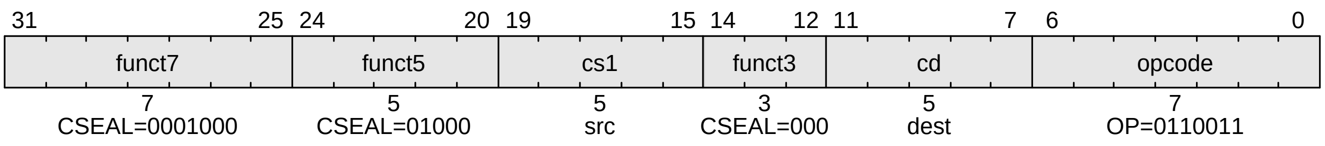
**Synopsis**

Capability seal

**Mnemonics**

`cseal cd, cs1`

**Encoding**



**Description**

Capability `cd` is written with the capability in `cs1` with its seal bit set to 1.

**Prerequisites**

Zcheri_purecap

**Operation**

TODO: The SAIL definition for CSEAL writes the OTYPE which does not exist anymore.

### 8.1.13. CTESTSUBSET

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

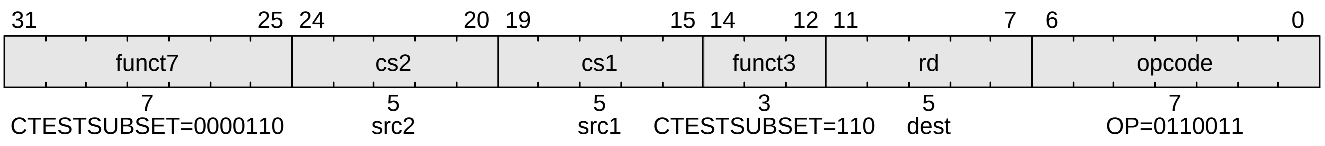
**Synopsis**

Capability test subset

**Mnemonic**

`ctestsubset 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 CBUILDCAP, although CTESTSUBSET does not include the sealed bit in the check.*

**Prerequisites**

Zcheri_purecap

**Operation**

TODO

### 8.1.14. CBUILDCAP

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

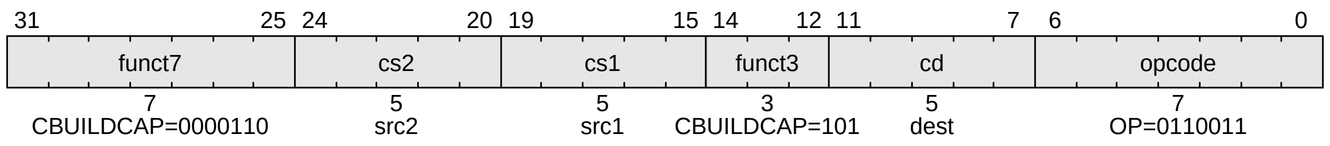
**Synopsis**

Capability build

**Mnemonic**

`cbuildcap cd, cs1, cs2`

**Encoding**



**Description**

Copy `cs2` to `cd` and set the tag to 1 if `cs1.tag` is set, `cs1` is not sealed, `cs1` 's permissions and bounds are equal or a superset of `cs2` 's, `cs2` 's bounds are not malformed (see Section 2.5), and all reserved bits in `cs2` 's metadata are 0. CBUILDCAP is typically used alongside CSETHIGH to build capabilities from integer values.

**Prerequisites**

Zcheri_purecap

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

```
let cs1_val = if unsigned(cs1) == 0 then DDC else 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 cs1 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

## 8.1.15. CGETTAG

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

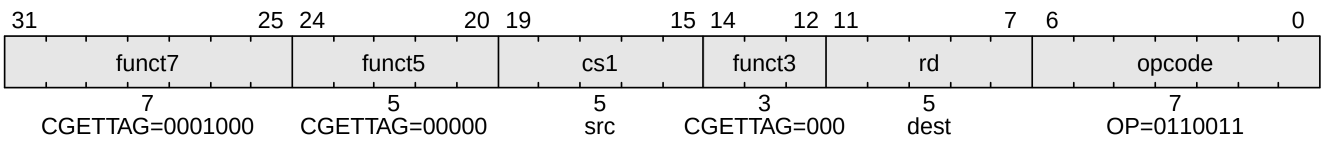
**Synopsis**

Capability get tag

**Mnemonic**

`cgettag rd, cs1`

**Encoding**



**Description**

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

**Prerequisites**

Zcheri_purecap

**Operation**

TODO

### 8.1.16. CGETPERM

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

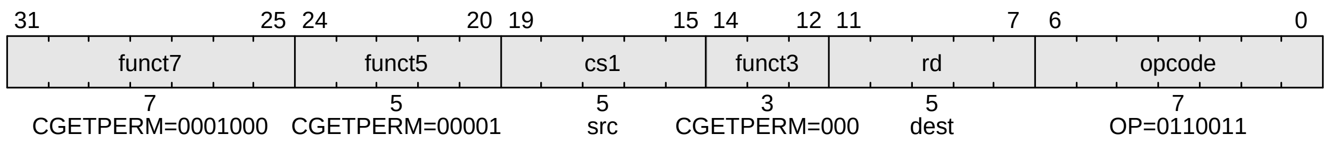
**Synopsis**

Capability get permissions

**Mnemonic**

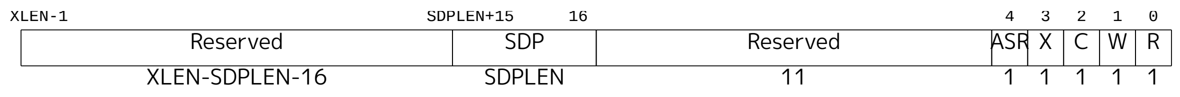
`cgetperm rd, cs1`

**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.



**Prerequisites**

Zcheri_purecap

**Operation**

TODO: The encoding of permissions changed.

## 8.1.17. CGETHIGH

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

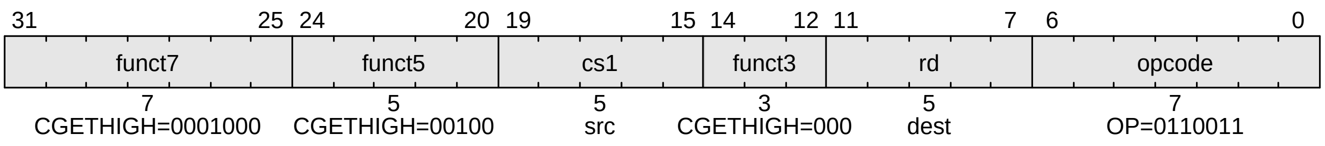
**Synopsis**

Capability get metadata

**Mnemonic**

`cgethigh rd, cs1`

**Encoding**



**Description**

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

**Prerequisites**

Zcheri_purecap

**Operation**

TODO this is correct but capToMemBits is redundant, as it’s now XORed with zero (null-cap)

TODO

### 8.1.18. CGETBASE

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

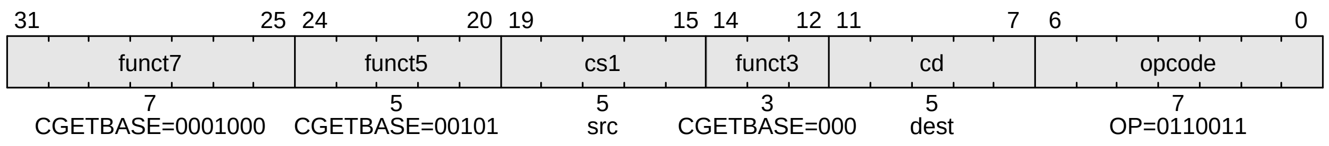
**Synopsis**

Capability get base address

**Mnemonic**

`cgetbase 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. CGETBASE outputs 0 if `cs1` 's bounds are malformed (see Section 2.5).

**Prerequisites**

Zcheri_purecap

**Operation**

TODO need to check that it returns 0 if malformed

TODO

## 8.1.19. CGETLEN

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

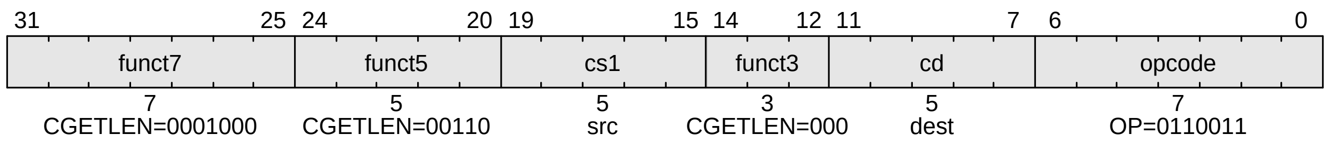
**Synopsis**

Capability get length

**Mnemonic**

`cgetlen 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. CGETLEN outputs 0 if `cs1` 's bounds are malformed (see Section 2.5).

**Prerequisites**

Zcheri_purecap

**Operation**

TODO need to check that it returns 0 if malformed

TODO

### 8.1.20. CSETBOUNDSIMM

See CSETBOUNDS.

### 8.1.21. CSETBOUNDS

CHERI v9 Note: *CSETBOUNDS was CSETBOUNDSEXACT, CSETBOUNDSIMM would have been CSETBOUNDSEXACTIMM if it had existed.*

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

CHERI v9 Note: *the immediate format has changed*

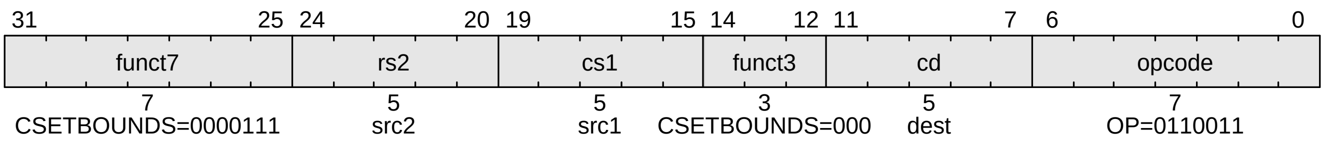
**Synopsis**

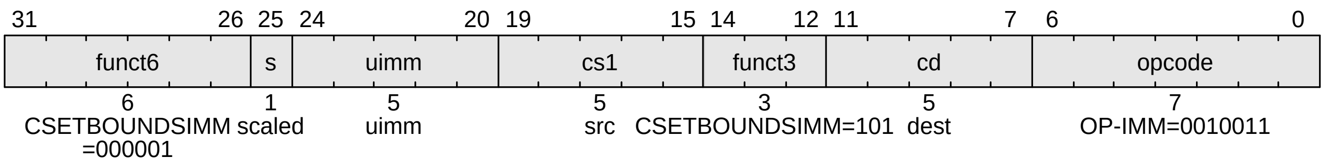
Capability set bounds

**Mnemonic**

```
csetbounds cd, cs1, rs2
csetboundsimm 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.

CSETBOUNDSIMM uses the `s` bit to scale the immediate by 4 places

```
immediate = ZeroExtend(s ? uimm<<4 : uimm)
```

**Prerequisites**

Zcheri_purecap

TODO: this is the CSetBoundsExact() function which will be renamed

**Operation for CSETBOUNDS**

TODO

**Operation for CSETBOUNDSIMM**

TODO

### 8.1.22. CSETBOUNDSINEXACT

**CHERI v9 Note:** *This instruction was called CSETBOUNDS.*

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

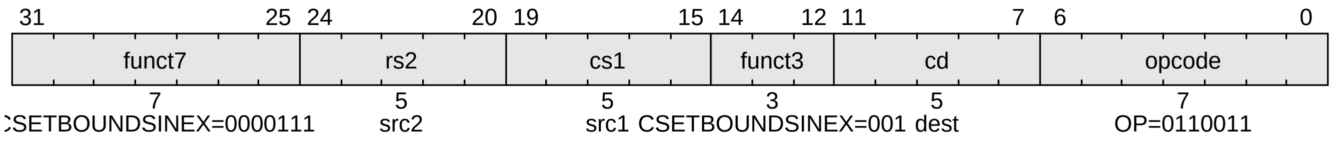
**Synopsis**

Capability set bounds, rounding up if necessary

**Mnemonic**

`csetboundsinexact 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

*TODO* *this is the CSetBounds() function which will be renamed*

**Operation for CSETBOUNDSINEXACT**

TODO

### 8.1.23. CRAM

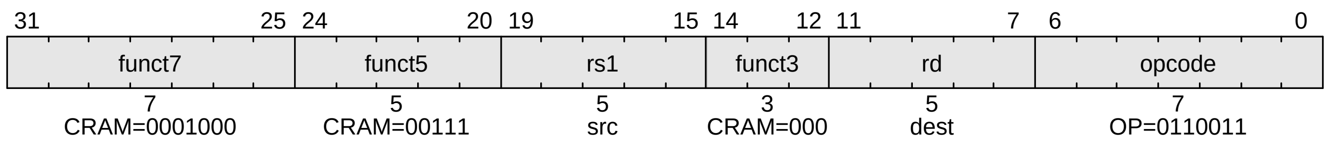
#### Synopsis

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

## 8.1.24. LC

See CLC.

## 8.1.25. CLC

**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**

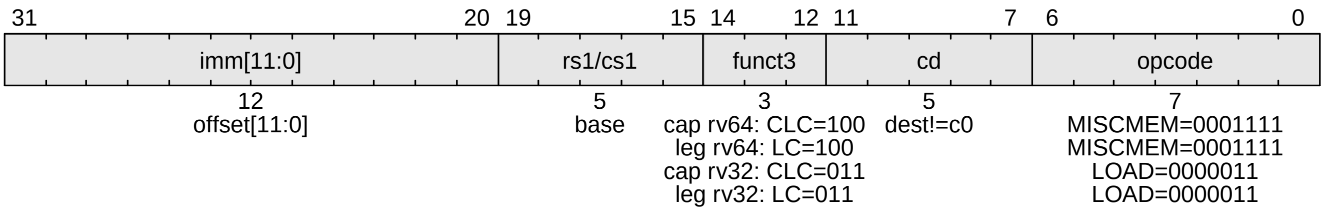
```
clc cd, offset(cs1)
```

**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 |

| CAUSE | Reason |
|---|---|
| Length violation | At least one byte accessed is outside the authority capability bounds |

### Prerequisites for CLC

Zcheri_purecap

### Prerequisites for LC

Zcheri_legacy

### CLC Operation

TODO

### LC Operation

TODO

## 8.1.26. SC

See CSC.

## 8.1.27. CSC

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

**Synopsis**

Store capability

**Capability Mode Mnemonics**

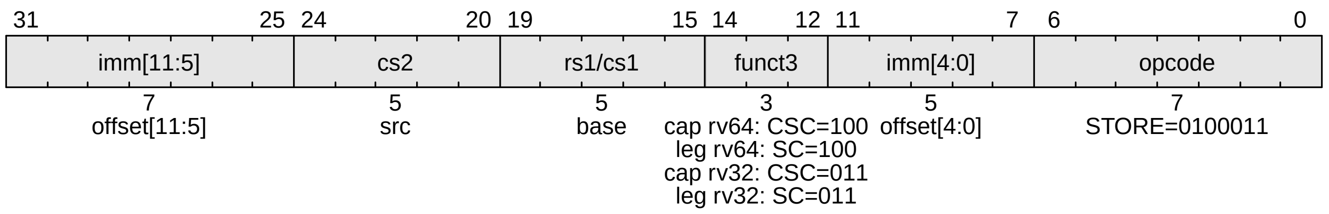
```
csc cs2, offset(cs1)
```

**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 CSC**

Zcheri_purecap

**Prerequisites for SC**

Zcheri_legacy

**CSC Operation**

TODO

**SC Operation**

TODO

## 8.2. RV32I/E and RV64I/E Base Integer Instruction Sets

## 8.2.1. AUIPC

See AUIPCC

## 8.2.2. AUIPCC

### Synopsis

Add upper immediate to **pc**/pcc

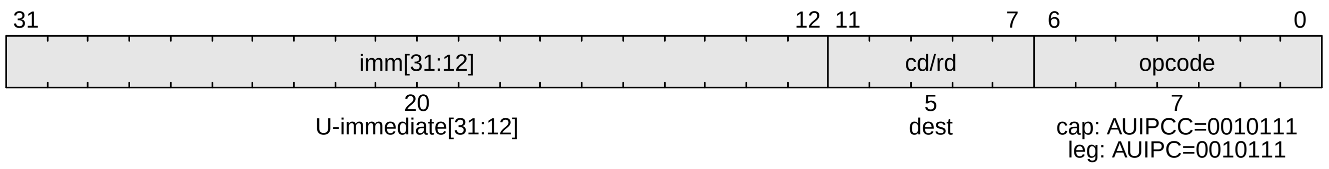
### Capability Mode Mnemonic

`auipcc 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 AUIPCC 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 region.

### 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.*

### Prerequisites for AUIPCC

Zcheri_purecap

### Prerequisites for AUIPC

Zcheri_legacy

### Operation for AUIPCC

TODO

## 8.2.3. BEQ, BNE, BLT[U], BGE[U]

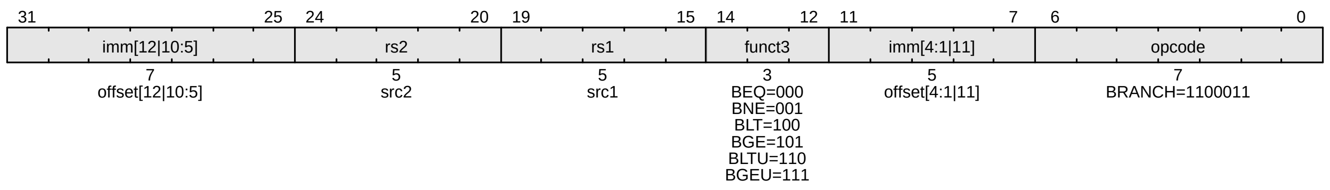
### Synopsis

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.*

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

### Operation

TODO

### 8.2.4. CJALR

See JALR

### 8.2.5. CJAL, JALR

**Synopsis**

Jump and link register

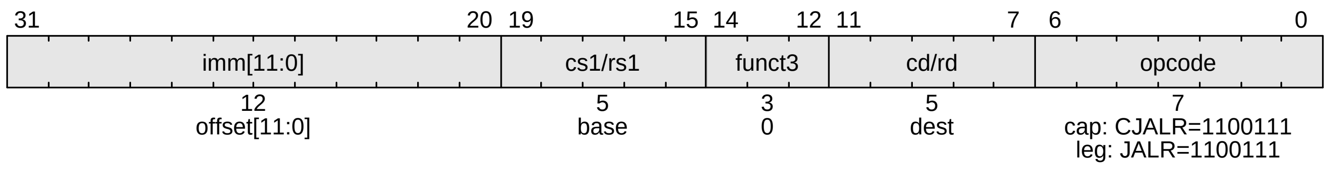
**Capability Mode Mnemonic**

`cjalr cd, cs1, offset`

**Legacy Mode Mnemonic**

`jalr rd, rs1, offset`

**Encoding**



**Capability Mode Description**

CJALR allows unconditional, indirect jumps to a target capability. The target capability is obtained by unsealing `cs1` and incrementing its address by the sign-extended 12-bit immediate, 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 | JALR | CJALR | 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 |

*The instructions on this page are either PC relative or may update the pcc. Therefore an implementation may make them illegal in debug mode.*

**Prerequisites CJALR**

Zcheri_purecap

**Prerequisites JALR**

Zcheri_legacy

**CJALR Operation**

TBD

**JALR Operation**

TBD

### 8.2.6. CJAL

See JAL

### 8.2.7. CJAL, JAL

#### Synopsis

Jump and link

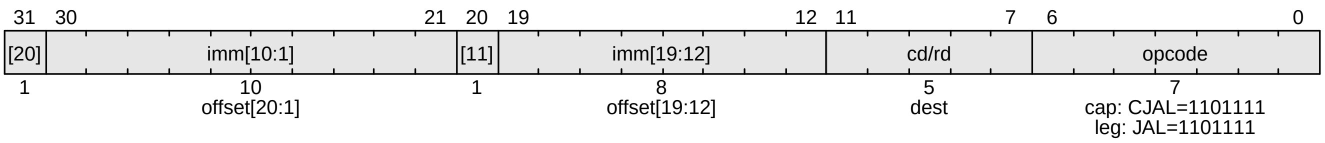
#### Capability Mode Mnemonic

`cjal cd, offset`

#### Legacy Mode Mnemonic

`jal rd, offset`

#### Encoding



#### Capability Mode Description

CJAL'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.

#### Prerequisites for CJAL

Zcheri_purecap

#### Prerequisites for JAL

Zcheri_legacy

#### CJAL Operation

TODO

#### JAL Operation TODO where's the target check?

TODO

### 8.2.8. CLWU

See CLD.

### 8.2.9. CLW

See CLD.

### 8.2.10. CLHU

See CLD.

### 8.2.11. CLH

See CLD.

### 8.2.12. CLBU

See CLD.

### 8.2.13. CLB

See CLD.

### 8.2.14. LD

See CLD.

### 8.2.15. LWU

See CLD.

### 8.2.16. LW

See CLD.

### 8.2.17. LHU

See CLD.

### 8.2.18. LH

See CLD.

### 8.2.19. LBU

See CLD.

### 8.2.20. LB

See CLD.

### 8.2.21. CLD

#### Synopsis

Load (CLD, CLW[U], CLH[U], CLB[U], LD, LW[U], LH[U], LB[U])

#### Capability Mode Mnemonics (RV64)

```
cld rd, offset(cs1)
clw[u] rd, offset(cs1)
clh[u] rd, offset(cs1)
clb[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)
```

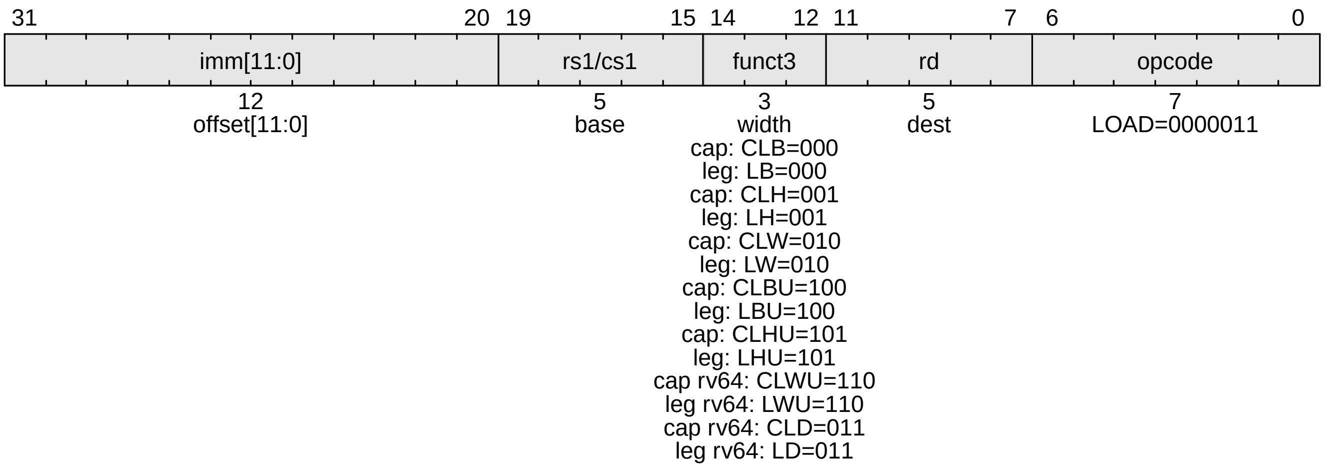
#### Capability Mode Mnemonics (RV32)

```
clw rd, offset(cs1)
clh[u] rd, offset(cs1)
clb[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 CLD

RV64, Zcheri_purecap

### Prerequisites for CLW[U], CLH[U], CLB[U]

Zcheri_purecap

### Prerequisites for LD

RV64, Zcheri_legacy

### Prerequisites for LW[U], LH[U], LB[U]

Zcheri_legacy

### Capability Mode Operation

TBD

### Legacy Mode Operation

TODO

### 8.2.22. CSW

See CSD

### 8.2.23. CSH

See CSD

### 8.2.24. CSB

See CSD

### 8.2.25. SD

See CSD

### 8.2.26. SW

See CSD

### 8.2.27. SH

See CSD

### 8.2.28. SB

See CSD

### 8.2.29. CSD

#### Synopsis

Stores (CSD, CSW, CSH, CSB, SD, SW, SH, SB)

#### Capability Mode Mnemonics (RV64)

```
csd rs2, offset(cs1)
csw rs2, offset(cs1)
csh rs2, offset(cs1)
csb rs2, offset(cs1)
```

#### Legacy Mode Mnemonics (RV64)

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

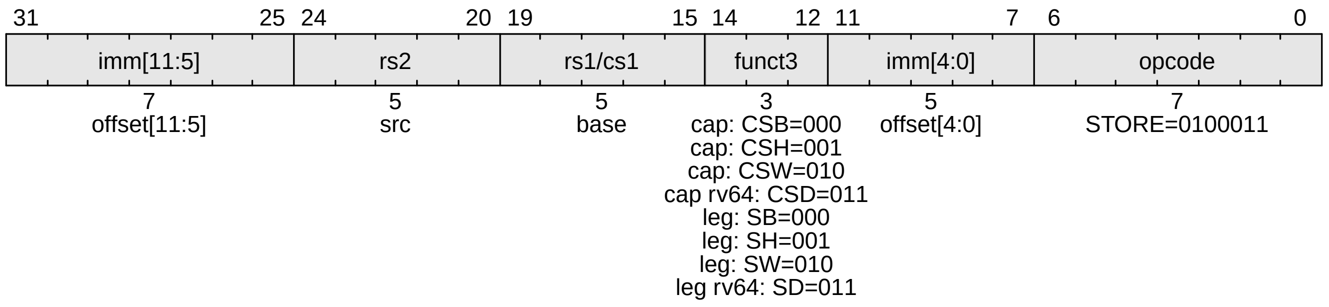
#### Capability Mode Mnemonics (RV32)

```
csw rs2, offset(cs1)
csh rs2, offset(cs1)
csb 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 CSD

RV64, Zcheri_purecap

### Prerequisites for CSW, CSH, CSB

Zcheri_purecap

### Prerequisites for SD

RV64, Zcheri_legacy

### Prerequisites for SW, SH, SB

Zcheri_legacy

### Operation

```
TBD
```

### 8.2.30. SRET

See MRET.

### 8.2.31. MRET

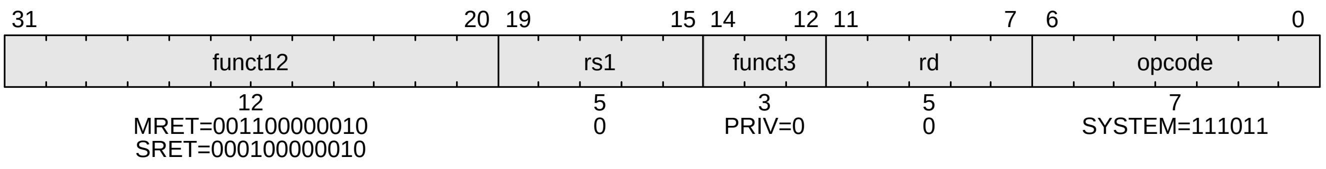
#### Synopsis

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
```

### 8.2.32. DRET

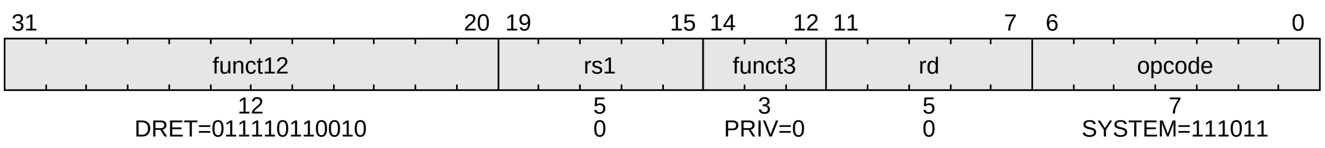
**Synopsis**

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
```

## 8.3. "A" Standard Extension for Atomic Instructions

### 8.3.1. CAMO<OP>.W

See AMO<OP>.D.

### 8.3.2. CAMO<OP>.D

See AMO<OP>.D.

### 8.3.3. AMO<OP>.W

See AMO<OP>.D.

### 8.3.4. CAMO<OP>.W

#### Synopsis

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

#### Capability Mode Mnemonics (RV64)

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

#### Capability Mode Mnemonics (RV32)

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

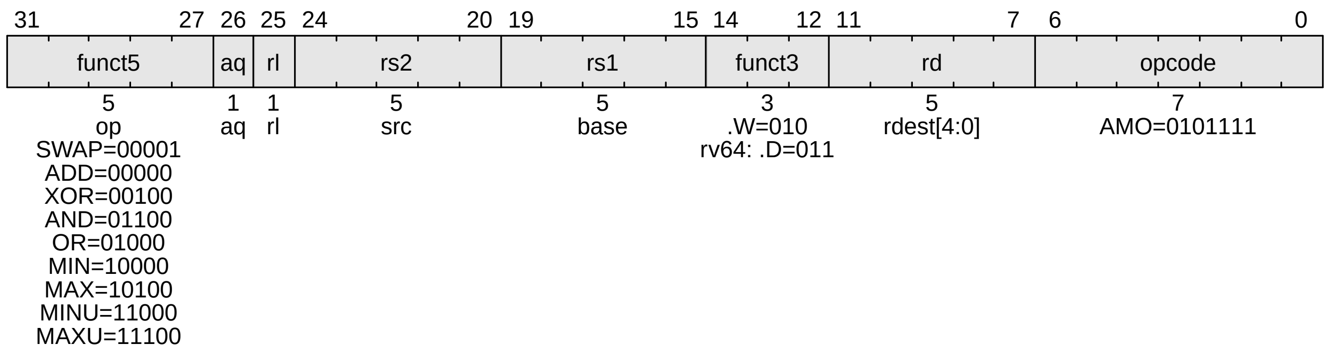
#### Legacy Mode Mnemonics (RV64)

```
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 |

| CAUSE | Reason |
|---|---|
| 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 CAMO<OP>.W, CAMO<OP>.D

Zcheri_purecap

### Prerequisites for AMO<OP>.W, AMO<OP>.D

Zcheri_legacy

### Capability Mode Operation

```
TBD
```

### Legacy Mode Operation

TODO

### 8.3.5. AMOSWAP.C

See CAMOSWAP.C.

### 8.3.6. CAMOSWAP.C

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

**Synopsis**

Atomic Operations (CAMOSWAP.C, AMOSWAP.C), 32-bit encodings

*These instructions have different encodings for RV64 and RV32.*

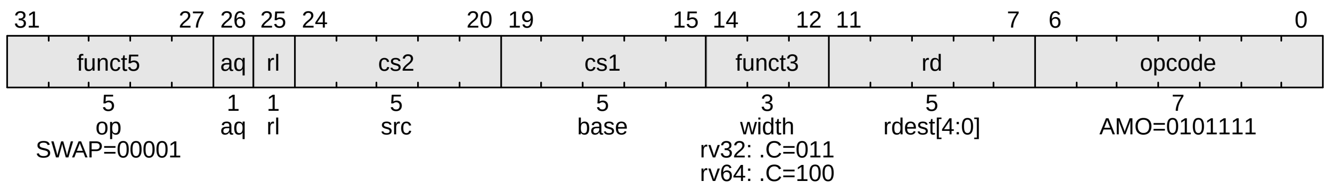
**Capability Mode Mnemonics**

`camoswap.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 |
| 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 CAMOSWAP.C

Zcheri_purecap

### Prerequisites for AMOSWAP.C

Zcheri_legacy

### Operation

TODO

### 8.3.7. CLR.D

See LR.B.

### 8.3.8. CLR.W

See LR.B.

### 8.3.9. CLR.H

See LR.B.

### 8.3.10. CLR.B

See LR.B.

### 8.3.11. LR.D

See LR.B.

### 8.3.12. LR.W

See LR.B.

### 8.3.13. LR.H

See LR.B.

### 8.3.14. LR.B

Synopsis

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

Capability Mode Mnemonics (RV64)

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

Capability Mode Mnemonics (RV32)

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

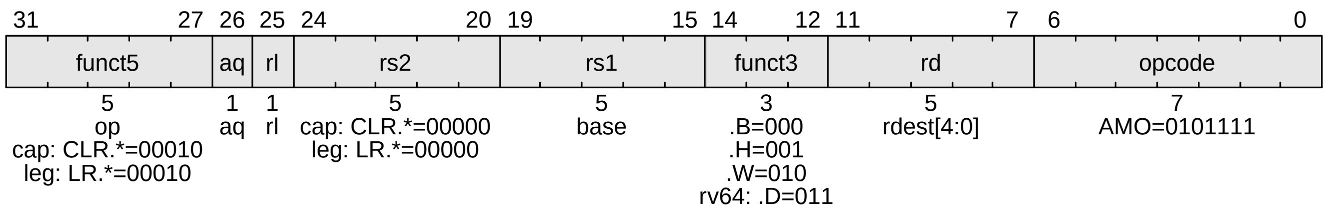
Legacy Mode Mnemonics (RV64)

```
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 CLR.D

RV64, and Zcheri_purecap

Prerequisites for CLR.W

Zcheri_purecap

**Prerequisites for CLR.H, CLR.B**
  Zbhlrsc and Zcheri_purecap

**Prerequisites for LR.D**
  RV64, and Zcheri_legacy

**Prerequisites for LR.W**
  Zcheri_legacy

**Prerequisites for LR.H, LR.B**
  Zbhlrsc and Zcheri_legacy

**Operation**

```
TBD
```

## 8.3.15. LR.C

See CLR.C.

## 8.3.16. CLR.C

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

Synopsis

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

*These instructions have different encodings for RV64 and RV32.*

Capability Mode Mnemonics (RV64)

```
clr.c cd, 0(cs1)
```

Capability Mode Mnemonics (RV32)

```
clr.c cd, 0(cs1)
```

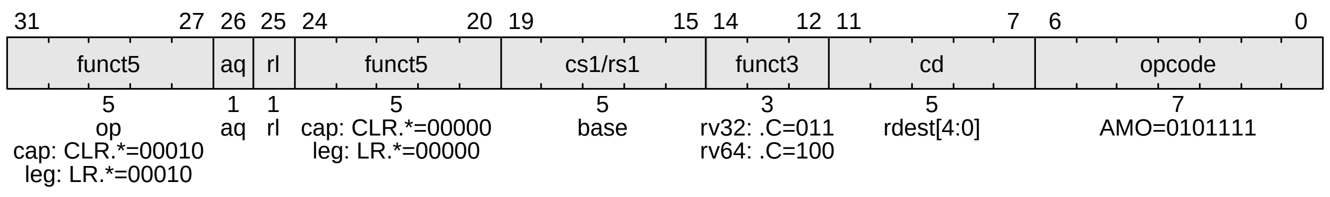
Legacy Mode Mnemonics (RV64)

```
lr.c cd, 0(rs1)
```

Legacy Mode Mnemonics (RV32)

```
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 |

| CAUSE | Reason |
|---|---|
| Length violation | At least one byte accessed is outside the authority capability bounds |

### Prerequisites for CLC

Zcheri_purecap

### Prerequisites for LC

Zcheri_legacy

### Operation

```
TBD
```

### 8.3.17. CSC.D

See SC.B.

### 8.3.18. CSC.W

See SC.B.

### 8.3.19. CSC.H

See SC.B.

### 8.3.20. CSC.B

See SC.B.

### 8.3.21. SC.D

See SC.B.

### 8.3.22. SC.W

See SC.B.

### 8.3.23. SC.H

See SC.B.

### 8.3.24. SC.B

#### Synopsis

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

#### Capability Mode Mnemonics (RV64)

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

#### Capability Mode Mnemonics (RV32)

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

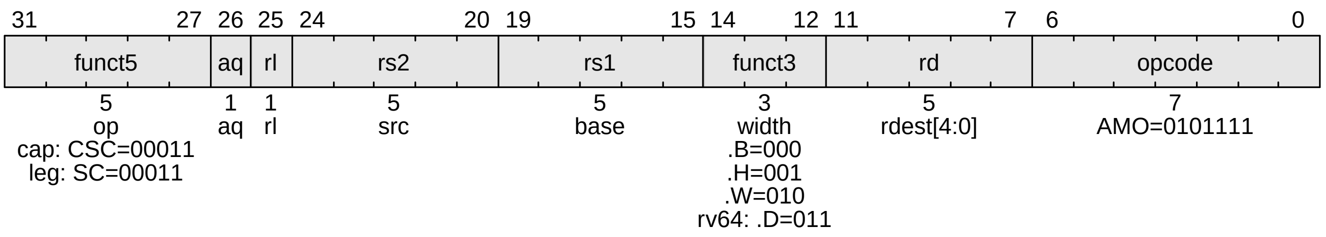
#### Legacy Mode Mnemonics (RV64)

```
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 CSC.D

RV64, and Zcheri_purecap

#### Prerequisites for CSC.W

Zcheri_purecap

**Prerequisites for CSC.H, CSC.B**

Zcheri_purecap, and Zbhlrsc

**Prerequisites for SC.D**

RV64, and Zcheri_legacy

**Prerequisites for SC.W**

Zcheri_legacy

**Prerequisites for SC.H, SC.B**

Zcheri_legacy, and Zbhlrsc

**Operation**

```
TBD
```

### 8.3.25. SC.C

See CSC.C.

### 8.3.26. CSC.C

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

#### Synopsis

Store Conditional (CSC.C, SC.C), 32-bit encodings

*These instructions have different encodings for RV64 and RV32.*

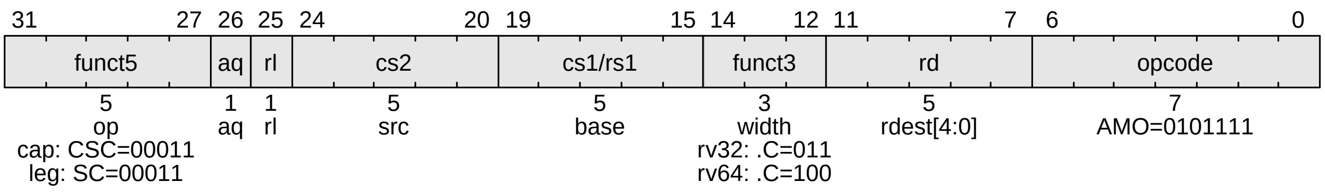
#### Capability Mode Mnemonics

```
csc.c cd, cs2, 0(cs1)
```

#### Legacy Mode Mnemonics

```
sc.c cd, 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 CSC.C

Zcheri_purecap

#### Prerequisites for SC.C

Zcheri_legacy

#### Operation

```
TBD
```

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

### 8.4.1. CSRRW

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

**Synopsis**

CSR access (CSRRW) 32-bit encodings

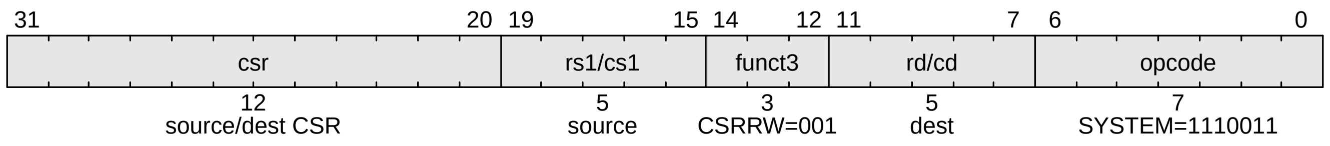
**Mnemonic (XLEN-wide target, and XLEN-wide aliases of CLEN-wide CSRs)**

`csrrw rd, rs1, csr`

**Mnemonics (CLEN-wide target)**

`csrrw cd, cs1, csr`

**Encoding**



**Description**

This is a standard RISC-V CSR instructions with extended functionality for accessing CLEN-wide CSRs, such as mtvec/mtvecc which can be accessed through either the RISC-V address or the capability address alias.

See Table 38 for a list of CLEN-wide CSRs and Table 39 for the action taken on writing each one.

CSRRW writes `cs1` to the CLEN-wide alias of extended CSRs, and reads a full capability into `cd`.

CSRRW writes `rs1` to the XLEN-wide alias of extended CSRs, and reads the address field into `rd`.

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

**Permissions**

All non-user mode accessible CSRs require ASR-permission, including existing RISC-V CSRs.

**Prerequisites for capability address aliases**

Zcheri_purecap

**Prerequisites for legacy address aliases**

Zcheri_legacy

**Operation**

```
TBD
```

### 8.4.2. CSRRWI

See CSRRCI.

### 8.4.3. CSRRS

See CSRRCI.

### 8.4.4. CSRRSI

See CSRRCI.

### 8.4.5. CSRRC

See CSRRCI.

### 8.4.6. CSRRCI

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

Synopsis

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

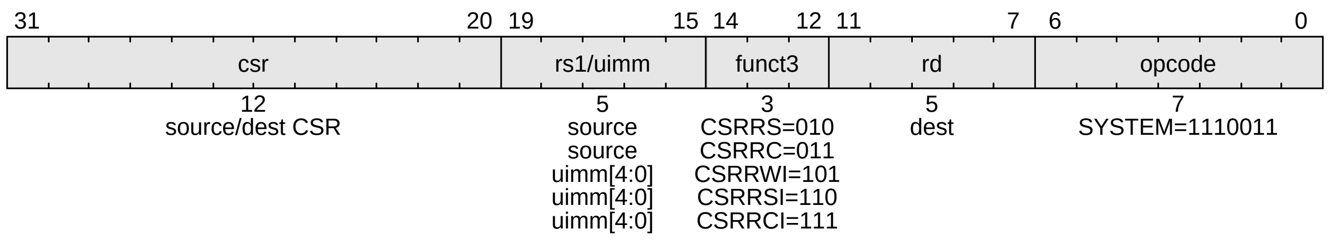
Register Source Mnemonics

```
csrr[s|c] rd, rs1, csr
```

Immediate Source Mnemonics

```
csrr[w|s|c]i rd, imm, csr
```

Encoding



Description

These are standard RISC-V CSR instructions with extended functionality for accessing CLEN-wide CSRs, such as mtvec/mtvecc which can be accessed through either the RISC-V address or the capability address alias.

Unlike CSRRW these instruction perform the same update to CLEN-wide CSRs to either the XLEN or CLEN-wide alias as they only every perform an XLEN-wide update. Where a CLEN-wide CSR is updated, through either alias, the final address is determined as defined by RISC-V for these instructions. The metadata and tag are updated as defined in Table 39.

See Table 38 for a list of CLEN-wide CSRs and Table 39 for the action taken on writing an XLEN-wide value to each one.

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

Permissions

All non-user mode accessible CSRs require ASR-permission, including existing RISC-V CSRs.

Prerequisites for capability address aliases

Zcheri_purecap

Prerequisites for legacy address aliases

Zcheri_legacy

Operation

```
TBD
```

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

### 8.5.1. CFLD

See FLH.

### 8.5.2. CFLW

See FLH.

### 8.5.3. CFLH

See FLH.

### 8.5.4. FLD

See FLH.

### 8.5.5. FLW

See FLH.

### 8.5.6. FLH

**Synopsis**

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

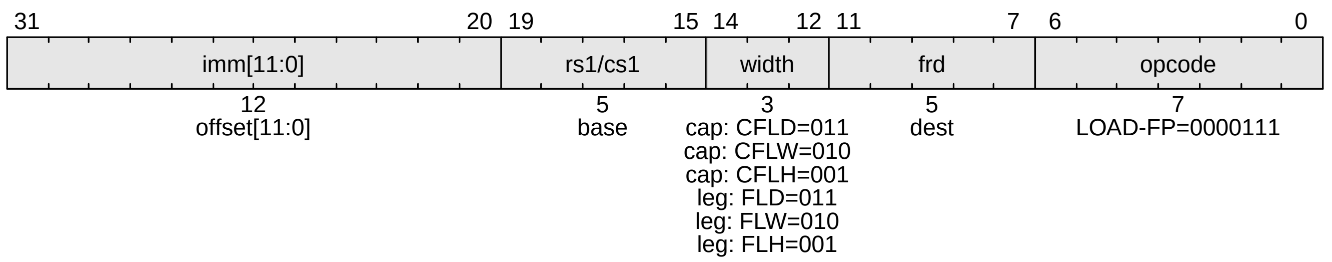
**Capability Mode Mnemonics**

`cfld/cflw/cflh 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 CFLD**

Zcheri_purecap, and D

**Prerequisites for CFLW**

Zcheri_purecap, and F

**Prerequisites for CFLH**

Zcheri_purecap, and Zfhmin or Zfh

**Prerequisites for FLD**

Zcheri_legacy, and D

**Prerequisites for FLW**

Zcheri_legacy, and F

**Prerequisites for FLH**

Zcheri_legacy, and Zfhmin or Zfh

**Operation**

TODO

### 8.5.7. CFSD

See FLH.

### 8.5.8. CFSW

See FLH.

### 8.5.9. CFSH

See FSH.

### 8.5.10. FSD

See FSH.

### 8.5.11. FSW

See FSH.

### 8.5.12. FSH

#### Synopsis

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

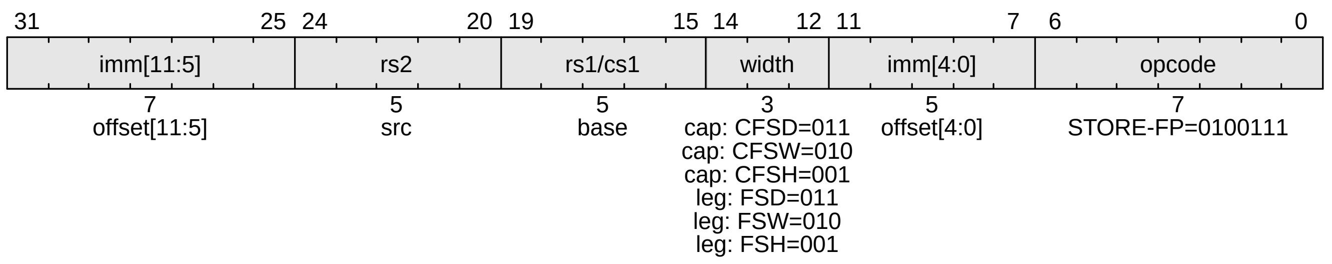
#### Capability Mode Mnemonics

```
cfsd/cfsw/cfsh 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 CFSD

Zcheri_purecap, and D

#### Prerequisites for CFSW

Zcheri_purecap, and F

#### Prerequisites for CFSH

Zcheri_purecap, and Zfh or Zfhmin

#### Prerequisites for FSD

Zcheri_legacy, and D

#### Prerequisites for FSW

Zcheri_legacy, and F

### Prerequisites for FSH

Zcheri_legacy, and Zfh or Zfhmin

### Operation

```
TBD
```

## 8.6. "C" Standard Extension for Compressed Instructions

### 8.6.1. 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**

| 15–13 | 12–10 | 9–7 | 6–2 | 1–0 |
|---|---|---|---|---|
| funct3 | imm | rs1' | imm | op |
| 3 | 3 | 3 | 5 | 2 |
| C.BEQZ | offset[8\|4:3] | src | offset[7:6\|2:1\|5] | C1 |
| C.BNEZ | offset[8\|4:3] | src | offset[7:6\|2:1\|5] | C1 |

**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.*

**Prerequisites**

C or Zca

**Operation (after expansion to 32-bit encodings)**

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

### 8.6.2. C.MV

See C.CMOVE.

### 8.6.3. C.CMOVE

**Synopsis**

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

**Capability Mode Mnemonic**

c.cmove cd, cs2`

**Capability Mode Expansion**

cmove cd, cs2`

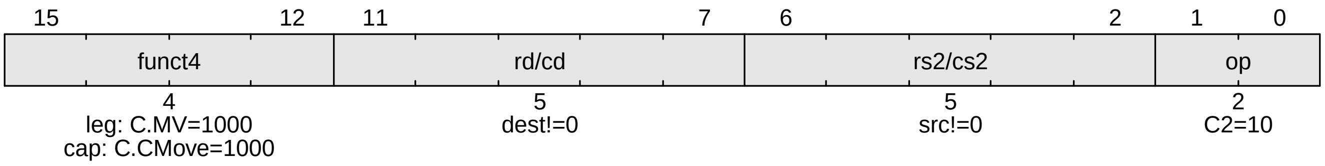
**Legacy Mode Mnemonic**

c.mv rd, rs2`

**Legacy Mode Expansion**

add rd, x0, rs2`

**Encoding**



**Capability Mode Description**

Capability register cd is replaced with the contents of cs1.

**Legacy Mode Description**

Standard RISC-V C.MV instruction.

**Prerequisites C.CMOVE**

C or Zca, Zcheri_purecap

**Prerequisites C.MV**

C or Zca, Zcheri_legacy

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

See CMOVE

### 8.6.4. C.ADDI16SP

See C.CINCOFFSET16CSP.

### 8.6.5. C.CINCOFFSET16CSP

#### Synopsis

Stack pointer increment in blocks of 16 (C.CINCOFFSET16CSP, C.ADDI16SP), 16-bit encodings

#### Capability Mode Mnemonic

```
c.cincoffset16csp imm
```

#### Capability Mode Expansion

```
cincoffset csp, csp, imm
```

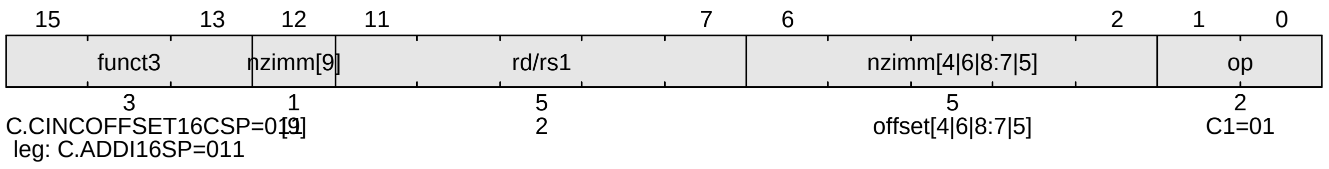
#### Legacy Mode Mnemonic

```
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 C.CINCOFFSET16CSP

C or Zca, Zcheri_purecap

#### Prerequisites for C.ADDI16SP

C or Zca, Zcheri_legacy

#### Capability Mode Operation

TODO

### 8.6.6. C.ADDI4SPN

See C.CINCOFFSET4CSPN.

### 8.6.7. C.CINCOFFSET4CSPN

**Synopsis**

Stack pointer increment in blocks of 4 (C.CINCOFFSET4CSPN, C.ADDI4SPN), 16-bit encodings

**Capability Mode Mnemonic**

`c.cincoffset4cspn rd', uimm`

**Capability Mode Expansion**

`cincoffset rd', csp, uimm`

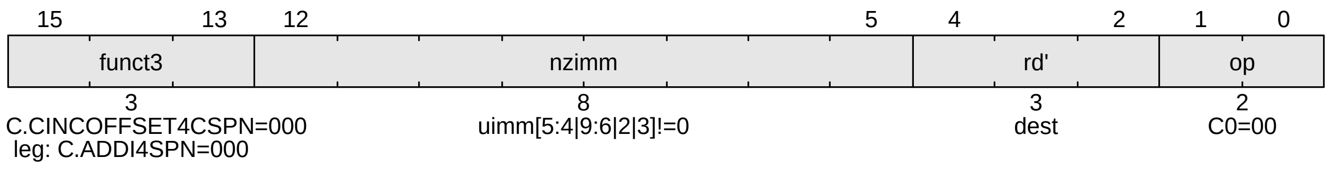
**Legacy Mode Mnemonic**

`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 `rd'`. 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.CINCOFFSET4CSPN**

C or Zca, Zcheri_purecap

**Prerequisites for C.ADDI4SPN**

C or Zca, Zcheri_legacy

**Capability Mode Operation**

TODO

## 8.6.8. C.CMODESWITCH

**CHERI v9 Note:** *This instruction is* **new**.

**Synopsis**

Capability/Legacy Mode switching (C.CMODESWITCH), 16-bit encodings

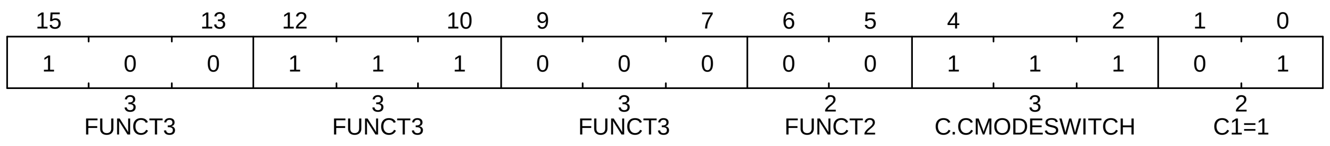
**Mnemonics**

`c.cmodeswitch`

**Expansions**

`cmodeswitch`

**Encoding**



**Capability Mode Description**

Directly switch to Legacy Mode.

**Legacy Mode Description**

Directly switch to Capability Mode.

**Exceptions**

None

*The instructions on this page are either PC relative or may update the pcc. Therefore an implementation may make them illegal in debug mode.*

**Prerequisites**

C or Zca, Zcheri_mode

**Operation (after expansion to 32-bit encodings)**

See CMODESWITCH

### 8.6.9. C.JALR

See C.CJALR.

### 8.6.10. C.CJALR

**Synopsis**

Register based jumps with link, 16-bit encodings

**Capability Mode Mnemonic**

`c.cjalr c1, cs1`

**Capability Mode Expansion**

`cjalr c1, 0(cs1)`

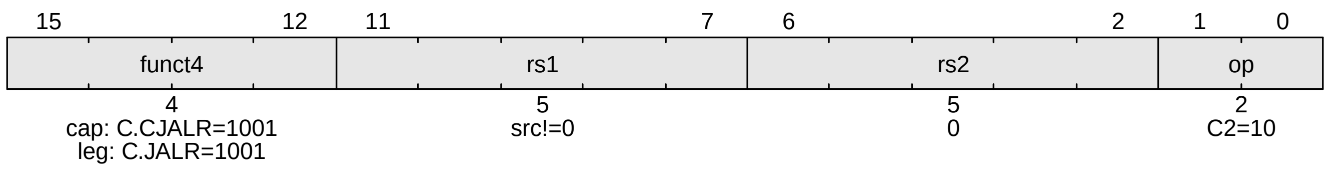
**Legacy Mode Mnemonic**

`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.

**Prerequisites C.CJALR**

C or Zca, Zcheri_purecap

**Prerequisites C.JALR**

C or Zca, Zcheri_legacy

**Operation (after expansion to 32-bit encodings)**

See CJALR, JALR

### 8.6.11. C.CJR

See C.JR.

### 8.6.12. C.JR

#### Synopsis

Register based jumps without link, 16-bit encodings

#### Capability Mode Mnemonic

```
c.cjr cs1
```

#### Capability Mode Expansion

```
cjalr c0, 0(cs1)
```

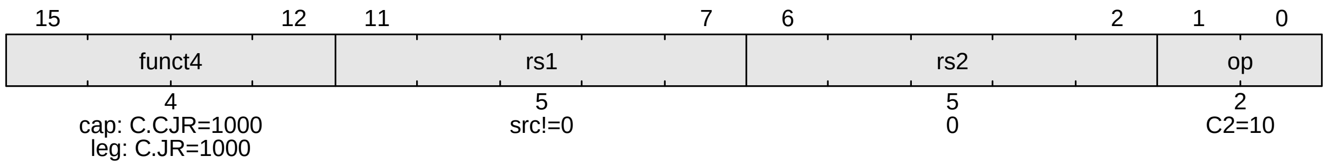
#### Legacy Mode Mnemonic

```
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 CJALR, 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.*

#### Prerequisites for C.CJALR

C or Zca, Zcheri_purecap

#### Prerequisites for C.JALR

C or Zca, Zcheri_legacy

#### Operation (after expansion to 32-bit encodings)

See CJALR, JALR

### 8.6.13. C.JAL

See C.CJAL.

### 8.6.14. C.CJAL

#### Synopsis

Register based jumps with link, 16-bit encodings

#### Capability Mode Mnemonic (RV32)

```
c.cjal c1, offset
```

#### Capability Mode Expansion (RV32)

```
cjal c1, offset
```

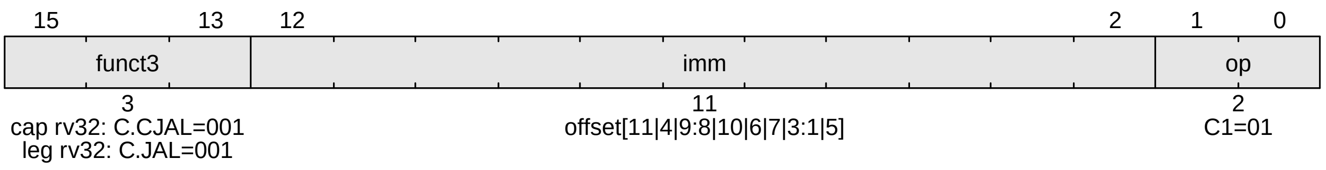
#### Legacy Mode Mnemonic (RV32)

```
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.

#### Prerequisites for C.CJAL

C or Zca, Zcheri_purecap

#### Prerequisites for C.JAL

C or Zca, Zcheri_legacy

#### Operation (after expansion to 32-bit encodings)

See CJAL, JAL

### 8.6.15. C.J

See C.CJ.

### 8.6.16. C.CJ

**Synopsis**

Register based jumps without link, 16-bit encodings

**Capability Mode Mnemonic**

`c.cj offset`

**Capability Mode Expansion**

`cjal c0, offset`

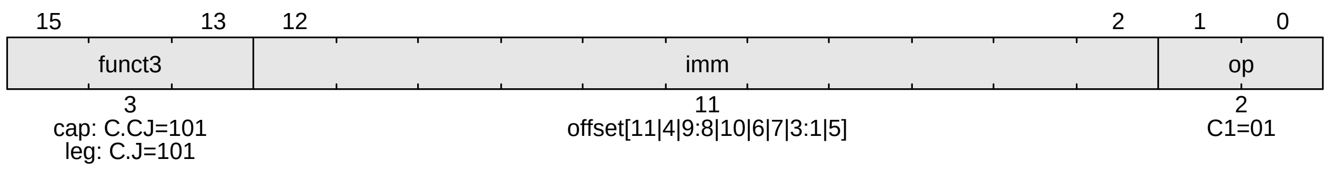
**Legacy Mode Mnemonic**

`c.j 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**

CHERI Length Violation

**Prerequisites for C.CJ**

C or Zca, Zcheri_purecap

**Prerequisites for C.J**

C or Zca, Zcheri_legacy

**Operation (after expansion to 32-bit encodings)**

See CJAL, JAL

### 8.6.17. C.CLD

See C.LW.

### 8.6.18. C.CLW

See C.LW.

### 8.6.19. C.LD

See C.LW.

### 8.6.20. C.LW

**Synopsis**

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

**Capability Mode Mnemonics (RV64)**

```
c.cld/c.clw rd', offset(cs1')
```

**Capability Mode Expansions (RV64)**

```
cld/clw 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.clw rd', offset(cs1')
```

**Capability Mode Expansions (RV32)**

```
clw rd', offset(cs1')
```

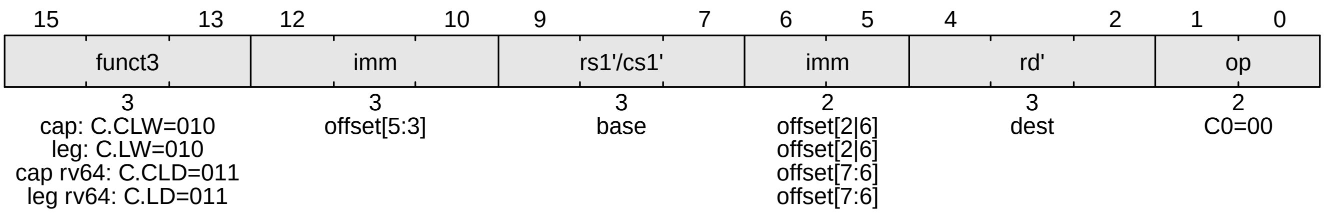
**Legacy Mode Mnemonics (RV32)**

```
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 C.CLD

RV64, and C or Zca, Zcheri_purecap

### Prerequisites C.CLW

C or Zca, Zcheri_purecap

### Prerequisites C.LD

RV64, C or Zca, Zcheri_legacy

### Prerequisites C.LW

C or Zca, Zcheri_legacy

### Operation (after expansion to 32-bit encodings)

See CLD, CLW, LD, LW

### 8.6.21. C.CLWSP

See C.LDSP.

### 8.6.22. C.CLDSP

See C.LDSP.

### 8.6.23. C.LWSP

See C.LDSP.

### 8.6.24. C.LDSP

Synopsis

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

Capability Mode Mnemonics (RV64)

```
c.cld/c.clw rd, offset(csp)
```

Capability Mode Expansions (RV64)

```
cld/clw 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.clw rd, offset(csp)
```

Capability Mode Expansions (RV32)

```
clw rd, offset(csp)
```

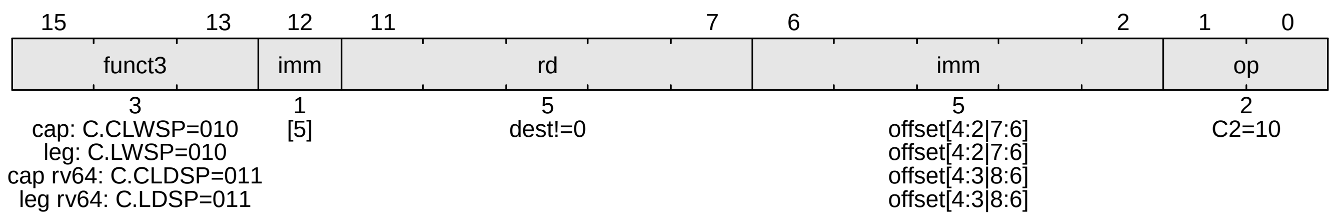
Legacy Mode Mnemonics (RV32)

```
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 C.CLDSP

RV64, and C or Zca, Zcheri_purecap

### Prerequisites for C.CLWSP

C or Zca, Zcheri_purecap

### Prerequisites for C.LDSP

RV64, and C or Zca, Zcheri_legacy

### Prerequisites for C.LWSP

C or Zca, Zcheri_legacy

### Operation (after expansion to 32-bit encodings)

See CLW, CLD, LW, LD

### 8.6.25. C.FLW

See C.FLWSP.

### 8.6.26. C.FLWSP

#### Synopsis

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

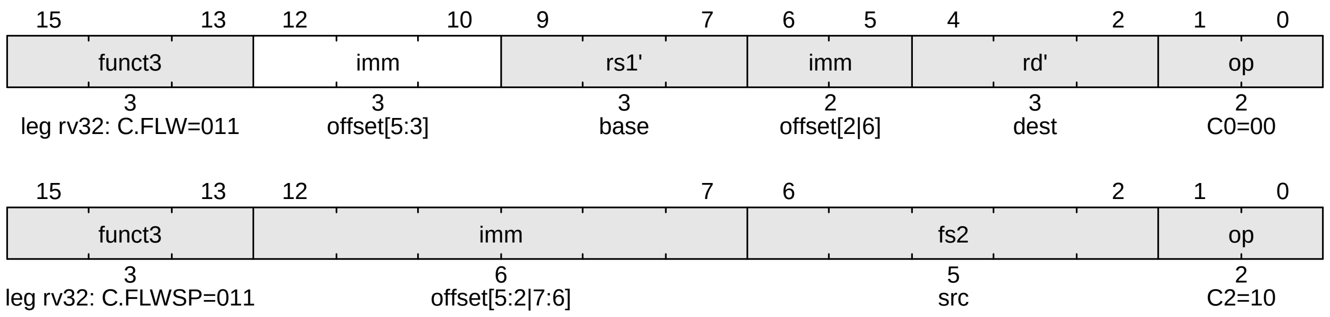
#### Legacy Mode Mnemonics (RV32)

```
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.CLC, C.CLCSP.

#### 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_legacy, and F

#### Operation (after expansion to 32-bit encodings)

See FLW

### 8.6.27. C.CFLD

See C.FLDSP.

### 8.6.28. C.FLD

See C.FLDSP.

### 8.6.29. C.CFLDSP

See C.FLDSP.

### 8.6.30. C.FLDSP

#### Synopsis

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

#### Capability Mode Mnemonics (RV32)

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

#### Capability Mode Expansions (RV32)

```
cfld frd', offset(csp)
```

#### Legacy Mode Mnemonics (RV32)

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

#### Legacy Mode Expansions (RV32)

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

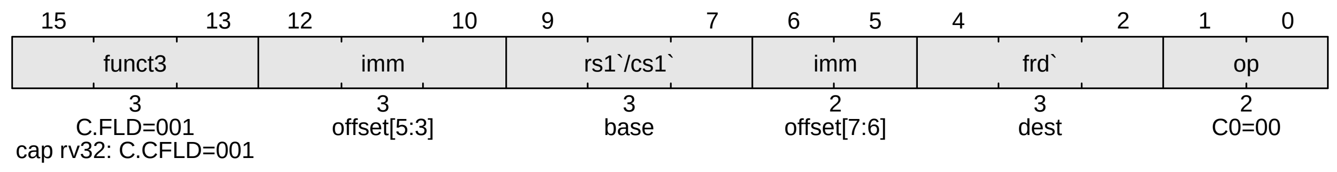
#### Legacy Mode Mnemonics (RV64)

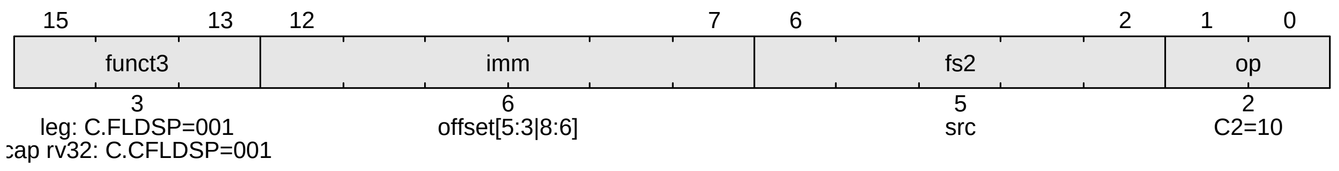
```
c.fld fs2, offset(rs1'/sp)
```

#### Legacy Mode Expansion (RV64)

```
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.CLC, C.CLCSP.

#### 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 C.CFLD, C.CFLDSP**

C or Zca, Zcheri_purecap, and D

**Prerequisites for C.FLD, C.FLDSP**

C or Zca, Zcheri_legacy, and D

**Operation (after expansion to 32-bit encodings)**

See FLD

### 8.6.31. C.CLC

see C.CLCSP.

### 8.6.32. C.CLCSP

#### Synopsis

Capability loads (C.CLC, C.CLCSP), 16-bit encodings

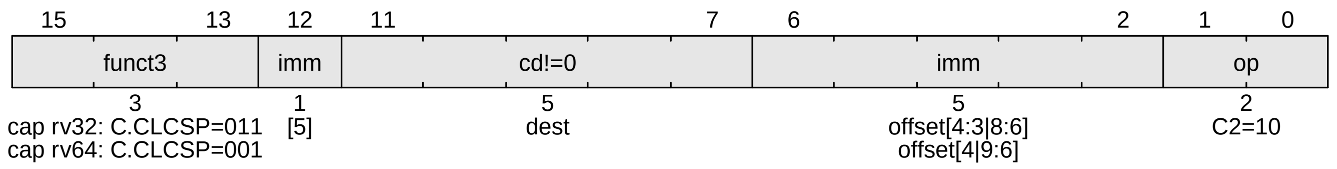
#### Capability Mode Mnemonics

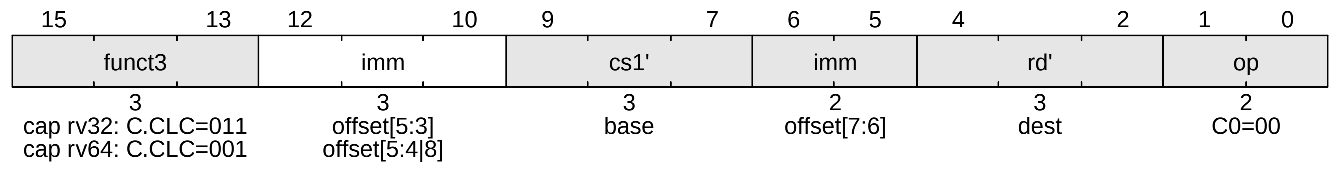
```
c.clc cd', offset(cs1'/csp)
```

#### Capability Mode Expansions

```
clc 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 CLC

### 8.6.33. C.CSD

See C.SW.

### 8.6.34. C.CSW

See C.SW.

### 8.6.35. C.SD

See C.SW.

### 8.6.36. C.SW

Synopsis

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

Capability Mode Mnemonics (RV64)

```
c.csd/c.csw rs2', offset(cs1')
```

Capability Mode Expansions (RV64)

```
csd/csw 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.csw rs2', offset(cs1')
```

Capability Mode Expansion (RV32)

```
csw rs2', offset(cs1')
```

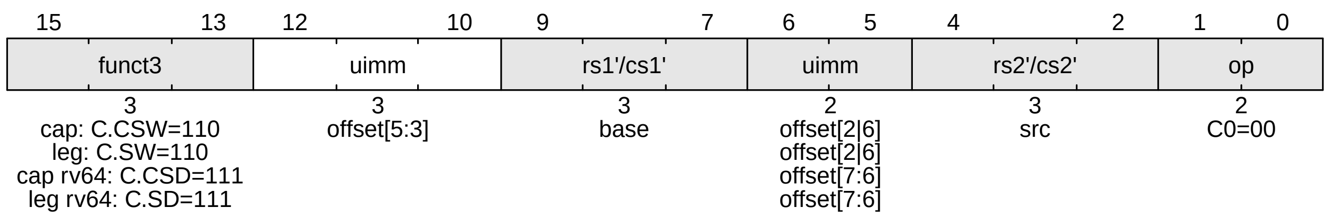
Legacy Mode Mnemonics (RV32)

```
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 C.CSD

RV64, and C or Zca, Zcheri_purecap

### Prerequisites for C.CSW

C or Zca, Zcheri_purecap

### Prerequisites for C.SD

RV64, and C or Zca, Zcheri_legacy

### Prerequisites for C.SW

C or Zca, Zcheri_legacy

### Operation (after expansion to 32-bit encodings)

See CSD, CSW, SD, SW

### 8.6.37. C.CSWSP

See C.SDSP.

### 8.6.38. C.CSDSP

See C.SDSP.

### 8.6.39. C.SWSP

See C.SDSP.

### 8.6.40. C.SDSP

Synopsis

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

Capability Mode Mnemonics (RV64)

```
c.csw/c.csd rs2, offset(csp)
```

Capability Mode Expansions (RV64)

```
csd/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.csw rs2, offset(csp)
```

Capability Mode Expansion (RV32)

```
csw rs2, offset(csp)
```

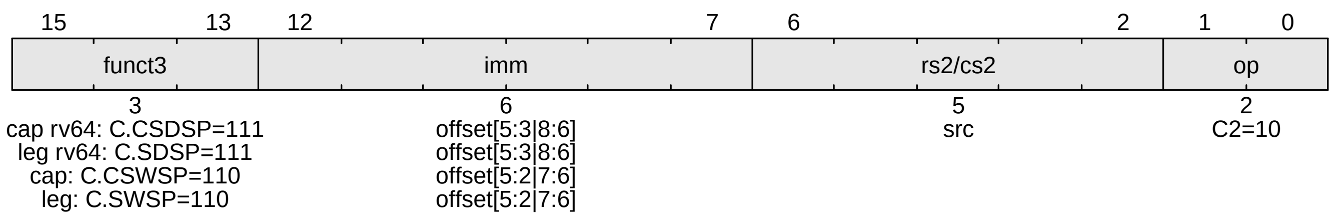
Legacy Mode Mnemonics (RV32)

```
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 C.CSDSP

RV64, and C or Zca, Zcheri_purecap

### Prerequisites for C.CSWSP

C or Zca, Zcheri_purecap

### Prerequisites for C.SDSP

RV64, and C or Zca, Zcheri_purecap

### Prerequisites for C.SWSP

C or Zca, Zcheri_purecap

### Operation (after expansion to 32-bit encodings)

See CSD, CSW, SD, SW

### 8.6.41. C.FSW

See C.FSWSP.

### 8.6.42. C.FSWSP

#### Synopsis

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

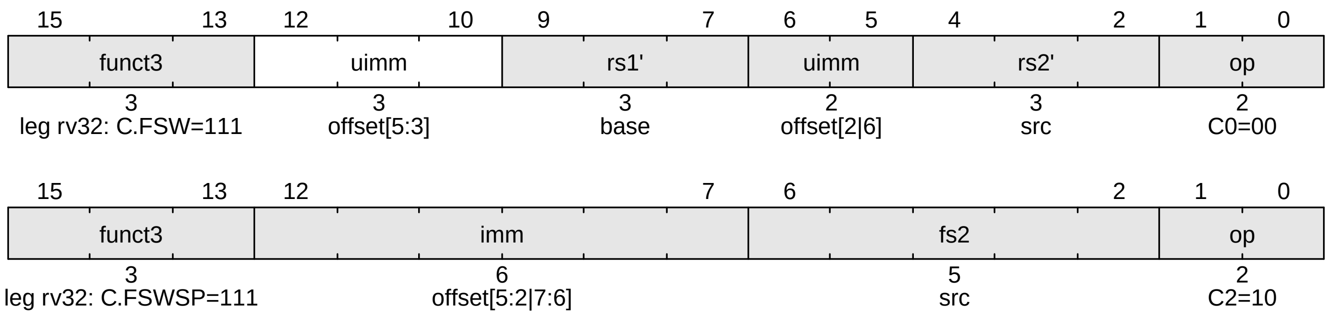
#### Legacy Mode Mnemonics (RV32)

```
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.CSC, C.CSCSP.*

#### 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 C.FSW, C.FSWSP

C or Zca, Zcheri_legacy

#### Operation (after expansion to 32-bit encodings)

See FSW

### 8.6.43. C.CFSD

See C.FSDSP.

### 8.6.44. C.CFSDSP

See C.FSDSP.

### 8.6.45. C.FSD

See C.FSDSP.

### 8.6.46. C.FSDSP

#### Synopsis

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

#### Capability Mode Mnemonics (RV32CD/RV32D_Zca)

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

#### Capability Mode Expansions (RV32)

```
cfsd fs2, offset(csp)
```

#### Legacy Mode Mnemonics (RV32CD/RV32D_Zca)

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

#### Legacy Mode Expansions (RV32)

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

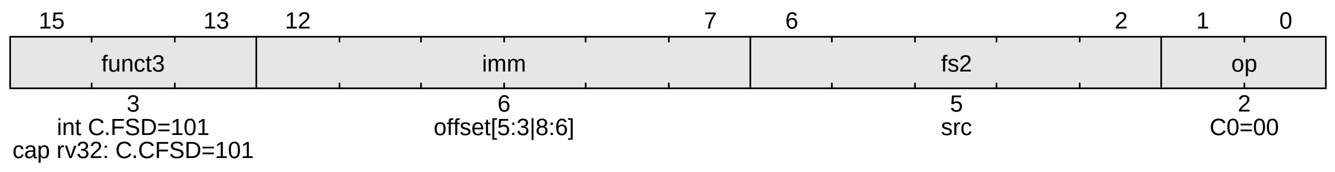
#### Legacy Mode Mnemonics (RV64CD/RV64D_Zca)

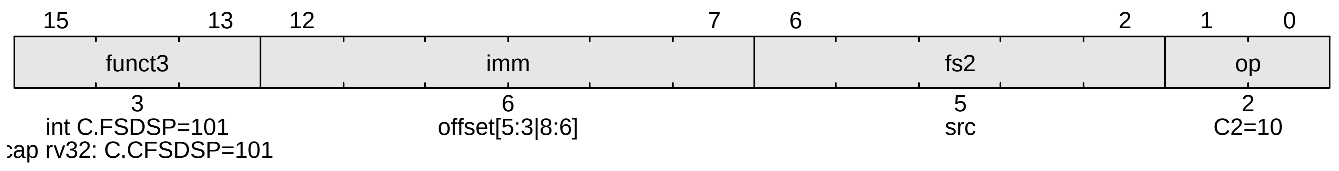
```
c.fsd fs2, offset(rs1'/sp)
```

#### 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 |

| CAUSE | Reason |
|---|---|
| Length violation | At least one byte accessed is outside the authority capability bounds |

## Prerequisites for C.CFSD, C.CFSDSP

C or Zca, Zcheri_purecap

## Prerequisites for C.FSD, C.FSDSP

C or Zca, Zcheri_legacy

## Operation (after expansion to 32-bit encodings)

See CFSD, FSD

### 8.6.47. C.CSC

see C.CSCSP.

### 8.6.48. C.CSC, C.CSCSP

#### Synopsis

Stores (C.CSC, C.CSCSP), 16-bit encodings

*These instructions have different encodings for RV64 and RV32.*

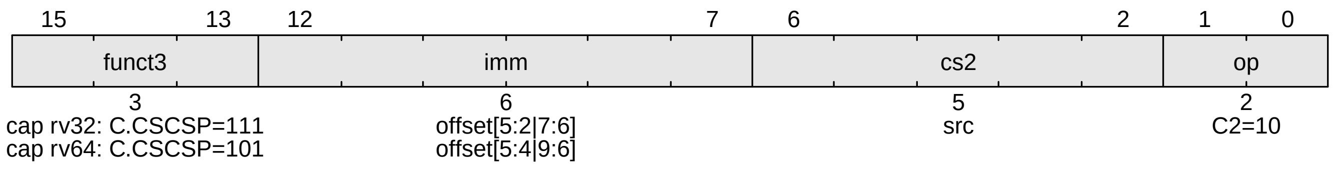
#### Capability Mode Mnemonics

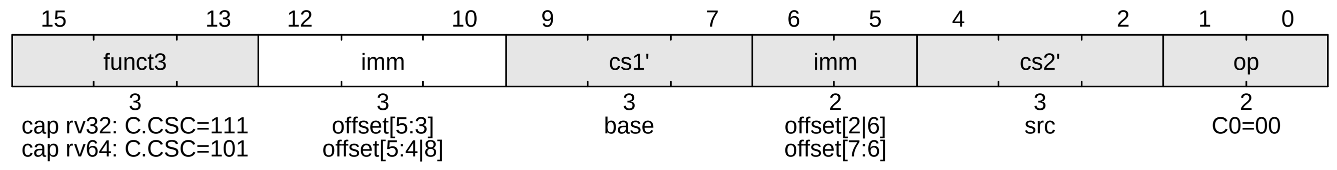
`c.csc cs2', offset(cs1'/csp)`

#### Capability Mode Expansions

`csc 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 CSC

## 8.7. "Zicbom", "Zicbop", "Zicboz" Standard Extensions for Base Cache Management Operations

## 8.7.1. CBO.CLEAN

See CBO.CLEAN.CAP.

## 8.7.2. CBO.CLEAN.CAP

### Synopsis

Perform a clean operation on a cache block

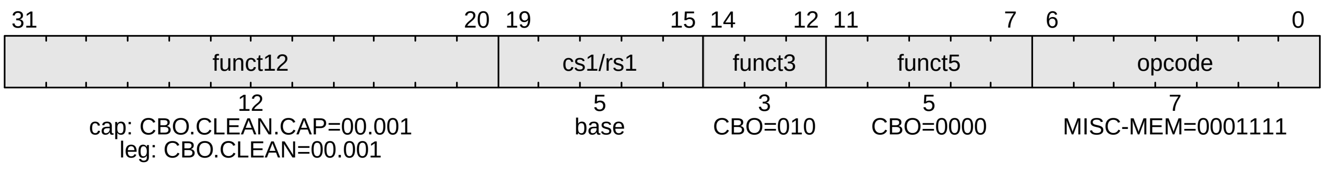
### Capability Mode Mnemonic

```
cbo.clean.cap 0(cs1)
```

### Legacy Mode Mnemonic

```
cbo.clean 0(rs1)
```

### Encoding



### Capability Mode Description

A CBO.CLEAN.CAP 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 CBO.CLEAN.CAP

Zicbom, Zcheri_purecap

### Prerequisites for CBO.CLEAN

Zicbom, Zcheri_legacy

### Operation

```
TBD
```

### 8.7.3. CBO.FLUSH

See CBO.FLUSH.CAP.

### 8.7.4. CBO.FLUSH.CAP

#### Synopsis

Perform a flush operation on a cache block

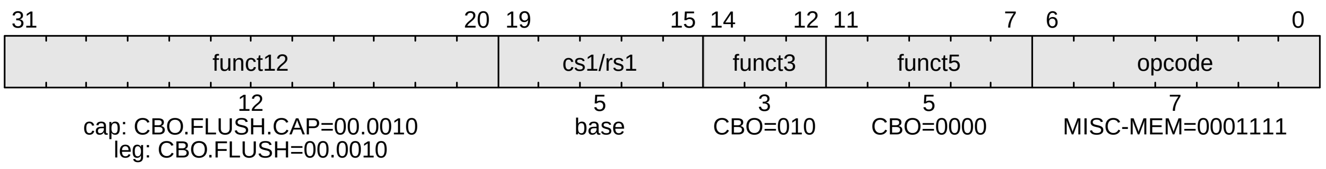
#### Capability Mode Mnemonic

`cbo.flush.cap 0(cs1)`

#### Legacy Mode Mnemonic

`cbo.flush 0(rs1)`

#### Encoding



#### Capability Mode Description

A CBO.FLUSH.CAP 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 CBO.FLUSH.CAP

Zicbom, Zcheri_purecap

#### Prerequisites for CBO.FLUSH

Zicbom, Zcheri_legacy

#### Operation

```
TBD
```

### 8.7.5. CBO.INVAL

See CBO.INVAL.CAP.

### 8.7.6. CBO.INVAL.CAP

#### Synopsis

Perform an invalidate operation on a cache block

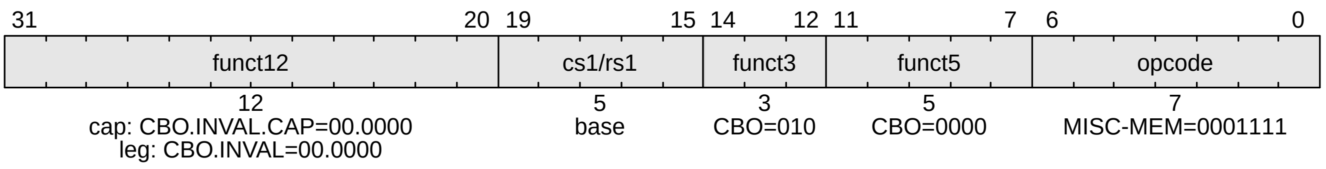
#### Capability Mode Mnemonic

```
cbo.inval.cap 0(cs1)
```

#### Legacy Mode Mnemonic

```
cbo.inval 0(rs1)
```

#### Encoding



#### Capability Mode Description

A CBO.INVAL.CAP 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.INVAL 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 in 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.INVAL.CAP and CBO.INVAL perform 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 CBO.INVAL.CAP

Zicbom, Zcheri_purecap

#### Prerequisites for CBO.INVAL

Zicbom, Zcheri_legacy

#### Operation

```
TBD
```

## 8.7.7. CBO.ZERO

See CBO.ZERO.CAP.

## 8.7.8. CBO.ZERO.CAP

#### Synopsis

Store zeros to the full set of bytes corresponding to a cache block

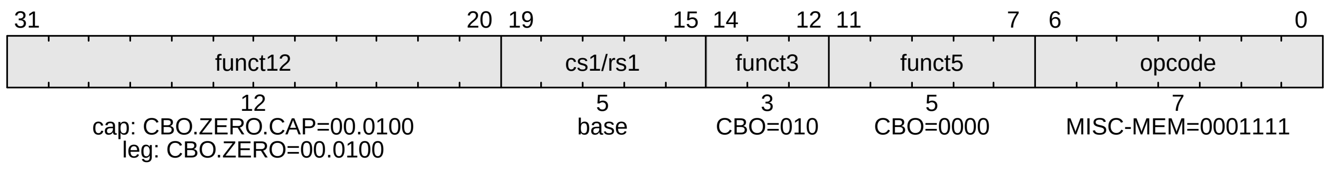
#### Capability Mode Mnemonic

```
cbo.zero.cap 0(cs1)
```

#### Legacy Mode Mnemonic

```
cbo.zero 0(rs1)
```

#### Encoding



#### Capability Mode Description

A `cbo.zero.cap` 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 CBO.ZERO.CAP

Zicboz, Zcheri_purecap

#### Prerequisites for CBO.ZERO

Zicboz, Zcheri_legacy

#### Operation

```
TBD
```

## 8.7.9. PREFETCH.I

See PREFETCH.I.CAP.

## 8.7.10. PREFETCH.I.CAP

### Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by an instruction fetch in the near future

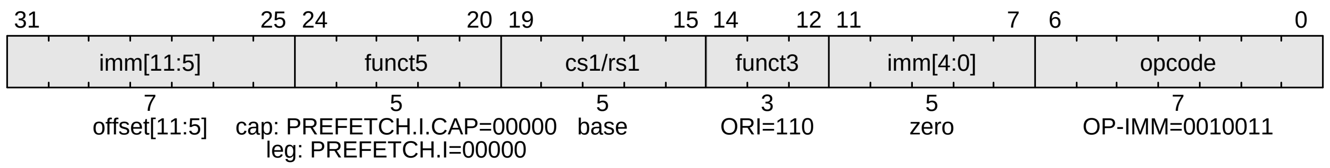
### Capability Mode Mnemonic

`prefetch.i.cap offset(cs1)`

### Legacy Mode Mnemonic

`prefetch.i offset(rs1)`

### Encoding



### Capability Mode Description

A PREFETCH.I.CAP 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 0b00000, 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`.

### 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 0b00000, 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.

### 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 X-permission |
| Length violation | At least one byte accessed is within the bounds |

### Prerequisites for PREFETCH.I.CAP

Zicbop, Zcheri_purecap

### Prerequisites for PREFETCH.I

Zicbop, Zcheri_legacy

### Operation

```
TODO
```

## 8.7.11. PREFETCH.R

See PREFETCH.R.CAP.

## 8.7.12. PREFETCH.R.CAP

### Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by a data read in the near future

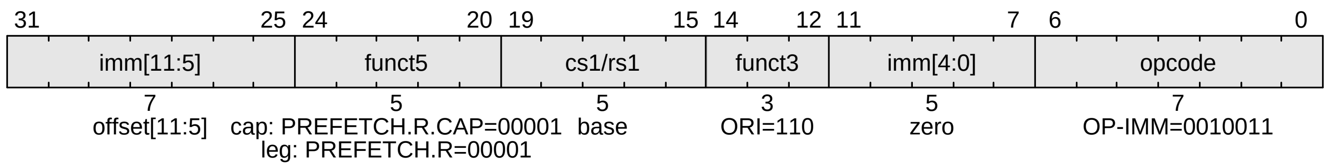
### Capability Mode Mnemonic

```
prefetch.r.cap offset(cs1)
```

### Legacy Mode Mnemonic

```
prefetch.r offset(rs1)
```

### Encoding



### Capability Mode Description

A PREFETCH.R.CAP 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 0b00000, 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`.

### 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 0b00000, 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.

### 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 R-permission |
| Length violation | At least one byte accessed is within the bounds |

### Prerequisites for PREFETCH.R.CAP

Zicbop, Zcheri_purecap

### Prerequisites for PREFETCH.R

Zicbop, Zcheri_legacy

### Operation

```
TODO
```

### 8.7.13. PREFETCH.W

See PREFETCH.W.CAP.

### 8.7.14. PREFETCH.W.CAP

#### Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by a data write in the near future

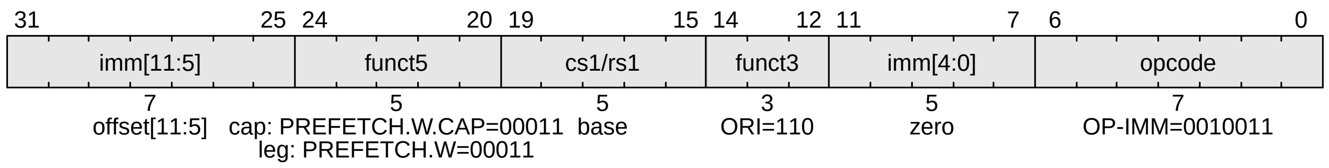
#### Capability Mode Mnemonic

```
prefetch.w.cap offset(cs1)
```

#### Legacy Mode Mnemonic

```
prefetch.w offset(rs1)
```

#### Encoding



#### Capability Mode Description

A PREFETCH.W.CAP 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 0b00000, 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`.

#### 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 0b00000, 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 PREFETCH.W.CAP

Zcheri_purecap

#### Prerequisites for PREFETCH.W

Zcheri_legacy

#### Operation

```
TODO
```

## 8.8. "Zba" Extension for Bit Manipulation Instructions

### 8.8.1. CSH1ADD

See SH3ADD.

### 8.8.2. CSH2ADD

See SH3ADD.

### 8.8.3. CSH3ADD

See SH3ADD.

### 8.8.4. SH1ADD

See SH3ADD.

### 8.8.5. SH2ADD

See SH3ADD.

## 8.8.6. SH3ADD

### Synopsis

Shift by *n* and add for address generation

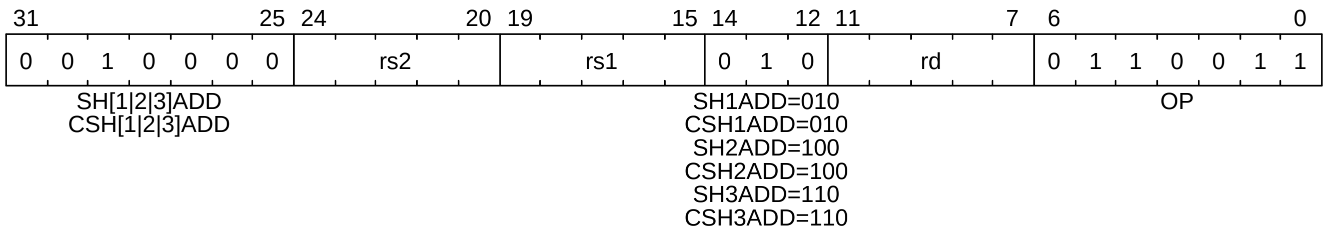
### Capability Mode Mnemonics

`csh[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 `cs1` by `rs2` shifted left by *n* bit positions. Clear the tag if the resulting capability is unrepresentable or `cs1` is sealed.

### Legacy Mode Description

Increment the address field of `rs1` by `rs2` shifted left by *n* bit positions.

### Prerequisites CSH[1|2|3]ADD

Zcheri_purecap, Zba

### Prerequisites for SH[1|2|3]ADD

Zcheri_legacy, Zba

### Capability Mode Operation

```
TBD
```

### Legacy Mode Operation

TODO

### 8.8.7. CSH1ADD.UW

See SH3ADD.UW.

### 8.8.8. CSH2ADD.UW

See SH3ADD.UW.

### 8.8.9. CSH3ADD.UW

See SH3ADD.UW.

### 8.8.10. SH1ADD.UW

See SH3ADD.UW.

### 8.8.11. SH2ADD.UW

See SH3ADD.UW.

### 8.8.12. SH3ADD.UW

**Synopsis**

Shift by *n* and add unsigned word for address generation

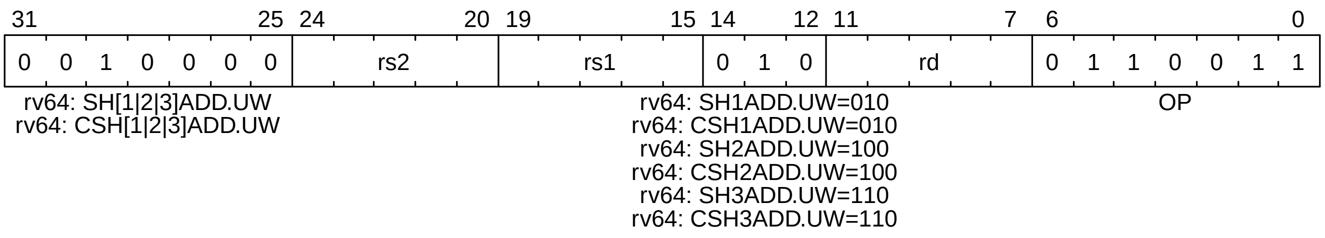
**Capability Mode Mnemonic (RV64)**

`csh[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 `cs1` by the unsigned word in `rs2` shifted left by *n* bit positions. Clear the tag if the resulting capability is unrepresentable or `cs1` is sealed.

**Legacy Mode Description**

Increment the address field of `rs1` by the unsigned word in `rs2` shifted left by *n* bit positions.

**Prerequisites CSH[1|2|3]ADD.UW**

Zcheri_purecap, Zba

**Prerequisites for SH[1|2|3]ADD.UW**

Zcheri_legacy, Zba

**Capability Mode Operation**

```
TBD
```

**Legacy Mode Operation**

TODO

### 8.8.13. SH4ADD

See CSH4ADD.

### 8.8.14. CSH4ADD

**CHERI v9 Note:** *This instruction is* **new**.

**Synopsis**

Shift by 4 and add for address generation (CSH4ADD, SH4ADD)

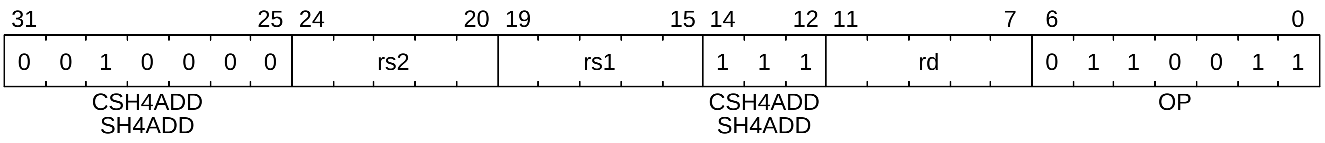
**Capability Mode Mnemonics**

`csh4add cd, rs1, cs2`

**Legacy Mode Mnemonics**

`sh4add rd, rs1, rs2`

**Encoding**



**Capability Mode Description**

Increment the address field of `cs1` by `rs2` shifted left by 4 bit positions. Clear the tag if the resulting capability is unrepresentable or `cs1` is sealed.

**Legacy Mode Description**

Increment the address field of `rs1` by `rs2` shifted left by 4 bit positions.

**Prerequisites CSH4ADD**

Zcheri_purecap

**Prerequisites for SH4ADD**

Zcheri_legacy

**Capability Mode Operation**

```
TBD
```

**Legacy Mode Operation**

TBD

### 8.8.15. SH4ADD.UW

See CSH4ADD.UW.

### 8.8.16. CSH4ADD.UW

#### Synopsis

Shift by 4 and add unsigned words for address generation (CSH4ADD.UW, SH4ADD.UW)

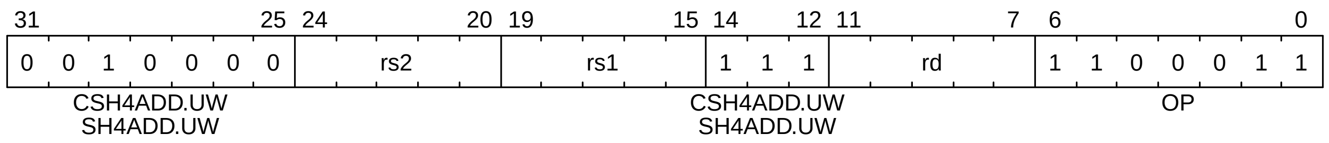
#### Capability Mode Mnemonics

```
csh4add.uw cd, rs1, cs2
```

#### Legacy Mode Mnemonics

```
sh4add.uw rd, rs1, rs2
```

#### Encoding



#### Capability Mode Description

Increment the address field of `cs1` by the unsigned word in `rs2` shifted left by 4 bit positions. Clear the tag if the resulting capability is unrepresentable or `cs1` is sealed.

#### Legacy Mode Description

Increment the address field of `rs1` by the unsigned word in `rs2` shifted left by 4 bit positions.

#### Prerequisites CSH4ADD

Zcheri_purecap

#### Prerequisites for SH4ADD

Zcheri_legacy

#### Capability Mode Operation

```
TBD
```

#### Legacy Mode Operation

TBD

## 8.9. "Zcb" Standard Extension For Code-Size Reduction

### 8.9.1. C.CLH

See C.LBU.

### 8.9.2. C.CLHU

See C.LBU.

### 8.9.3. C.CLBU

See C.LBU.

### 8.9.4. C.LH

See C.LBU.

### 8.9.5. C.LHU

See C.LBU.

### 8.9.6. C.LBU

#### Synopsis

Load (C.CLH, C.CLHU, C.CLBU, C.LH, C.LHU, C.LBU), 16-bit encodings

#### Capability Mode Mnemonics

`c.clh/c.clhu/c.clbu rd', offset(cs1')`

#### Capability Mode Expansions

`clh/clhu/clbu rd, offset(cs1)`

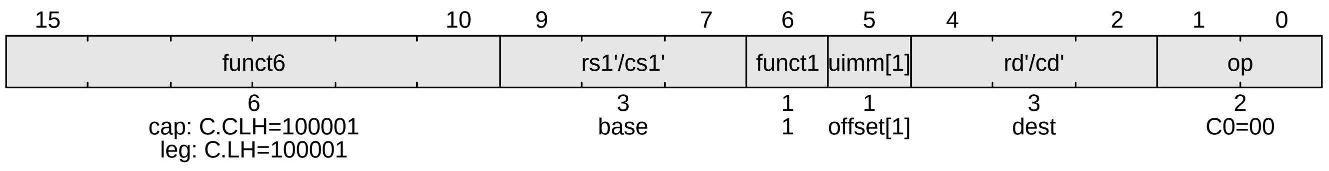
#### Legacy Mode Mnemonics

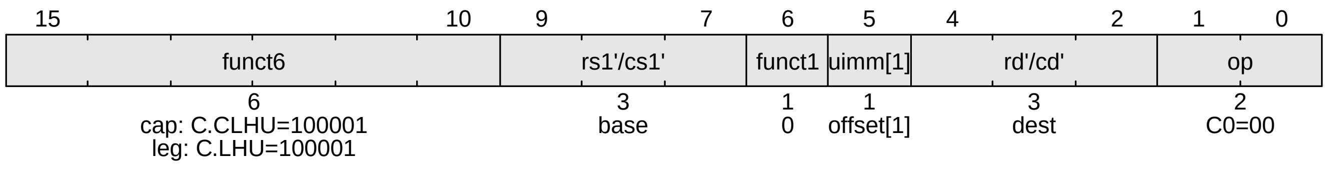
`c.lh/c.lhu/c.lbu rd', offset(rs1')`

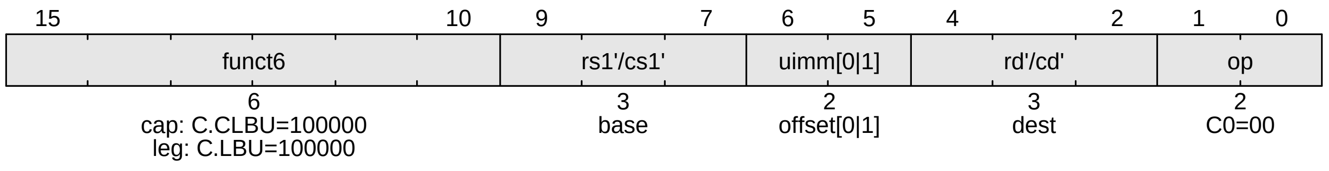
#### Legacy Mode Expansions

`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 C.CLH, C.CLHU, C.CLBU**

C or Zca, Zcheri_purecap, and Zcb

**Prerequisites C.LH, C.LHU, C.LBU**

C or Zca, Zcheri_legacy, and Zcb

**Operation (after expansion to 32-bit encodings)**

See C.CLH, CLHU, CLBU, LH, LHU, LBU

### 8.9.7. C.CSH

See C.SB.

### 8.9.8. C.CSB

See C.SB.

### 8.9.9. C.SH

See C.SB.

### 8.9.10. C.CSH, C.CSB, C.SH, C.SB

#### Synopsis

Stores (C.CSH, C.CSB, C.SH, C.SB), 16-bit encodings

#### Capability Mode Mnemonics

```
c.csh/c.csb rs2', offset(cs1')
```

#### Capability Mode Expansions

```
csh/csb rs2', offset(cs1')
```

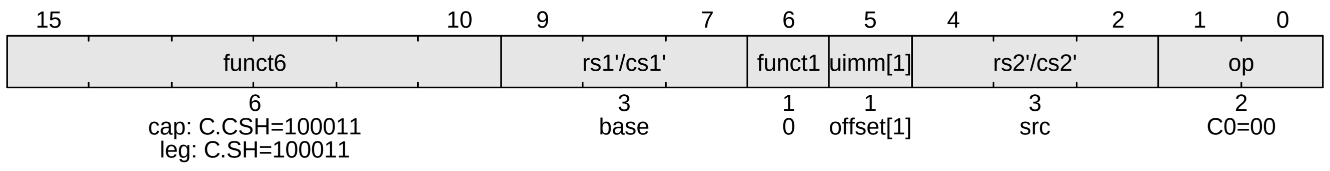
#### Legacy Mode Mnemonics

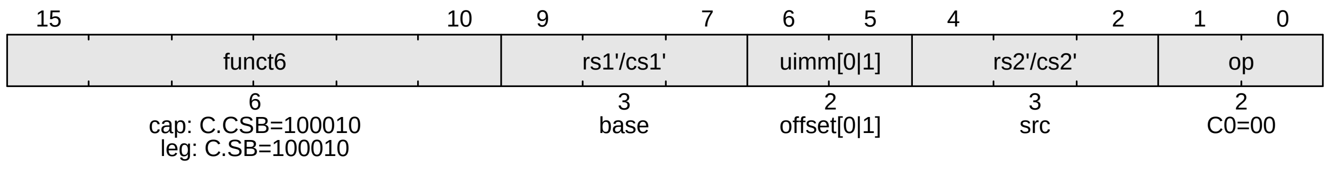
```
c.sh/c.sb rs2', offset(rs1')
```

#### 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 C.CSH, C.CSB

C or Zca, Zcheri_purecap, and Zcb

#### Prerequisites for C.SH, C.SB

C or Zca, Zcheri_legacy, and Zcb

#### Operation (after expansion to 32-bit encodings)

See CSH, CSB, SH, SB

## 8.10. "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.MVSA01, CM.MVA01S) are redefined in capability mode to move full capabilities between registers. The saved register mapping is as shown in

| 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 |

*Table 30. saved register mapping for Zcmp*

All instructions are defined in (RISC-V, 2023).

### 8.10.1. CM.PUSH

See CM.CPUSH and (RISC-V, 2023).

### 8.10.2. CM.CPUSH

#### Synopsis

Create stack frame (CM.CPUSH, 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.

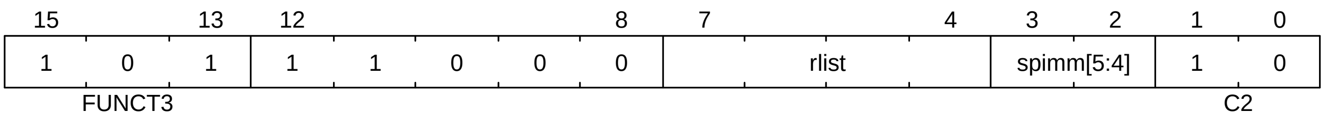
#### Capability Mode Mnemonic

```
cm.cpush {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 CM.CPUSH

C or Zca, Zcheri_purecap, Zcmp

#### Prerequisites for CM.PUSH

C or Zca, Zcheri_legacy, Zcmp

#### Operation

```
TBD
```

### 8.10.3. CM.POP

See CM.CPOP and (RISC-V, 2023).

### 8.10.4. CM.CPOP

#### Synopsis

Destroy stack frame (CM.CPOP, CM.POP): load the return address register and 0 to 12 saved registers from the stack frame, deallocate the stack frame. 16-bit encodings.

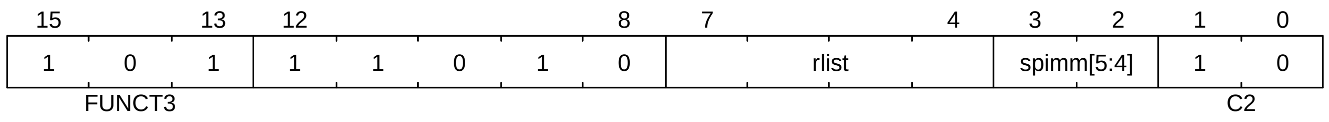
#### Capability Mode Mnemonic

```
cm.cpop {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 CM.CPOP

C or Zca, Zcheri_purecap, Zcmp

#### Prerequisites for CM.POP

C or Zca, Zcheri_legacy, Zcmp

#### Operation

```
TBD
```

### 8.10.5. CM.POPRET

See CM.CPOPRET and (RISC-V, 2023).

### 8.10.6. CM.CPOPRET

**Synopsis**

Destroy stack frame (CM.CPOPRET, 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.

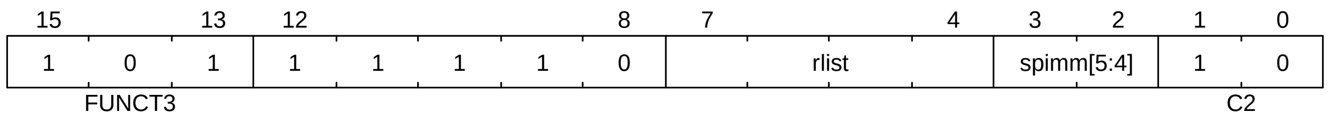
**Capability Mode Mnemonic**

`cm.cpopret {creg_list}, -stack_adj`

**Legacy Mode Mnemonics**

`cm.popret {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. Return by calling CJALR 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 CLC for Capability Mode or LC for Legacy Mode.

The return is checked as for CJALR for Capability Mode, or JALR for Legacy Mode.

**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.*

**Prerequisites for CM.CPOPRET**

C or Zca, Zcheri_purecap, Zcmp

**Prerequisites for CM.POPRET**

C or Zca, Zcheri_legacy, Zcmp

**Operation**

```
TBD
```

### 8.10.7. CM.POPRETZ

See CM.CPOPRETZ and (RISC-V, 2023).

### 8.10.8. CM.CPOPRETZ

#### Synopsis

Destroy stack frame (CM.CPOPRETZ, CM.POPRETZ): load the return address register and 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.

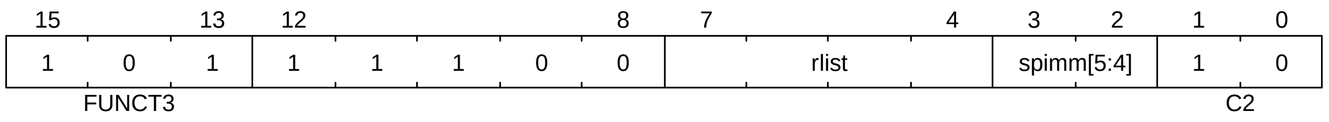
#### Capability Mode Mnemonic

```
cm.cpopretz {creg_list}, -stack_adj
```

#### Legacy Mode Mnemonics

```
cm.popretz {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. Move zero into `ca0`. Return by calling CJALR 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 CLC for Capability Mode or LC for Legacy Mode.

The return is checked as for CJALR for Capability Mode, or JALR for Legacy Mode.

#### 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.*

**Prerequisites for CM.CPOPRETZ**

C or Zca, Zcheri_purecap, Zcmp

**Prerequisites for CM.POPRETZ**

C or Zca, Zcheri_legacy, Zcmp

### Operation

```
TBD
```

### 8.10.9. CM.MVSA01

See CM.CMVSA01 and (RISC-V, 2023).

### 8.10.10. CM.CMVSA01

#### Synopsis

CM.CMVSA01, CM.MVSA01: Move argument registers 0 and 1 into two saved registers.

#### Capability Mode Mnemonic

`cm.cmvsa01 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 30.*

#### 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 CM.CMVSA01

C or Zca, Zcheri_purecap, Zcmp

#### Prerequisites for CM.MVSA01

C or Zca, Zcheri_legacy, Zcmp

#### Operation

```
TBD
```

### 8.10.11. CM.MVA01S

See CM.CMVA01S and (RISC-V, 2023).

### 8.10.12. CM.CMVA01S

#### Synopsis

CM.CMVA01S, CM.MVA01S: Move two saved registers into argument registers 0 and 1.

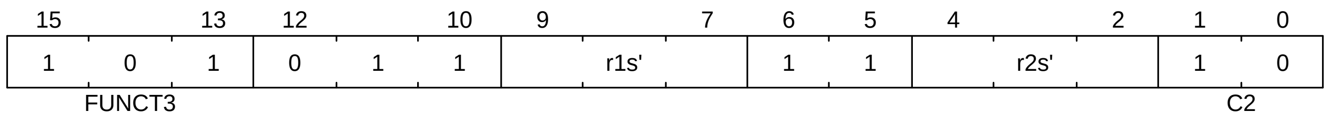
#### Capability Mode Mnemonic

`cm.cmva01s cr1s', cr2s'`

#### Legacy Mode Mnemonics

`cm.mva01s r1s', r2s'`

#### Encoding



*The encoding uses sreg number specifiers instead of xreg number specifiers to save encoding space. The saved register encoding is shown in Table 30.*

#### 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 CM.CMVA01S

C or Zca, Zcheri_purecap, Zcmp

#### Prerequisites for CM.MVA01S

C or Zca, Zcheri_legacy, Zcmp

#### Operation

```
TBD
```

## 8.11. "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, and *not* against pcc. This allows the jump table to be accessed when the pcc bounds are set narrowly to the local function only.

*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.*

### 8.11.1. Jump Vector Table CSR (jvt)

The JVT CSR is exactly as defined by (RISC-V, 2023). It is aliased to jvtc.

### 8.11.2. Jump Vector Table CSR (jvtc)

jvtc extends jvt to be a capability width CSR, as shown in Table 20.

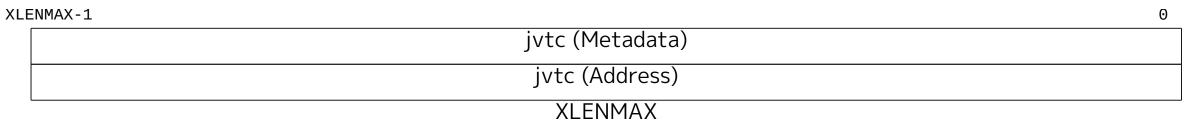


*Figure 39. Jump Vector Table Capability register*

All instruction fetches from the jump vector table are checked against jvtc.

See CM.CJALT, CM.JALT, CM.CJT, CM.JT.

### 8.11.3. CM.JALT

See CM.CJALT and (RISC-V, 2023).

### 8.11.4. CM.CJALT

#### Synopsis

Jump via table with link (CM.CJALT, CM.JALT), 16-bit encodings

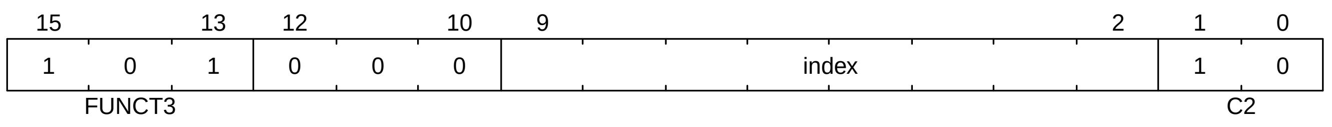
#### Capability Mode Mnemonic

`cm.cjalt index`

#### Legacy Mode Mnemonics

`cm.jalt index`

#### Encoding



*For this encoding to decode as CM.CJALT/CM.JALT, index>=32, otherwise it decodes as CM.CJT/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 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 jump table access against jvtc bounds (not against pcc) and requiring X-permission. Link to `ra`.

#### Permissions

Requires jvtc to be tagged, not sealed, have X-permission and for the full XLEN-wide access to be in jvtc bounds.

#### 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.*

**Prerequisites for CM.CJALT**

C or Zca, Zcheri_purecap, Zcmt

**Prerequisites for CM.JALT**

C or Zca, Zcheri_legacy, Zcmt

**Operation**

```
TBD
```

### 8.11.5. CM.JT

See CM.CJT and (RISC-V, 2023).

### 8.11.6. CM.CJT

#### Synopsis

Jump via table with link (CM.CJT, CM.JT), 16-bit encodings

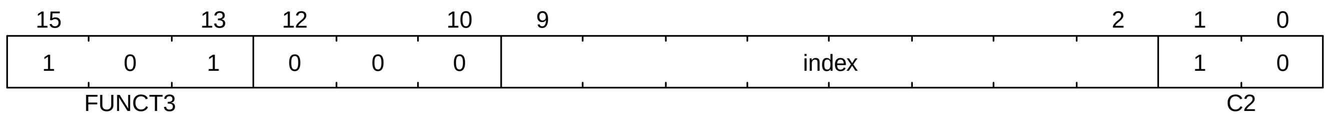
#### Capability Mode Mnemonic

`cm.cjt index`

#### Legacy Mode Mnemonics

`cm.jt index`

#### Encoding



*For this encoding to decode as CM.CJT/CM.JT, index<32, otherwise it decodes as CM.CJALT/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 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 jump table access against jvtc bounds (not against pcc) and requiring X-permission.

#### Permissions

Requires jvtc to be tagged, not sealed, have X-permission and for the full XLEN-wide access to be in jvtc bounds.

#### 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.*

#### Prerequisites for CM.CJT

C or Zca, Zcheri_purecap, Zcmt

#### Prerequisites for CM.JT

C or Zca, Zcheri_legacy, Zcmt

#### Operation

```
TBD
```

# Chapter 9. Extension summary

## 9.1. Zbhlrsc

Zbhlrsc is a separate extension independent of CHERI, but is required for CHERI software.

| Mnemonic | Zcheri_legacy | Zcheri_purecap | Function |
|---|---|---|---|
| LR.H | ✓ | | Load reserve half via int pointer, authorise with DDC |
| LR.B | ✓ | | Load reserve byte via int pointer, authorise with DDC |
| CLR.H | | ✓ | Load reserve half via cap |
| CLR.B | | ✓ | Load reserve byte via cap |
| SC.H | ✓ | | Store conditional half via int pointer, authorise with DDC |
| SC.B | ✓ | | Store conditional byte via int pointer, authorise with DDC |
| CSC.H | | ✓ | Store conditional half via cap |
| CSC.B | | ✓ | Store conditional byte via cap |

*Table 31. Zbhlrsc instruction extension*

## 9.2. 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 32

| Mnemonic | RV32 | RV64 | A | Zbhlrsc | Zicbo [mpz] | C or Zca | Zba | Zcb | Zcmp | Zcmt | Zfh | F | D | V | Function |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLC | ✓ | ✓ | | | | | | | | | | | | | Load cap via cap |
| CSC | ✓ | ✓ | | | | | | | | | | | | | Store cap via cap |
| C.CLCSP | ✓ | ✓ | | | | ✓ | | | | | | | | | Load cap via cap, SP relative |
| C.CSCSP | ✓ | ✓ | | | | ✓ | | | | | | | | | Store cap via cap, SP relative |
| C.CLC | ✓ | ✓ | | | | ✓ | | | | | | | | | Load cap via cap |
| C.CSC | ✓ | ✓ | | | | ✓ | | | | | | | | | Store cap via cap |
| C.CLWSP | ✓ | ✓ | | | | ✓ | | | | | | | | | Load word via cap, SP relative |
| C.CSWSP | ✓ | ✓ | | | | ✓ | | | | | | | | | Store word via cap, SP relative |
| C.CLW | ✓ | ✓ | | | | ✓ | | | | | | | | | Load word via cap |
| C.CSW | ✓ | ✓ | | | | ✓ | | | | | | | | | Store word via cap |
| C.CLD | | ✓ | | | | ✓ | | | | | | | | | Load word via cap |
| C.CSD | | ✓ | | | | ✓ | | | | | | | | | Store word via cap |
| C.CLDSP | | ✓ | | | | ✓ | | | | | | | | | Load word via cap |
| C.CSDSP | | ✓ | | | | ✓ | | | | | | | | | Store word via cap |
| CLB | ✓ | ✓ | | | | | | | | | | | | | Load signed byte via cap |
| CLH | ✓ | ✓ | | | | | | | | | | | | | Load signed half via cap |
| C.CLH | ✓ | ✓ | | | | | | ✓ | | | | | | | Load signed half via cap |
| CLW | ✓ | ✓ | | | | | | | | | | | | | Load signed word via cap |
| CLBU | ✓ | ✓ | | | | | | | | | | | | | Load unsigned byte via cap |
| C.CLBU | ✓ | ✓ | | | | | | ✓ | | | | | | | Load unsigned byte via cap |
| CLHU | ✓ | ✓ | | | | | | | | | | | | | Load unsigned half via cap |
| C.CLHU | ✓ | ✓ | | | | | | ✓ | | | | | | | Load unsigned half via cap |
| CLWU | | ✓ | | | | | | | | | | | | | Load unsigned word via cap |
| CLD | | ✓ | | | | | | | | | | | | | Load double via cap |

| Mnemonic | RV32 | RV64 | A | Zbhlrsc | Zicbo [mpz] | C or Zca | Zba | Zcb | Zcmp | Zcmt | Zfh | F | D | V | Function |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CSB | ✓ | ✓ | | | | | | | | | | | | | Store byte via cap |
| C.CSB | ✓ | ✓ | | | | | | ✓ | | | | | | | Store byte via cap |
| CSH | ✓ | ✓ | | | | | | | | | | | | | Store half via cap |
| C.CSH | ✓ | ✓ | | | | | | ✓ | | | | | | | Store half via cap |
| CSW | ✓ | ✓ | | | | | | | | | | | | | Store word via cap |
| CSD | | ✓ | | | | | | | | | | | | | Store double via cap |
| AUIPCC | ✓ | ✓ | | | | | | | | | | | | | Add immediate to PCC address, representability check |
| CINCOFFSET | ✓ | ✓ | | | | | | | | | | | | | Increment cap address by register, representability check |
| CINCOFFSETIMM | ✓ | ✓ | | | | | | | | | | | | | Increment cap address by immediate, representability check |
| CSETADDR | ✓ | ✓ | | | | | | | | | | | | | Replace capability address, representability check |
| CGETTAG | ✓ | ✓ | | | | | | | | | | | | | Get tag field |
| CGETPERM | ✓ | ✓ | | | | | | | | | | | | | Get hperm and uperm fields as 1-bit per permission, packed together |
| CMOVE | ✓ | ✓ | | | | | | | | | | | | | Move capability register |
| CANDPERM | ✓ | ✓ | | | | | | | | | | | | | AND capability permissions (expand to 1-bit per permission before ANDing) |
| CGETHIGH | ✓ | ✓ | | | | | | | | | | | | | Get metadata |
| CSETHIGH | ✓ | ✓ | | | | | | | | | | | | | Set metadata and clear tag |
| CSETEQUALEXACT | ✓ | ✓ | | | | | | | | | | | | | Full capability bitwise compare |
| CSEAL | ✓ | ✓ | | | | | | | | | | | | | Seal capability |
| CTESTSUBSET | ✓ | ✓ | | | | | | | | | | | | | Set register bounds on capability with rounding, clear tag if rounding is required |
| CBUILDCAP | ✓ | ✓ | | | | | | | | | | | | | Set cd to cs2 with its tag set after checking that cs2 is a subset of cs1 |
| CSETBOUNDS | ✓ | ✓ | | | | | | | | | | | | | Set register bounds on capability with rounding, clear tag if rounding is required |
| CSETBOUNDSIMM | ✓ | ✓ | | | | | | | | | | | | | Set immediate bounds on capability with rounding, clear tag if rounding is required |
| CSETBOUNDSINEXACT | ✓ | ✓ | | | | | | | | | | | | | Set bounds on capability with rounding up as required |
| CRAM | ✓ | ✓ | | | | | | | | | | | | | Representable Alignment Mask: Return mask to apply to address to get the requested bounds |
| CGETBASE | ✓ | ✓ | | | | | | | | | | | | | Get capability base |
| CGETLEN | ✓ | ✓ | | | | | | | | | | | | | Get capability length |
| C.CINCOFFSET16CSP | ✓ | ✓ | | | | ✓ | | | | | | | | | ADD immediate to stack pointer, representability check |
| C.CINCOFFSET4CSPN | ✓ | ✓ | | | | ✓ | | | | | | | | | ADD immediate to stack pointer, representability check |
| C.CMOVE | ✓ | ✓ | | | | ✓ | | | | | | | | | Same as CMove |

| Mnemonic | RV32 | RV64 | A | Zbhlrsc | Zicbo [mpz] | C or Zca | Zba | Zcb | Zcmp | Zcmt | Zfh | F | D | V | Function |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C.CJ | ✓ | ✓ | | | | ✓ | | | | | | | | | Jump to PC+offset, bounds check minimum size target instruction |
| C.CJAL | ✓ | | | | | ✓ | | | | | | | | | Jump to PC+offset, bounds check minimum size target instruction, link to cd |
| CJAL | ✓ | ✓ | | | | ✓ | | | | | | | | | Jump to PC+offset, bounds check minimum size target instruction, link to cd |
| JALR.PCC | ✓ | ✓ | | | | | | | | | | | | | RISC-V JALR available in capability modes (with zero offset) |
| CJALR | ✓ | ✓ | | | | | | | | | | | | | Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap |
| C.CJALR | ✓ | ✓ | | | | ✓ | | | | | | | | | Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap |
| C.CJR | ✓ | ✓ | | | | ✓ | | | | | | | | | Indirect cap jump, bounds check minimum size target instruction, unseal target cap |
| CBO.INVAL.CAP | ✓ | ✓ | | | ✓ | | | | | | | | | | Cache block invalidate (implemented as clean), via cap |
| CBO.CLEAN.CAP | ✓ | ✓ | | | ✓ | | | | | | | | | | Cache block clean, via cap |
| CBO.FLUSH.CAP | ✓ | ✓ | | | ✓ | | | | | | | | | | Cache block flush, via cap |
| CBO.ZERO.CAP | ✓ | ✓ | | | ✓ | | | | | | | | | | Cache block zero, via cap |
| PREFETCH.R.CAP | ✓ | ✓ | | | ✓ | | | | | | | | | | Prefetch read-only data cache line, via cap |
| PREFETCH.W.CAP | ✓ | ✓ | | | ✓ | | | | | | | | | | Prefetch writeable data cache line, via cap |
| PREFETCH.I.CAP | ✓ | ✓ | | | ✓ | | | | | | | | | | Prefetch instruction cache line, via cap |
| CLR.C | ✓ | ✓ | ✓ | | | | | | | | | | | | Load reserve cap via cap |
| CLR.D | | ✓ | ✓ | | | | | | | | | | | | Load reserve double via cap |
| CLR.W | ✓ | ✓ | ✓ | | | | | | | | | | | | Load reserve word via cap |
| CLR.H | ✓ | ✓ | | ✓ | | | | | | | | | | | Load reserve half via cap |
| CLR.B | ✓ | ✓ | | ✓ | | | | | | | | | | | Load reserve byte via cap |
| CSC.C | ✓ | ✓ | ✓ | | | | | | | | | | | | Store conditional cap via cap |
| CSC.D | | ✓ | ✓ | | | | | | | | | | | | Store conditional double via cap |
| CSC.W | ✓ | ✓ | ✓ | | | | | | | | | | | | Store conditional word via cap |
| CSC.H | ✓ | ✓ | | ✓ | | | | | | | | | | | Store conditional half via cap |
| CSC.B | ✓ | ✓ | | ✓ | | | | | | | | | | | Store conditional byte via cap |
| CAMOSWAP.C | ✓ | ✓ | ✓ | | | | | | | | | | | | Atomic swap of cap via cap |
| CAMO<OP>.W | ✓ | ✓ | ✓ | | | | | | | | | | | | Atomic op of word via cap |
| CAMO<OP>.D | | ✓ | ✓ | | | | | | | | | | | | Atomic op of double via cap |
| C.CFLD | ✓ | | | | | | | | | | ✓ | | ✓ | | Load floating point double via cap |
| C.CFLDSP | ✓ | | | | | | | | | | | | ✓ | | Load floating point double via cap, sp relative |
| C.CFSD | ✓ | | | | | | | | | | | | ✓ | | Store floating point double via cap |
| C.CFSDSP | ✓ | | | | | | | | | | | | ✓ | | Store floating point double via cap, sp relative |
| CFLH | ✓ | ✓ | | | | | | | | | ✓ | | | | Load floating point half via cap |

| Mnemonic | RV32 | RV64 | A | Zbhlr sc | Zicbo [mpz] | C or Zca | Zba | Zcb | Zcmp | Zcmt | Zfh | F | D | V | Function |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CFSH | ✓ | ✓ | | | | | | | | | ✓ | | | | Store floating point half via cap |
| CFLW | ✓ | ✓ | | | | | | | | | | ✓ | | | Load floating point word via cap |
| CFSW | ✓ | ✓ | | | | | | | | | | ✓ | | | Store floating point word via cap |
| CFLD | ✓ | ✓ | | | | | | | | | | | ✓ | | Load floating point double via cap |
| CFSD | ✓ | ✓ | | | | | | | | | | | ✓ | | Store floating point double via cap |
| CM.CPUSH | ✓ | ✓ | | | | | | | ✓ | | | | | | Push capability stack frame |
| CM.CPOP | ✓ | ✓ | | | | | | | ✓ | | | | | | Pop capability stack frame |
| CM.CPOPRET | ✓ | ✓ | | | | | | | ✓ | | | | | | Pop capability stack frame and return |
| CM.CPOPRETZ | ✓ | ✓ | | | | | | | ✓ | | | | | | Pop capability stack frame and return zero |
| CM.CMVSA01 | ✓ | ✓ | | | | | | | ✓ | | | | | | Move two capability registers |
| CM.CMVA01S | ✓ | ✓ | | | | | | | ✓ | | | | | | Move two capability registers |
| CM.CJALT | ✓ | ✓ | | | | | | | | ✓ | | | | | Table jump and link |
| CM.CJT | ✓ | ✓ | | | | | | | | ✓ | | | | | Table jump |
| CSH1ADD | ✓ | ✓ | | | | | ✓ | | | | | | | | shift and add, representability check on the result |
| CSH1ADD.UW | ✓ | ✓ | | | | | ✓ | | | | | | | | shift and add, representability check on the result |
| CSH2ADD | ✓ | ✓ | | | | | ✓ | | | | | | | | shift and add, representability check on the result |
| CSH2ADD.UW | ✓ | ✓ | | | | | ✓ | | | | | | | | shift and add, representability check on the result |
| CSH3ADD | ✓ | ✓ | | | | | ✓ | | | | | | | | shift and add, representability check on the result |
| CSH3ADD.UW | ✓ | ✓ | | | | | ✓ | | | | | | | | shift and add, representability check on the result |
| CSH4ADD | | ✓ | | | | | | | | | | | | | shift and add, representability check on the result |
| CSH4ADD.UW | | ✓ | | | | | | | | | | | | | shift and add, representability check on the result |

*Table 32. Zcheri_purecap instruction extension - Pure Capability Mode instructions*

## 9.3. Zcheri_legacy

Zcheri_legacy defines the set of instructions added by the legacy mode, in addition to Zcheri_purecap.

*Zcheri_legacy implies Zcheri_purecap*

| Mnemonic | RV32 | RV64 | A | Zbhlr sc | Zicbo [mpz] | C or Zca | Zba | Zcb | Zcmp | Zcmt | Zfh | F | D | V | Function |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LC | ✓ | ✓ | | | | | | | | | | | | | Load cap via int pointer, authorise with DDC |
| SC | ✓ | ✓ | | | | | | | | | | | | | Store cap via int pointer, authorise with DDC |
| LB | ✓ | ✓ | | | | | | | | | | | | | Load signed byte |
| LH | ✓ | ✓ | | | | | | | | | | | | | Load signed half |
| C.LH | ✓ | ✓ | | | | | | ✓ | | | | | | | Load signed half |
| LW | ✓ | ✓ | | | | | | | | | | | | | Load signed word |
| LBU | ✓ | ✓ | | | | | | | | | | | | | Load unsigned byte |

| Mnemonic | RV32 | RV64 | A | Zbhlr sc | Zicbo [mpz] | C or Zca | Zba | Zcb | Zcmp | Zcmt | Zfh | F | D | V | Function |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |
| C.ADDI16SP | ✓ | ✓ | | | | ✓ | | | | | | | | | ADD immediate to stack pointer |
| C.ADDI4SPN | ✓ | ✓ | | | | ✓ | | | | | | | | | ADD immediate to stack pointer, representability check |
| C.MV | ✓ | ✓ | | | | ✓ | | | | | | | | | Register Move |
| JALR.CAP | ✓ | ✓ | | | | | | | | | | | | | CJALR available in legacy mode (with zero offset) |
| CBO.INVAL | ✓ | ✓ | | | ✓ | | | | | | | | | | Cache block invalidate (implemented as clean), authorise with DDC |
| CBO.CLEAN | ✓ | ✓ | | | ✓ | | | | | | | | | | Cache block clean, authorise with DDC |
| CBO.FLUSH | ✓ | ✓ | | | ✓ | | | | | | | | | | Cache block flush, authorise with DDC |
| CBO.ZERO | ✓ | ✓ | | | ✓ | | | | | | | | | | Cache block zero, authorise with DDC |
| PREFETCH.R | ✓ | ✓ | | | ✓ | | | | | | | | | | Prefetch instruction cache line, always valid |
| PREFETCH.W | ✓ | ✓ | | | ✓ | | | | | | | | | | Prefetch read-only data cache line, authorise with DDC |
| PREFETCH.I | ✓ | ✓ | | | ✓ | | | | | | | | | | Prefetch writeable data cache line, authorise with DDC |
| LR.C | ✓ | ✓ | ✓ | | | | | | | | | | | | Load reserve cap via int pointer, authorise with DDC |
| LR.H | ✓ | ✓ | | ✓ | | | | | | | | | | | Load reserve half via int pointer, authorise with DDC |
| LR.B | ✓ | ✓ | | ✓ | | | | | | | | | | | Load reserve byte via int pointer, authorise with DDC |
| SC.C | ✓ | ✓ | ✓ | | | | | | | | | | | | Store conditional cap via int pointer, authorise with DDC |
| SC.H | ✓ | ✓ | | ✓ | | | | | | | | | | | Store conditional half via int pointer, authorise with DDC |
| SC.B | ✓ | ✓ | | ✓ | | | | | | | | | | | Store conditional byte via int pointer, authorise with DDC |
| AMOSWAP.C | ✓ | ✓ | ✓ | | | | | | | | | | | | Atomic swap of cap |
| AMO<OP>.W | ✓ | ✓ | ✓ | | | | | | | | | | | | Atomic op of word |
| AMO<OP>.D | | ✓ | ✓ | | | | | | | | | | | | Atomic op of double |
| C.FLW | ✓ | | | | | | | | | | | ✓ | | | Load floating point word via cap |
| C.FLWSP | ✓ | | | | | | | | | | | ✓ | | | Load floating point word, sp relative |
| C.FSW | ✓ | | | | | | | | | | | ✓ | | | Store floating point word via cap |

| Mnemonic | RV32 | RV64 | A | Zbhlr sc | Zicbo [mpz] | C or Zca | Zba | Zcb | Zcmp | Zcmt | Zfh | F | D | V | Function |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |
| 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.MVSA01 | ✓ | ✓ | | | | | | | ✓ | | | | | | Move two integer registers |
| CM.MVA01S | ✓ | ✓ | | | | | | | ✓ | | | | | | Move two integer registers |
| CM.JALT | ✓ | ✓ | | | | | | | | ✓ | | | | | Table jump and link |
| CM.JT | ✓ | ✓ | | | | | | | | ✓ | | | | | Table jump |
| SH4ADD | | ✓ | | | | | | | | | | | | | shift and add |
| SH4ADD.UW | | ✓ | | | | | | | | | | | | | shift and add |

*Table 33. Zcheri_legacy instruction extension - legacy mode instructions*

## 9.4. Zcheri_mode

Zcheri_legacy defines the set of instructions added by the mode switching mode, in addition to Zcheri_legacy.

*Zcheri_mode implies Zcheri_legacy*

| Mnemonic | RV32 | RV64 | A | Zbhlr sc | Zicbo [mpz] | C or Zca | Zba | Zcb | Zcmp | Zcmt | Zfh | F | D | V | Function |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Section 8.1.9 | ✓ | ✓ | | | | | | | | | | | | | Set the mode bit of a capability, no permissions required |
| CMODESWITCH | ✓ | ✓ | | | | | | | | | | | | | Directly switch mode (legacy / capability) |
| C.CMODESWITCH | ✓ | ✓ | | | | | | | | | | | | | Directly switch mode (legacy / capability) |

*Table 34. Zcheri_mode instruction extension - mode switching instructions*

## 9.5. Instruction Modes

The tables summarise which operating modes each instruction may be executed in.

| Mnemonic | Zcheri_mode | Zcheri_legacy | Zcheri_purecap | Function |
|---|---|---|---|---|
| CLC | | | ✓ | Load cap via cap |
| CSC | | | ✓ | Store cap via cap |
| C.CLCSP | | | ✓ | Load cap via cap, SP relative |
| C.CSCSP | | | ✓ | Store cap via cap, SP relative |
| C.CLC | | | ✓ | Load cap via cap |
| C.CSC | | | ✓ | Store cap via cap |
| C.CLWSP | | | ✓ | Load word via cap, SP relative |

| Mnemonic | Zcheri_mode | Zcheri_legacy | Zcheri_purecap | Function |
|---|---|---|---|---|
| C.CSWSP | | | ✓ | Store word via cap, SP relative |
| C.CLW | | | ✓ | Load word via cap |
| C.CSW | | | ✓ | Store word via cap |
| C.CLD | | | ✓ | Load word via cap |
| C.CSD | | | ✓ | Store word via cap |
| C.CLDSP | | | ✓ | Load word via cap |
| C.CSDSP | | | ✓ | Store word via cap |
| CLB | | | ✓ | Load signed byte via cap |
| CLH | | | ✓ | Load signed half via cap |
| C.CLH | | | ✓ | Load signed half via cap |
| CLW | | | ✓ | Load signed word via cap |
| CLBU | | | ✓ | Load unsigned byte via cap |
| C.CLBU | | | ✓ | Load unsigned byte via cap |
| CLHU | | | ✓ | Load unsigned half via cap |
| C.CLHU | | | ✓ | Load unsigned half via cap |
| CLWU | | | ✓ | Load unsigned word via cap |
| CLD | | | ✓ | Load double via cap |
| CSB | | | ✓ | Store byte via cap |
| C.CSB | | | ✓ | Store byte via cap |
| CSH | | | ✓ | Store half via cap |
| C.CSH | | | ✓ | Store half via cap |
| CSW | | | ✓ | Store word via cap |
| CSD | | | ✓ | Store double via cap |
| AUIPCC | | | ✓ | Add immediate to PCC address, representability check |
| C.CINCOFFSET16CSP | | | ✓ | ADD immediate to stack pointer, representability check |
| C.CINCOFFSET4CSPN | | | ✓ | ADD immediate to stack pointer, representability check |
| C.CMOVE | | | ✓ | Same as CMove |
| C.CJ | | | ✓ | Jump to PC+offset, bounds check minimum size target instruction |
| C.CJAL | | | ✓ | Jump to PC+offset, bounds check minimum size target instruction, link to cd |
| CJAL | | | ✓ | Jump to PC+offset, bounds check minimum size target instruction, link to cd |
| JALR.PCC | | | ✓ | RISC-V JALR available in capability modes (with zero offset) |
| CJALR | | | ✓ | Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap |
| C.CJALR | | | ✓ | Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap |
| C.CJR | | | ✓ | Indirect cap jump, bounds check minimum size target instruction, unseal target cap |
| CBO.INVAL.CAP | | | ✓ | Cache block invalidate (implemented as clean), via cap |
| CBO.CLEAN.CAP | | | ✓ | Cache block clean, via cap |
| CBO.FLUSH.CAP | | | ✓ | Cache block flush, via cap |
| CBO.ZERO.CAP | | | ✓ | Cache block zero, via cap |
| PREFETCH.R.CAP | | | ✓ | Prefetch read-only data cache line, via cap |
| PREFETCH.W.CAP | | | ✓ | Prefetch writeable data cache line, via cap |
| PREFETCH.I.CAP | | | ✓ | Prefetch instruction cache line, via cap |
| CLR.C | | | ✓ | Load reserve cap via cap |
| CLR.D | | | ✓ | Load reserve double via cap |

| Mnemonic | Zcheri_mode | Zcheri_legacy | Zcheri_purecap | Function |
|---|---|---|---|---|
| CLR.W | | | ✔ | Load reserve word via cap |
| CLR.H | | | ✔ | Load reserve half via cap |
| CLR.B | | | ✔ | Load reserve byte via cap |
| CSC.C | | | ✔ | Store conditional cap via cap |
| CSC.D | | | ✔ | Store conditional double via cap |
| CSC.W | | | ✔ | Store conditional word via cap |
| CSC.H | | | ✔ | Store conditional half via cap |
| CSC.B | | | ✔ | Store conditional byte via cap |
| CAMOSWAP.C | | | ✔ | Atomic swap of cap via cap |
| CAMO<OP>.W | | | ✔ | Atomic op of word via cap |
| CAMO<OP>.D | | | ✔ | Atomic op of double via cap |
| C.CFLD | | | ✔ | Load floating point double via cap |
| C.CFLDSP | | | ✔ | Load floating point double via cap, sp relative |
| C.CFSD | | | ✔ | Store floating point double via cap |
| C.CFSDSP | | | ✔ | Store floating point double via cap, sp relative |
| CFLH | | | ✔ | Load floating point half via cap |
| CFSH | | | ✔ | Store floating point half via cap |
| CFLW | | | ✔ | Load floating point word via cap |
| CFSW | | | ✔ | Store floating point word via cap |
| CFLD | | | ✔ | Load floating point double via cap |
| CFSD | | | ✔ | Store floating point double via cap |
| CM.CPUSH | | | ✔ | Push capability stack frame |
| CM.CPOP | | | ✔ | Pop capability stack frame |
| CM.CPOPRET | | | ✔ | Pop capability stack frame and return |
| CM.CPOPRETZ | | | ✔ | Pop capability stack frame and return zero |
| CM.CMVSA01 | | | ✔ | Move two capability registers |
| CM.CMVA01S | | | ✔ | Move two capability registers |
| CM.CJALT | | | ✔ | Table jump and link |
| CM.CJT | | | ✔ | Table jump |
| CSH1ADD | | | ✔ | shift and add, representability check on the result |
| CSH1ADD.UW | | | ✔ | shift and add, representability check on the result |
| CSH2ADD | | | ✔ | shift and add, representability check on the result |
| CSH2ADD.UW | | | ✔ | shift and add, representability check on the result |
| CSH3ADD | | | ✔ | shift and add, representability check on the result |
| CSH3ADD.UW | | | ✔ | shift and add, representability check on the result |
| CSH4ADD | | | ✔ | shift and add, representability check on the result |
| CSH4ADD.UW | | | ✔ | shift and add, representability check on the result |

*Table 35. Instructions valid for execution in capability mode only*

| Mnemonic | Zcheri_mode | Zcheri_legacy | Zcheri_purecap | Function |
|---|---|---|---|---|
| LC | | ✓ | | Load cap via int pointer, authorise with DDC |
| SC | | ✓ | | Store cap via int pointer, authorise with DDC |
| 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 |
| C.ADDI16SP | | ✓ | | ADD immediate to stack pointer |
| C.ADDI4SPN | | ✓ | | ADD immediate to stack pointer, representability check |
| C.MV | | ✓ | | Register Move |
| JALR.CAP | | ✓ | | CJALR available in legacy mode (with zero offset) |
| DRET | | | | Return from debug mode, sets DDC from DDDC and PCC from DPCC |
| CBO.INVAL | | ✓ | | Cache block invalidate (implemented as clean), authorise with DDC |
| CBO.CLEAN | | ✓ | | Cache block clean, authorise with DDC |
| CBO.FLUSH | | ✓ | | Cache block flush, authorise with DDC |
| CBO.ZERO | | ✓ | | Cache block zero, authorise with DDC |
| PREFETCH.R | | ✓ | | Prefetch instruction cache line, always valid |
| PREFETCH.W | | ✓ | | Prefetch read-only data cache line, authorise with DDC |
| PREFETCH.I | | ✓ | | Prefetch writeable data cache line, authorise with DDC |
| LR.C | | ✓ | | Load reserve cap via int pointer, authorise with DDC |
| LR.H | | ✓ | | Load reserve half via int pointer, authorise with DDC |
| LR.B | | ✓ | | Load reserve byte via int pointer, authorise with DDC |
| SC.C | | ✓ | | Store conditional cap via int pointer, authorise with DDC |
| SC.H | | ✓ | | Store conditional half via int pointer, authorise with DDC |
| SC.B | | ✓ | | Store conditional byte via int pointer, authorise with DDC |
| AMOSWAP.C | | ✓ | | Atomic swap of cap |
| AMO<OP>.W | | ✓ | | Atomic op of word |
| AMO<OP>.D | | ✓ | | Atomic op of double |
| 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 |

| Mnemonic | Zcheri_mode | Zcheri_legacy | Zcheri_purecap | Function |
|---|---|---|---|---|
| C.FLDSP |  | ✓ |  | Load floating point double, sp relative |
| C.FSD |  | ✓ |  | Store floating point double |
| C.FSDSP |  | ✓ |  | Store floating point double, sp relative |
| 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.MVSA01 |  | ✓ |  | Move two integer registers |
| CM.MVA01S |  | ✓ |  | Move two integer registers |
| CM.JALT |  | ✓ |  | Table jump and link |
| CM.JT |  | ✓ |  | Table jump |
| SH4ADD |  | ✓ |  | shift and add |
| SH4ADD.UW |  | ✓ |  | shift and add |

*Table 36. Instructions valid for execution in legacy mode only*

| Mnemonic | Zcheri_mode | Zcheri_legacy | Zcheri_purecap | Function |
|---|---|---|---|---|
| CINCOFFSET |  |  | ✓ | Increment cap address by register, representability check |
| CINCOFFSETIMM |  |  | ✓ | Increment cap address by immediate, representability check |
| CSETADDR |  |  | ✓ | Replace capability address, representability check |
| CGETTAG |  |  | ✓ | Get tag field |
| CGETPERM |  |  | ✓ | Get hperm and uperm fields as 1-bit per permission, packed together |
| CMOVE |  |  | ✓ | Move capability register |
| CANDPERM |  |  | ✓ | AND capability permissions (expand to 1-bit per permission before ANDing) |
| CGETHIGH |  |  | ✓ | Get metadata |
| CSETHIGH |  |  | ✓ | Set metadata and clear tag |
| CSETEQUALEXACT |  |  | ✓ | Full capability bitwise compare |
| CSEAL |  |  | ✓ | Seal capability |
| CTESTSUBSET |  |  | ✓ | Set register bounds on capability with rounding, clear tag if rounding is required |
| CBUILDCAP |  |  | ✓ | Set cd to cs2 with its tag set after checking that cs2 is a subset of cs1 |
| CSETBOUNDS |  |  | ✓ | Set register bounds on capability with rounding, clear tag if rounding is required |
| CSETBOUNDSIMM |  |  | ✓ | Set immediate bounds on capability with rounding, clear tag if rounding is required |
| CSETBOUNDSINEXACT |  |  | ✓ | Set bounds on capability with rounding up as required |
| CRAM |  |  | ✓ | Representable Alignment Mask: Return mask to apply to address to get the requested bounds |
| CGETBASE |  |  | ✓ | Get capability base |
| CGETLEN |  |  | ✓ | Get capability length |
| Section 8.1.9 | ✓ |  |  | Set the mode bit of a capability, no permissions required |
| CMODESWITCH | ✓ |  |  | Directly switch mode (legacy / capability) |
| C.CMODESWITCH | ✓ |  |  | Directly switch mode (legacy / capability) |
| MRET |  |  |  | Return from machine mode handler, sets PCC from MTVECC, needs ASR permission |
| 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 |

| Mnemonic | Zcheri_mode | Zcheri_legacy | Zcheri_purecap | Function |
|---|---|---|---|---|
| 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 |

*Table 37. Instructions valid for execution in both capability and legacy modes*

# Chapter 10. Capability Width CSR Summary

| Extended 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 |
| sepcc | sepc | S-mode |
| jvtc | jvt | Zcmt |

*Table 38. CSRs extended to capability width, accessible through an alias*

| Extended CSR | Action on XLEN write | Action on CLEN write |
|---|---|---|
| dpcc | Apply Invalid address conversion. Always update the CSR with CSETADDR 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 CSETADDR. | direct write |
| dscratch1c | Update the CSR using CSETADDR. | direct write |
| mtvecc | Apply Invalid address conversion. Always update the CSR with CSETADDR 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 CSETADDR 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 CSETADDR. | direct write |
| mepcc | Apply Invalid address conversion. Always update the CSR with CSETADDR 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 |
| stvecc | Apply Invalid address conversion. Always update the CSR with CSETADDR 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 CSETADDR 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 CSETADDR. | direct write |
| sepcc | Apply Invalid address conversion. Always update the CSR with CSETADDR 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 |
| jvtc | Apply Invalid address conversion. Always update the CSR with CSETADDR 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 |

*Table 39. Action taken on writing to extended CSRs.*

* The vector range check is to ensure that vectored entry to the handler in 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 * XLENMAX bits where HICAUSE=16).

*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.*

| Extended CSR | Executable Vector | Unseal On Execution |
|---|---|---|
| dpcc | ✓ | ✓ |

| Extended CSR | Executable Vector | Unseal On Execution |
|---|---|---|
| mtvecc | ✓ | |
| mepcc | ✓ | ✓ |
| stvecc | ✓ | |
| sepcc | ✓ | ✓ |
| jvtc | ✓ | |
| pcc | ✓ | |

*Table 40. CLEN-wide CSRs storing executable vectors*

Some CSRs store executable vectors as shown in Table 40. 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.

| Extended CSR | Store full metadata |
|---|---|
| dscratch0c | ✓ |
| dscratch1c | ✓ |
| mscratchc | ✓ |
| sscratchc | ✓ |

*Table 41. CLEN-wide CSRs which store all CLEN+1 bits*

Table 41 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.

| Extended CSR | Zcheri_legacy | Zcheri_purecap | Prerequisites | CLEN Address | Permissions | Reset Value |
|---|---|---|---|---|---|---|
| dpcc | ✓ | ✓ | Sdext | 0x7b9 | DRW, ASR-permission | Infinity |
| dscratch0c | ✓ | ✓ | Sdext | 0x7ba | DRW, ASR-permission | Infinity |
| dscratch1c | ✓ | ✓ | Sdext | 0x7bb | DRW, ASR-permission | NULL |
| mtvecc | ✓ | ✓ | M-mode | 0x765 | MRW, ASR-permission | Infinity |
| mscratchc | ✓ | ✓ | M-mode | 0x760 | MRW, ASR-permission | NULL |
| mepcc | ✓ | ✓ | M-mode | 0x761 | MRW, ASR-permission | Infinity |
| stvecc | ✓ | ✓ | S-mode | 0x505 | SRW, ASR-permission | Infinity |
| sscratchc | ✓ | ✓ | S-mode | 0x540 | SRW, ASR-permission | NULL |
| sepcc | ✓ | ✓ | S-mode | 0x541 | SRW, ASR-permission | Infinity |
| jvtc | ✓ | ✓ | Zcmt | 0x417 | URW | Infinity |
| dddc | ✓ | | Sdext | 0x7bc | DRW, ASR-permission | NULL |
| mtdc | ✓ | | M-mode | 0x74c | MRW, ASR-permission | NULL |
| stdc | ✓ | | S-mode | 0x163 | SRW, ASR-permission | NULL |
| ddc | ✓ | | none | 0x416 | URW | Infinity |
| pcc | ✓ | ✓ | none | 0xcb0 | URO | Infinity (address = boot address) |

*Table 42. All CLEN-wide CSRs*

## 10.1. Other tables

| Mnemonic | Legacy mnemonic RV32 | Legacy mnemonic RV64 |
|---|---|---|
| CLC | LC | LC |
| CSC | SC | SC |
| C.CLCSP | C.FLWSP | C.FLDSP |
| C.CSCSP | C.FSWSP | C.FSDSP |
| C.CLC | C.FLW | C.FLD |

| Mnemonic | Legacy mnemonic RV32 | Legacy mnemonic RV64 |
|---|---|---|
| C.CSC | C.FSW | C.FSD |
| C.CLWSP | C.LWSP | C.LWSP |
| C.CSWSP | C.SWSP | C.SWSP |
| C.CLW | C.LW | C.LW |
| C.CSW | C.SW | C.SW |
| C.CLD | C.LD | C.LD |
| C.CSD | C.SD | C.SD |
| C.CLDSP | C.LDSP | C.LDSP |
| C.CSDSP | C.SDSP | C.SDSP |
| CLB | LB | LB |
| CLH | LH | LH |
| C.CLH | C.LH | C.LH |
| CLW | LW | LW |
| CLBU | LBU | LBU |
| C.CLBU | C.LBU | C.LBU |
| CLHU | LHU | LHU |
| C.CLHU | C.LHU | C.LHU |
| CLWU | LWU | LWU |
| CLD | LD | LD |
| CSB | SB | SB |
| C.CSB | C.SB | C.SB |
| CSH | SH | SH |
| C.CSH | C.SH | C.SH |
| CSW | SW | SW |
| CSD | SD | SD |
| AUIPCC | AUIPC | AUIPC |
| C.CINCOFFSET16CSP | C.ADDI16SP | C.ADDI16SP |
| C.ADDI4SPN | C.ADDI4SPN | C.ADDI4SPN |
| C.CINCOFFSET4CSPN | C.ADDI4SPN | C.ADDI4SPN |
| C.CMOVE | C.MV | C.MV |
| C.CJ | C.J | C.J |
| C.CJAL | C.JAL | C.JAL |
| CJAL | JAL | JAL |
| JALR.CAP | JALR.PCC | JALR.PCC |
| CJALR | JALR | JALR |
| C.CJALR | C.JALR | C.JALR |
| C.CJR | C.JR | C.JR |
| CBO.INVAL.CAP | CBO.INVAL | CBO.INVAL |
| CBO.CLEAN.CAP | CBO.CLEAN | CBO.CLEAN |
| CBO.FLUSH.CAP | CBO.FLUSH | CBO.FLUSH |
| CBO.ZERO.CAP | CBO.ZERO | CBO.ZERO |
| PREFETCH.R.CAP | PREFETCH.R | PREFETCH.R |
| PREFETCH.W.CAP | PREFETCH.W | PREFETCH.W |
| PREFETCH.I.CAP | PREFETCH.I | PREFETCH.I |
| CLR.C | LR.C | LR.C |
| CLR.D | LR.D | LR.D |
| CLR.W | LR.W | LR.W |
| CLR.H | LR.H | LR.H |
| CLR.B | LR.B | LR.B |

| Mnemonic | Legacy mnemonic RV32 | Legacy mnemonic RV64 |
|---|---|---|
| CSC.C | SC.C | SC.C |
| CSC.D | SC.D | SC.D |
| CSC.W | SC.W | SC.W |
| CSC.H | SC.H | SC.H |
| CAMOSWAP.C | AMOSWAP.C | AMOSWAP.C |
| CAMO<OP>.W | AMO<OP>.W | AMO<OP>.W |
| CAMO<OP>.D | AMO<OP>.D | AMO<OP>.D |
| CFLH | FLH | FLH |
| CFSH | FSH | FSH |
| CFLW | FLW | FLW |
| CFSW | FSW | FSW |
| CFLD | FLD | FLD |
| CFSD | FSD | FSD |
| CM.CPUSH | CM.PUSH | CM.PUSH |
| CM.CPOP | CM.POP | CM.POP |
| CM.CPOPRET | CM.POPRET | CM.POPRET |
| CM.CPOPRETZ | CM.POPRETZ | CM.POPRETZ |
| CM.CMVSA01 | CM.MVSA01 | CM.MVSA01 |
| CM.CMVA01S | CM.MVA01S | CM.MVA01S |
| CM.CJALT | CM.JALT | CM.JALT |
| CM.CJT | CM.JT | CM.JT |
| CSH1ADD | SH1ADD | SH1ADD |
| CSH1ADD.UW | SH1ADD.UW | SH1ADD.UW |
| CSH2ADD | SH2ADD | SH2ADD |
| CSH2ADD.UW | SH2ADD.UW | SH2ADD.UW |
| CSH3ADD | SH3ADD | SH3ADD |
| CSH3ADD.UW | SH3ADD.UW | SH3ADD.UW |
| CSH4ADD | SH4ADD | SH4ADD |
| CSH4ADD.UW | SH4ADD.UW | SH4ADD.UW |

*Table 43. Mnemonics with the same encoding but mapped to different instructions in Legacy and Capability Mode*

| Mnemonic | Function |
|---|---|
| LC | Load cap via int pointer, authorise with DDC |
| SC | Store cap via int pointer, authorise with DDC |
| CLC | Load cap via cap |
| CSC | Store cap via cap |
| C.CLCSP | Load cap via cap, SP relative |
| C.CSCSP | Store cap via cap, SP relative |
| C.CLC | Load cap via cap |
| C.CSC | Store cap via cap |
| LR.C | Load reserve cap via int pointer, authorise with DDC |
| CLR.C | Load reserve cap via cap |
| SC.C | Store conditional cap via int pointer, authorise with DDC |
| CSC.C | Store conditional cap via cap |
| AMOSWAP.C | Atomic swap of cap |
| CAMOSWAP.C | Atomic swap of cap via cap |

*Table 44. Instruction encodings which vary depending on the current XLEN*

*CMODESWITCH and Section 8.1.9 only exist in capability mode if legacy mode is also* *present. A purecap core does not implement the mode bit in the capability.*

| Mnemonic | illegal insn if (1) | OR illegal insn if (2) | OR illegal insn if (3) |
|---|---|---|---|
| CMODESWITCH | mode==D (optional) | | |
| C.CMODESWITCH | mode==D (optional) | | |
| C.CJ | mode==D (optional) | | |
| C.CJAL | mode==D (optional) | | |
| CJAL | mode==D (optional) | | |
| JALR.CAP | mode==D (optional) | | |
| JALR.PCC | mode==D (optional) | | |
| CJALR | mode==D (optional) | | |
| C.CJALR | mode==D (optional) | | |
| C.CJR | 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.INVAL | 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 | |
| CBO.INVAL.CAP | MODE<M AND menvcfg.CBIE[0]==0 | MODE<S AND senvcfg.CBIE[0]==0 | |
| CBO.CLEAN.CAP | MODE<M AND menvcfg.CBIE[0]==0 | MODE<S AND senvcfg.CBIE[0]==0 | |
| CBO.FLUSH.CAP | MODE<M AND menvcfg.CBIE[0]==0 | MODE<S AND senvcfg.CBIE[0]==0 | |
| CBO.ZERO.CAP | 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.FSD | Xstatus.fs==0 | | |
| C.FSDSP | Xstatus.fs==0 | | |
| C.CFLD | Xstatus.fs==0 | | |
| C.CFLDSP | Xstatus.fs==0 | | |
| C.CFSD | Xstatus.fs==0 | | |
| C.CFSDSP | Xstatus.fs==0 | | |
| CFLH | Xstatus.fs==0 | | |
| CFSH | Xstatus.fs==0 | | |
| CFLW | Xstatus.fs==0 | | |
| CFSW | Xstatus.fs==0 | | |
| CFLD | Xstatus.fs==0 | | |
| CFSD | Xstatus.fs==0 | | |

*Table 45. Illegal instruction detect for CHERI instructions*

# 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.