RISC-V Specification for CHERI Extensions

Authors: Thomas Aird, Hesham Almatary, Andres Amaya Garcia, John Baldwin, Paul Buxton, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Franz A. Fuchs, Timothy Hutt, Alexandre Joannou, Martin Kaiser, Tariq Kurd, Ben Laurie, Marno van der Maas, Maja Malenko, 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, Ricki Tura, Robert N. M. Watson, Toby Wenman, Jonathan Woodruff, Jason Zhijingcheng Yu

Version v0.9.1, 2024-11-13: 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:

- Thomas Aird <thomas.aird@codasip.com>
- Hesham Almatary <hesham.almatary@cl.cam.ac.uk>
- Andres Amaya Garcia <andres.amaya@codasip.com>
- John Baldwin <jhb61@cl.cam.ac.uk>
- Paul Buxton <paul.buxton@codasip.com>
- David Chisnall <david.chisnall@cl.cam.ac.uk>
- Jessica Clarke <jessica.clarke@cl.cam.ac.uk>
- Brooks Davis <brooks.davis@sri.com>
- Lawrence Esswood <lesswood@google.com>
- Nathaniel Wesley Filardo <nwf20@cam.ac.uk>
- Franz A. Fuchs <franz.fuchs@cl.cam.ac.uk>
- Timothy Hutt <timothy.hutt@codasip.com>
- Alexandre Joannou <alexandre.joannou@cl.cam.ac.uk>
- Martin Kaiser <martin.kaiser@codasip.com>
- Tariq Kurd <tariq.kurd@codasip.com>
- Ben Laurie <benl@google.com>
- Marno van der Maas <mvdmaas@lowrisc.org>
- Maja Malenko <maja.malenko@codasip.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>
- Ricki Tura <ricki.tura@codasip.com>
- Robert N. M. Watson <robert.watson@cl.cam.ac.uk>
- Toby Wenman <toby.wenman@codasip.com>

- Jonathan Woodruff <jonathan.woodruff@cl.cam.ac.uk>
- Jason Zhijingcheng Yu <yu.zhi@comp.nus.edu.sg>

Chapter 1. Quick Start

This document describes the RISC-V extensions for supporting CHERI capabilities in hardware. Capabilities can be used to provide memory safety, mitigating up to 70% of memory safety issues (Joly et al., 2020), as well as to provide efficient compartmentalisation. The extensions are split into the core features required for a working capability system (Zcheripurecap), and features required to support a mix-and-match of binaries compiled for CHERI and unchanged binaries (Zcherihybrid). Some other smaller extensions are described that provide additional functionality relevant to CHERI.

1.1. Capability Properties

Capabilities are 2*XLEN (which we call CLEN) bit structures, containing all the information required to identify and authorise access to a region of memory. This includes:

- An XLEN bit address, describing where the capability currently points.
- Bounds: a *base* and a *top* address, describing the range of addresses the capability can be used to access.
- Permissions (read, write, execute, read capability, ...) describing the kinds of accesses the capability can be used for.
- Sealing information: a capability can be *sealed*, restricting it to only be used or modified in particular ways.

A one-bit integrity tag is stored alongside a capability: this is maintained by hardware and cannot be directly modified by software. It indicates whether the capability is valid. An initial Infinite capability with access to all of memory with all permissions is provided in system registers on reset: all valid capabilities are derived from it. This is the only way to obtain a valid capability: no software, even machine mode, can *forge* a capability.

1.2. Added State

A CHERI core adds state to allow capabilities to be used from within registers, and to ensure they are not corrupted as they flow through the system. This means the following state is added:

- Metadata within architectural registers: XLEN-wide integer registers (e.g. `sp`, `a0`) are all extended with another XLEN bits of capability metadata, including bounds and permissions. The resulting CLEN bits in full form a capability, and we refer to the same register prefixed with a `c`, i.e. `csp`, `ca0`. The integer part of the register is interpreted as the address field of the capability. The zero register is extended with zero metadata and a cleared tag: this is called the NULL capability. As well as general purpose registers, system registers that store addresses are extended to contain capabilities. For example, mtvec is extended to a capability version mtvecc (the machine trap vector capability) to allow the code bounds to be changed on an exception.
- Tags in registers, caches, and memory:
  - Every register has a one-bit tag, indicating whether the capability in the register is valid to be dereferenced. This tag is cleared if the register is written as an integer.
  - The tags are also tracked through the memory subsystem: every aligned CLEN-bits wide region has a non-addressable one-bit tag, which the hardware manages atomically with the data. The tag is cleared if the memory region is ever written other than using a capability store from a tagged capability register. Any caches must preserve this abstraction.

1.3. Checking Memory

Every memory access performed by a CHERI core must be authorised by a capability. It is explicitly defined for every instruction where to find the capability to check against. In *purecap* code, where all pointers are individual capabilities, the capability and address are used together, so e.g. `lw t0, 16(csp)` loads a word from memory, getting the address and bounds from the `csp` register. For code that has not yet been fully adapted to CHERI (*hybrid* code), the processor can run in a pointer mode (not to be confused with a privilege mode) where the authorising capability is instead taken from a special CSR: the default data capability (ddc).

Instruction fetch is also authorised by a capability: the program counter capability (pcc) which extends PC. This allows code fetch to be bounded, preventing a wide range of attacks that subvert control flow with integer data. Where Zcherihybrid is supported, the pcc also contains the mode bit indicating whether the processor is running in integer or capability pointer mode. Changing the bounds used for instruction fetch or the pointer mode can be as easy as performing a capability-based jump (JALR in capability pointer mode). MODESW.CAP and MODESW.INT instructions are also added to allow cheap mode switching.

Exception codes are added for CHERI-specific exceptions on fetch, jumps, and memory access. No other exception paths are added: in particular, capability manipulations do not trap, but may clear the tag on the result capability if the operation is not permitted.

1.4. Added Instructions

The added instructions can be split into the following categories:

- Capability manipulations (e.g. CADD, SCBNDS): for security, capabilities can only be modified in restricted ways. Special instructions are provided to perform these allowed operations, for example *shrinking* the bounds or *reducing* the permissions. Any attempt to manipulate capabilities without using the instructions clears the tag, rendering them unusable for accessing memory.
- Capability inspection (e.g. GCBASE, GCPERM): capability fields (for example the *bounds* describing what addresses the capability gives access to) are stored compressed in registers and memory. These instructions give convenient access to allow software to query them.
- Memory access instructions (e.g. LC, SC): capabilities must be read from and written to memory atomically along with their tag. Instructions are added to perform these wider accesses, allowing capability flow between the memory and the register file.

1.5. Existing Instructions

Existing RISC-V instructions are largely unmodified: in *Integer Pointer Mode*, there is binary compatibility. Instructions that access memory, as well as branches and jumps, are automatically checked against ddc and pcc, raising an exception if the checks fail. However, ddc and pcc are reset to Infinite capabilities, meaning the checks will always pass on systems that have not written to CHERI system registers.

In *Capability Pointer Mode*, these instructions are instead modified to check against the full capability from the address register (e.g. `lw t0, 16(csp)`). In some cases, they are also changed to return a full capability value, e.g. AUIPC will return the full pcc including the metadata.

Chapter 2. Introduction

2.1. 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) that 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
- *type* which is used to support higher-level software encapsulation

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

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

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

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

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

2.2. 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:

Zcheripurecap

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

Zcherihybrid

Extends Zcheripurecap with features to ensure that the ISA extended with CHERI allows backwards binary compatibility with RISC-V.

Zish4add

Addition of SH4ADD and SH4ADD.UW for RV64 only, as CHERI capabilities are 16 bytes when XLEN=64

Zabhlrsc

Addition of LR.B, LR.H, SC.B, SC.H for more accurate atomic locking as the memory ranges are restricted by using bounds, therefore precise locking is needed.

Zcheripte

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

Zstid

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

Zcherilevels

Extension for supporting capability flow control. This extension allows limiting storing of capabilities to specific regions and can be used e.g. for safer data sharing between compartments.

The extension names are provisional and subject to change.

Table 1. Extension status and summary

Extension	Status	Comment
Zcheripurecap	Stable	This extension is a candidate for freezing
Zcherihybrid	Stable	This extension is a candidate for freezing

Extension	Status	Comment
Zish4add	Stable	This extension is a candidate for freezing
Zabhlrsc	Stable	This extension is a candidate for freezing
Zcheripte	Prototype	This extension is a prototype, software is being developed to use it to increase the maturity level
Zstid	Prototype	This extension is a prototype, software is being developed to use it to increase the maturity level
Zcherilevels with `LVLBITS=1`	Prototype	This extension is a prototype, software is being developed to use it to increase the maturity level.

Zcheripurecap is defined as the base extension which all CHERI RISC-V implementations must support. Zcherihybrid and Zcheripte are optional extensions in addition to Zcheripurecap.

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 Zcheripurecap. 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 Zcheripurecap and Zcherihybrid.

See Chapter 12 for compatibility with other RISC-V extensions.

2.3. Risks and Known Uncertainty

- All extensions could be divided up differently in the future, including after ratification
- The RISC-V Architecture Review Committee (ARC) are likely to update all encodings
- The ARC are likely to update all CSR addresses
- Instruction mnemonics may be renamed
  - Any changes will affect assembly code, but assembler aliases can provide backwards compatibility

2.3.1. Partially Incompatible Extensions

There are RISC-V extensions in development that may duplicate some aspects of CHERI functionality or directly conflict with CHERI and should only be available in *Integer Pointer Mode* on a CHERI-enabled hart. These include:

- RISC-V CFI specification
- "J" Pointer Masking (see Chapter 10).

Chapter 3. Anatomy of Capabilities in Zcheripurecap

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

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

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

We briefly note that the capability encoding described in this section could be replaced with an entirely different design without changing how CHERI integrates with the RISC-V ISA. In particular, this capability encoding specification was designed to run software initially ported to CHERIv9 while providing spatial safety, temporal safety and compartmentalization support alongside a good measure of compatibility with RISC-V software that is not aware of CHERI. Alternative capability encoding specifications must provide key primitives, such as permissions and bounds, from this specification while using a different encoding that, for example, changes the granularity of bounds or adds new features. For simplicity of expression, the text is written as if this was the only possible capability encoding for CHERI RISC-V.

3.1. Capability Encoding

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

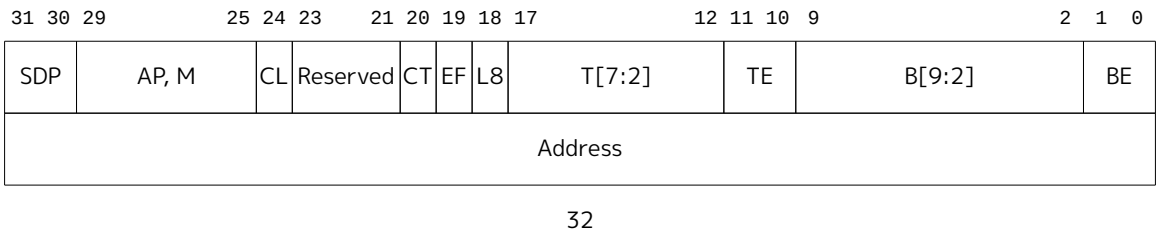


Figure 1. Capability encoding for MXLEN=32

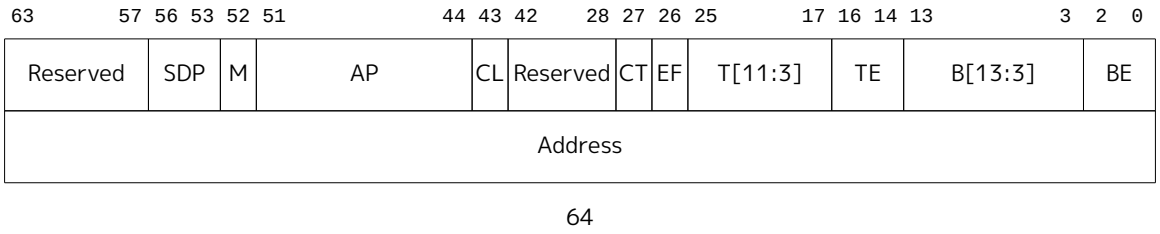


Figure 2. Capability encoding for MXLEN=64

Reserved bits are available for future extensions to Zcheripurecap.

Reserved bits must be 0 in tagged capabilities.

The CL field is only present if Zcherilevels is implemented, otherwise it is reserved.

3.2. Components of a Capability

Capabilities contain the software accessible fields described in this section.

3.2.1. Tag

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

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

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

3.2.2. Address

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

Table 2. Address widths depending on MXLEN

MXLEN	Address width
32	32
64	64

3.2.3. Architectural Permissions (AP)

Description

This field encodes architecturally defined permissions of the capability. Permissions grant access subject to the tag being set, the capability being unsealed (see Section 3.2.5), and bounds checks (see Section 3.2.6). An operation is also contingent on requirements imposed by other RISC-V architectural features, such as virtual memory, PMP and PMAs, even if the capability grants sufficient permissions.

The permissions currently defined in Zcheripurecap are listed below.

Read Permission (R)

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

Write Permission (W)

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

Capability Permission (C)

Allow reading capability data from memory if the authorising capability also grants R-permission. Allow writing capability data to memory if the authorising capability also grants W-permission.

Execute Permission (X)

Allow instruction execution.

Load Mutable Permission (LM)

Allow preserving the W-permission of capabilities loaded from memory. If a capability grants R-permission and C-permission, but no LM-permission, then a capability loaded via this authorizing capability will have W-permission and LM-permission removed provided that the loaded capability has its tag set and is not sealed; loaded capabilities that are sealed or untagged do not have their permissions changed. The rules specified by ACPERM are followed when W-permission and LM-permission are removed, so additional permissions may also be removed. Clearing a capability's LM-permission and W-permission allows sharing a read-only version of a data structure (e.g. a tree or linked list) without making a copy.

Implementations are allowed to retain invalid capability permissions loaded from memory instead of following the ACPERM behaviour of reducing them to no permissions.

Access System Registers Permission (ASR)

Allow read and write access to all privileged (M-mode and S-mode) CSRs. If Zstid is supported the utid, utidc, stid, stidc, mtid, mtidc registers are all considered privileged for the purposes of writing and unprivileged for reading, and thus require ASR-permission for writes but not reads. In all cases a suitable privilege mode is required for access.

Permission Encoding

The bit width of the permissions field depends on the value of MXLEN as shown in Table 3. A 5-bit vector encodes the permissions when MXLEN=32. For this case, the legal encodings of permissions are listed in Table 4. Certain combinations of permissions are impractical. For example, C-permission is superfluous when the capability does not grant either R-permission or W-permission. Therefore, it is only possible to encode a subset of all combinations.

Table 3. Permissions widths depending on MXLEN

MXLEN	AP field width	Comment
32	5	Encodes some combinations of 6 permission bits, including the M-bit if Zcherihybrid is supported.
64	6	Separate bits for each architectural permission.

if Zcherilevels is supported then there are 8 architectural permission bits.

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

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

Table 4. Encoding of architectural permissions for MXLEN=32

Encoding[2:0]	R	W	C	LM	X	ASR	Mode[1]	Notes
Quadrant 0: Non-capability data read/write								
bit[2] - write, bit[1] - reserved (0), bit[0] - read								
Reserved bits for future extensions are 0 so new permissions are not implicitly granted								
0							N/A	No permissions
1	✔						N/A	Data RO
2-3	reserved							
4		✔					N/A	Data WO
5	✔	✔					N/A	Data RW
6-7	reserved							
Quadrant 1: Executable capabilities								
bit[0] - M-bit (0-*Capability Pointer Mode*, 1-*Integer Pointer Mode*)								
0-1	✔	✔	✔	✔	✔	✔	Mode[1]	Execute + ASR (see Infinite)
2-3	✔		✔	✔	✔		Mode[1]	Execute + Data & Cap RO
4-5	✔	✔	✔	✔	✔		Mode[1]	Execute + Data & Cap RW
6-7	✔	✔			✔		Mode[1]	Execute + Data RW
Quadrant 2: Restricted capability data read/write								
R and C implicitly granted, LM dependent on W permission.								
Reserved bits for future extensions must be 1 so they are implicitly granted								
bit[2] is reserved to mean write for future encodings								
0-2	reserved							
3	✔		✔				N/A	Data & Cap RO (no LM)
4-7	reserved							
Quadrant 3: Capability data read/write								
bit[2] - write, R and C implicitly granted.								
Reserved bits for future extensions must be 1 so they are implicitly granted								
0-2	reserved							
3	✔		✔	✔			N/A	Data & Cap RO
4-6	reserved							
7	✔	✔	✔	✔			N/A	Data & Cap RW

[1] *Mode (M-bit) can only be set on a tagged capability when Zcherihybrid is supported. Despite being encoded here it is* ***not*** *an architectural permission.*

When MXLEN=32 there are many reserved permission encodings (see Table 4). It is not possible for a tagged capability to have one of these values since ACPERM will never create it. It is possible for untagged capabilities to have reserved values. GCPERM will

> *interpret reserved values as if it were 0b00000 (no permissions). Future extensions may assign meanings to the reserved bit patterns, in which case GCPERM is allowed to report a non-zero value.*

A 6-bit vector encodes the permissions when MXLEN=64 (8-bit if Zcherilevels is supported). In this case, there is a bit per permission as shown in Table 5. A permission is granted if its corresponding bit is set, otherwise the capability does not grant that permission.

Table 5. Encoding of architectural permissions for MXLEN=64

Bit	Name
0	C-permission
1	W-permission
2	R-permission
3	X-permission
4	ASR-permission
5	LM-permission
6	EL-permission[1]
7	SL-permission[1]

[1] This permission is only supported if the implementation supports Zcherilevels.

The M-bit is only assigned meaning when the implementation supports Zcherihybrid *and* X-permission is set.

1. For MXLEN=64, the bit assigned to the M-bit must be zero if X-permission isn't set.
2. For MXLEN=32, the M-bit is only encoded in quadrant 1 and does *not* exist in the other quadrants.

Permission Transitions

Executing ACPERM can result in sets of permissions which cannot be represented when MXLEN=32 (see Table 4) or permission combinations which are not useful for MXLEN=64, such as ASR-permission set without X-permission.

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

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

3.2.4. Software-Defined Permissions (SDP)

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

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

Table 6. SDP widths depending on MXLEN

MXLEN	SDPLEN
32	2
64	4

3.2.5. Capability Type (CT) Bit

This bit indicates the type of the capability: it is a sealed capability if the bit is 1 or unsealed if it is 0.

Sealed capabilities (`CT ≠ 0`) cannot be dereferenced to access memory and are immutable such that modifying any of its fields clears the tag of the output capability.

Sealed capabilities might be useful to software as tokens that can be passed around. The only way of clearing the type bit of a capability is by rebuilding it via a superset capability with CBLD. Zcheripurecap does not offer an unseal instruction.

The Capability Level (CL) field can be reduced even if the capability is sealed, see Table 30.

For code capabilities, the sealing bit is used to implement immutable capabilities that describe function entry points, known as sealed entry (sentry) capabilities. Such capabilities can be leveraged to establish a form of control-flow integrity between mutually distrusting code. A program may jump to a sentry capability to begin executing the instructions it references. A JALR instruction with zero offset automatically unseals a sentry target capability and installs it in the program counter capability (see Section 4.2). The jump-and-link instructions also seal the return address capability which serves as an entry point the callee can return to but cannot use to authorize memory loads or stores.

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

Concept

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 3.1.

The bounds field has the following components:

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

The bit width of T and B are defined in terms of the mantissa width (MW) which is set depending on the value of MXLEN as shown in Table 7.

Table 7. Mantissa width (MW) values depending on MXLEN

MXLEN	MW
32	10
64	14

The exponent E indicates the position of T and B within the capability's address as described in Section 3.1. The bit width of the exponent (EW) is set depending on the value of MXLEN. The maximum value of the exponent is calculated as follows:

```
CAP_MAX_E = MXLEN - MW + 2
```

The possible values for EW and CAP_MAX_E are shown in Table 8.

Table 8. Exponent widths and CAP_MAX_E depending on MXLEN

MXLEN	EW	CAP_MAX_E
32	5	24
64	6	52

The address and bounds must be representable in valid capabilities i.e. when the tag is set (see Section 3.2.6.3).

Decoding

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

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

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

If EF = 1:
    E               = 0
    T[EW / 2 - 1:0] = TE
    B[EW / 2 - 1:0] = BE
    LCout           = (T[MW - 3:0] < B[MW - 3:0]) ? 1 : 0
    LMSB            = (MXLEN == 32) ? L8 : 0
else:
    E               = CAP_MAX_E - ( (MXLEN == 32) ? { L8, TE, BE } : { TE, BE } )
    T[EW / 2 - 1:0] = 0
    B[EW / 2 - 1:0] = 0
    LCout           = (T[MW - 3:EW / 2] < B[MW - 3:EW / 2]) ? 1 : 0
    LMSB            = 1
```

Reconstituting the top two bits of T:

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

Decoding the bounds:

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

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

10.

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

Table 9. Calculation of top address correction

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

Table 10. Calculation of base address correction

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

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

The EF bit selects between two cases:

1. EF = 1: The exponent is 0 for regions less than 2^{MW-2} bytes long. L_8 is used to encode the MSB of the length and is added to B along with T[MW-3:0] to form the decoded top.
2. EF = 0: The exponent is *internal* with E stored in the lower bits of T and B along with L_8 when MXLEN=32. E is chosen so that the most significant non-zero bit of the length of the region aligns with T[MW - 2] in the decoded top. Therefore, the most significant two bits of T can be derived from B using the equality `T = B + L`, where L[MW - 2] is known from the values of EF and E and a carry out is implied if `T[MW - 3:0] < B[MW - 3:0]` since it is guaranteed that the top is larger than the base.

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

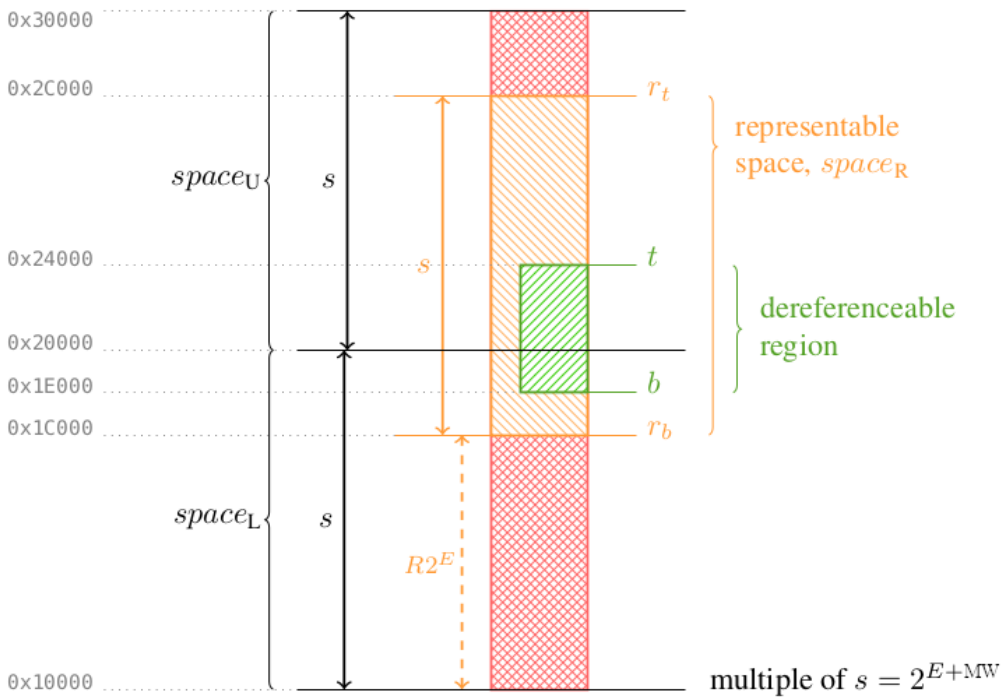


Figure 3. Memory address bounds encoded within a capability

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

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

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

A capability has infinite bounds if E=CAP_MAX_E and it is not malformed (see Section 3.2.6.3); this check is equivalent to b=0 and $t \geq 2^{MXLEN}$.

Malformed Capability Bounds

A capability is *malformed* if its bounds cannot be correctly decoded. The following check indicates whether a capability is malformed. `enableL8` is true when MXLEN=32 and false otherwise, indicating whether the `L8` bit is available for extra precision when `EF=1`.

```
malformedMSB =  (E == CAP_MAX_E     && B         != 0)
             || (E == CAP_MAX_E - 1 && B[MW - 1] != 0)
malformedLSB =  (E  < 0) || (E == 0 && enableL8)
malformed    =  !EF && (malformedMSB || malformedLSB)
```

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

CHERI enforces the following invariants for all valid (i.e., tagged) capabilities:

1. The bounds are not malformed.
2. No reserved bit in the capability encoding is set.

A tagged capability that violates those invariants (i.e., a tagged but malformed capability or a tagged capability with any reserved bit set) can only possibly be caused by a logic or memory fault (e.g., bit flipping).

Capabilities with malformed bounds:

1. Return both base and top bounds as zero, which affects instructions like GCBASE.
2. Cause certain manipulation instructions like CADDI to always clear the tag of the result.

See specific instruction pages for full details of the effect of malformed capabilities.

3.3. Special Capabilities

3.3.1. NULL Capability

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

Table 11. Field values of the NULL capability

Field	Value	Comment
Tag	zero	Capability is not valid
SDP	zeros	Grants no permissions
AP	zeros	Grants no permissions
M	zero	No meaning since non-executable (MXLEN=64 only)
CL	zero[1]	*Local*
CT	zero	Unsealed
EF	zero	Internal exponent format
L_8	zero	Top address reconstruction bit (MXLEN=32 only)
T	zeros	Top address bits
T_E	zeros	Exponent bits
B	zeros	Base address bits
B_E	zeros	Exponent bits
Address	zeros	Capability address
Reserved	zeros	All reserved fields

[1] This field only exists if Zcherilevels is implemented.

3.3.2. Infinite Capability

The Infinite capability grants all permissions while its bounds also cover the whole address space. It includes X-permission and so includes the M-bit if Zcherihybrid is supported.

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

Table 12. Field values of the Infinite capability

Field	Value	Comment
Tag	one	Capability is valid
SDP	ones	Grants all permissions
AP (MXLEN=32)	0x8/0x9[1] (see Table 4)	Grants all permissions
AP (MXLEN=64)	0xFF (see Table 5)	Grants all permissions
CL	one[2]	*Global*
CT	zero	Unsealed
EF	zero	Internal exponent format
L_8	zero	Top address reconstruction bit (MXLEN=32 only)
T	zeros	Top address bits
T_E	zeros	Exponent bits
B	zeros	Base address bits
B_E	zeros	Exponent bits
Address	zeros	Capability address
Reserved	zeros	All reserved fields

[1]If Zcherihybrid is supported, then the Infinite capability must represent *Integer Pointer Mode* for compatibility with standard RISC-V code. Therefore:

- For MXLEN=32, the M-bit is set to 1 in the AP field, giving the value 0x9
- For MXLEN=64, the M-bit is set to 1 in a separate M field which is *not shown* in the table above.

[2] This field only exists if Zcherilevels is implemented.

3.4. Representable Range Check

3.4.1. Concept

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

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

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

- Instructions such as CADD which include pointer arithmetic.

- The SCADDR instruction which updates the capability address field.

3.4.2. Practical Information

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

- Upper bits from the address
  - This is only if the other sections do not fill the available bits (E + MW ≤ MXLEN)
- Middle bits from T and B decoded from the metadata
- Lower bits are set to zero
  - This is only if there is an internal exponent (EF=0)

Table 13. Composition of the decoded top address bound

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

The top described by Table 13 is MXLEN+1 bits wide to allow capabilities to span the whole address space. The address is zero-extended by one bit. The malformed check (see Section 3.2.6.3) ensures that the top never overflows into MXLEN+2 bits and that the base never overflows into MXLEN+1 bits.

The *representable range* defines the range of addresses which do not corrupt the bounds encoding. The encoding was first introduced in Section 3.1, and is repeated in a different form in Table 13 to aid this description.

For the address to be valid for the current bounds encoding, the value in the *Upper Section* of Table 13 *must not change* as this will change the meaning of the bounds.

This gives a range of $s=2^{E+MW}$, as shown in Figure 3.

The gap between the object bounds and the bound of the representable range is always guaranteed to be at least 1/4 of s. This is represented by $R = B - 2^{MW-2}$ in Section 3.1. This gives useful guarantees, such that if an executed instruction is in pcc bounds, then it is also guaranteed that the next linear instruction is *representable*.

Chapter 4. Integrating Zcheripurecap with the RISC-V Base Integer Instruction Set

Zcheripurecap 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, Zcheripurecap 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 Zcheripurecap is compatible with RV32E.

RV128 is not currently supported by any CHERI extension.

In line with the base RISC-V ISA, the unprivileged component with its corresponding Zcheripurecap changes as described in this chapter can be used with an entirely different privileged-level design. The changes for the privileged component described in this chapter are designed to support existing popular operating systems, and assume the standard privileged architecture specified in the RISC-V ISA.

4.1. Memory

A hart supporting Zcheripurecap 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 Range described in Section 3.1 is also circular, so address 0 is within the Representable Range of a capability where address 2^{MXLEN} - 1 is within the bounds. However, the decoded top field of a capability is MXLEN + 1 bits wide and does **not** wrap, so a capability with base 2^{MXLEN} - 1 and top 2^{MXLEN} + 1 is not a subset of the Infinite capability and does not authorise access to the byte at address 0. Like malformed bounds (see Section 3.2.6.3), it is impossible for a CHERI core to generate a tagged capability with top > 2^{MXLEN}. If such a capability exists then it must have been caused by a logic or memory fault. Unlike malformed bounds, the top overflowing is not treated as a special case in the architecture: normal bounds check rules should be followed.

4.2. Programmer's Model for Zcheripurecap

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

The tag of the unprivileged **c** registers must be reset to zero. The reset values of the metadata and

address fields are UNSPECIFIED for all unprivileged **c** registers except **c0**.

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

4.2.1. PCC - The Program Counter Capability

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

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

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

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

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

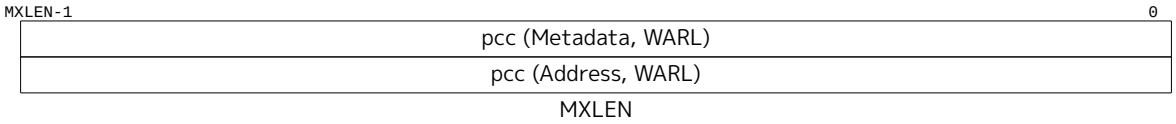


Figure 4. Program Counter Capability

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

4.3. Capability Instructions

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

4.3.1. Capability Inspection Instructions

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

- GCTAG: inspects the tag of the input capability. The output is 1 if the tag is set and 0 otherwise

- GCPERM: outputs the architectural (AP) and software-defined (SDP) permission fields of the input capability
- GCTYPE: outputs the type (e.g. unsealed or sentry) of the input capability
- GCBASE: outputs the expanded base address of the input capability
- GCLEN: outputs the length of the input capability. Length is defined as `top - base`. The output is 2^{MXLEN}-1 when the capability's length is 2^{MXLEN}
- CRAM: outputs the nearest bounds alignment that a valid capability can represent
- GCHI: outputs the compressed capability metadata
- SCEQ: compares two capabilities including tag, metadata and address
- SCSS: tests whether the bounds and permissions of a capability are a subset of those from another capability

GCBASE and GCLEN output 0 when a capability with malformed bounds is provided as an input (see Section 3.2.6.3).

4.3.2. Capability Manipulation Instructions

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

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

4.3.3. Capability Load and Store Instructions

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

A store capability instruction, SC, writes CLEN bits and the tag in a **c** register to memory. The capability authorising the memory access is provided in a **c** source register, so the effective address is

obtained by incrementing that capability with the sign-extended 12-bit offset.

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

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

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

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

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

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

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

4.4. Existing RISC-V Instructions

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

4.4.1. Integer Computational Instructions

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

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

4.4.2. Control Transfer Instructions

Control transfer instructions operate as described in the base RISC-V ISA. They also may cause metadata updates and/or cause exceptions in addition to the base behaviour as described below.

Unconditional Jumps

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

JALR allows unconditional, indirect 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 offset, then setting the least significant bit of the result to zero. The target capability is unsealed if it is a sentry with zero offset. The capability with the address of the instruction following the jump is sealed and written to a **c** register.

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

JALR causes a CHERI exception when:

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

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

Conditional Branches

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

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

4.4.3. 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 Zcheripurecap: the capability authorising the access is in the **c** register operand and the memory address is given by incrementing the address of that capability by the sign-extended 12-bit immediate offset.

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

following checks:

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

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

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

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

Zcheripurecap requires that RISC-V CSRs intended to hold addresses, like mtvec, are now able to hold capabilities. Therefore, such registers are renamed and extended to CLEN-bit in Zcheripurecap.

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

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

The CLEN-bit CSRs are summarised in Appendix C.

4.5.1. CSR Instructions

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 48.

When CSRRS and CSRRC instructions are accessing a capability width CSR, such as mtvecc, then the destination operand is a **c** register and the source operand is an **x** register. Therefore, the instructions atomically read CLEN bits from the CSR, calculate the final address using standard RISC-V behaviour (set bits, clear bits, etc.), and that final address is written to the CSR capability's address field. The update typically uses the semantics of a SCADDR instruction which clears the tag if the capability is sealed, or if the updated address is not representable. Table 48 shows the exact action taken for each capability width CSR.

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

All CSR instructions cause CHERI exceptions if the pcc does not grant ASR-permission and the CSR

accessed is privileged.

4.6. Control and Status Registers (CSRs)

Zcheripurecap extends the CSRs listed in Table 14, Table 15, Table 16, Table 17 and Table 18 from the base RISC-V ISA and its extensions. The CSRs are renamed to reflect the fact that they are extended to CLEN+1 bits wide, as the **x** registers are renamed to **c** registers.

Table 14. Renamed debug-mode CSRs in Zcheripurecap

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

Table 15. Renamed machine-mode CSRs in Zcheripurecap

Zcheripurecap CSR	Address	Extended CSR	Prerequisites	Permissions	Description
mtvecc	0x305	mtvec	M-mode	MRW, ASR-permission	Machine Trap-Vector Base-Address Capability
mscratchc	0x340	mscratch	M-mode	MRW, ASR-permission	Machine Scratch Capability
mepcc	0x341	mepc	M-mode	MRW, ASR-permission	Machine Exception Program Counter Capability
mtidc	0x780	mtid	Zstid	Read: M, Write: M, ASR-permission	Machine thread ID

Table 16. Renamed supervisor-mode CSRs in Zcheripurecap

Zcheripurecap CSR	Address	Extended CSR	Prerequisites	Permissions	Description
stvecc	0x105	stvec	S-mode	SRW, ASR-permission	Supervisor Trap-Vector Base-Address Capability
sscratchc	0x140	sscratch	S-mode	SRW, ASR-permission	Supervisor Scratch Capability
sepcc	0x141	sepc	S-mode	SRW, ASR-permission	Supervisor Exception Program Counter Capability
stidc	0x580	stid	Zstid	Read: S, Write: S, ASR-permission	Supervisor thread ID

Table 17. Renamed virtual supervisor-mode CSRs in Zcheripurecap

Zcheripurecap CSR	Address	Extended CSR	Prerequisites	Permissions	Description
vstvecc	0x205	vstvec	H	HRW, ASR-permission	Virtual Supervisor Trap-Vector Base-Address Capability

Zcheripurecap CSR	Address	Extended CSR	Prerequisites	Permissions	Description
vsscratchc	0x240	vsscratch	H	HRW, ASR-permission	Virtual Supervisor Scratch Capability
vsepcc	0x241	vsepc	H	HRW, ASR-permission	Virtual Supervisor Exception Program Counter Capability

Table 18. Renamed user-mode CSRs in Zcheripurecap

Zcheripurecap CSR	Address	Extended CSR	Prerequisites	Permissions	Description
jvtc	0x017	jvt	Zcmt	URW	Jump Vector Table Capability
utidc	0x480	utid	Zstid	Read: U, Write: U, ASR-permission	User thread ID

4.7. Machine-Level CSRs

Zcheripurecap extends some M-mode CSRs to hold capabilities or otherwise add new functions. pcc must grant ASR-permission to access M-mode CSRs regardless of the RISC-V privilege mode.

4.7.1. Machine Status Registers (mstatus and mstatush)

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

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

The MBE, SBE, and UBE fields determine the endianness of memory accesses other than instruction fetches performed from M-mode, S-mode, or U-mode, respectively. xBE=0 indicates little endian and xBE=1 is big endian. MBE must be read-only. SBE and UBE must be read only and equal to MBE, if S-mode or U-mode, respectively, is implemented, or read-only zero otherwise.

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

4.7.2. Machine Trap Vector Base Address Register (mtvec)

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

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

4.7.3. Machine Trap Vector Base Address Capability Register (mtvecc)

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

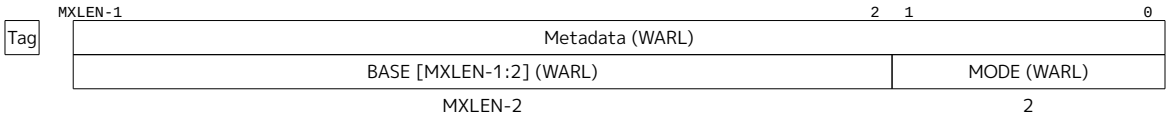


Figure 6. Machine-mode trap-vector base-capability register

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

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

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

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

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

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

4.7.4. Machine Scratch Register (mscratch)

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

Figure 7. Machine-mode scratch register

4.7.5. Machine Scratch Capability Register (mscratchc)

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

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

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

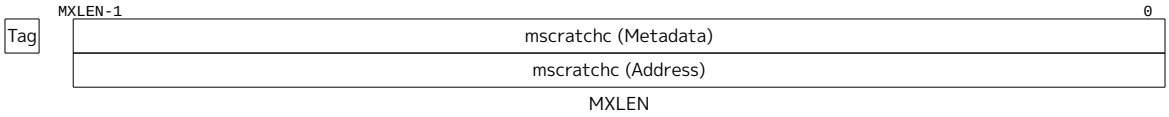


Figure 8. Machine-mode scratch capability register

4.7.6. Machine Exception Program Counter (mepc)

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

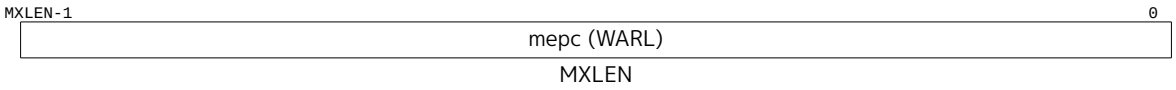


Figure 9. Machine exception program counter register

4.7.7. Machine Exception Program Counter Capability (mepcc)

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

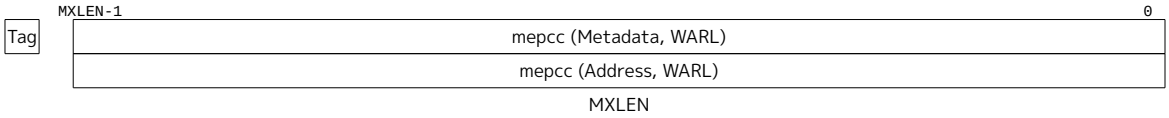


Figure 10. Machine exception program counter capability register

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

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 50, mepcc is an executable vector, so it does not need to be able to hold all possible invalid addresses. Additionally, the capability in mepcc is unsealed when it is installed in pcc on execution of an MRET instruction.

4.7.8. Machine Cause Register (mcause)

Zcheripurecap 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 19 and Table 20 respectively. The behavior and usage of mcause otherwise remains as described in (RISC-V, 2023).

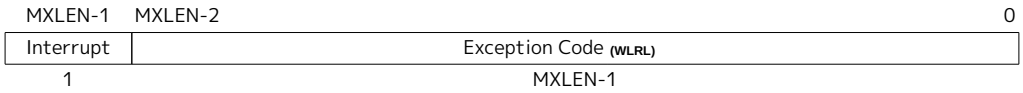


Figure 11. Machine cause register

Table 19. Machine cause register (mcause) values after trap. Entries added in Zcheripurecap are in **bold**

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*
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 20. Synchronous exception priority in decreasing priority order. Entries added in Zcheripurecap are in **bold**

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

The full details of the CHERI exceptions are in Table 24.

4.7.9. Machine Trap Delegation Register (medeleg)

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

4.7.10. Machine Trap Value Register (mtval)

The mtval register is an MXLEN-bit read-write register formatted as shown in Figure 12. When a data memory access gives rise to a CHERI fault taken into M-mode, mtval is written with the MXLEN-bit effective address which caused the fault according to the existing rules for reporting load/store addresses from (RISC-V, 2023). In this case the TYPE field of mtval2 shown in Table 21 is set to 1. For all other CHERI faults it is set to zero.

The behavior of mtval is otherwise as described in (RISC-V, 2023).

If the hardware platform specifies that no exceptions set mtval to a non-zero value, then mtval is read-only zero for all CHERI exceptions.

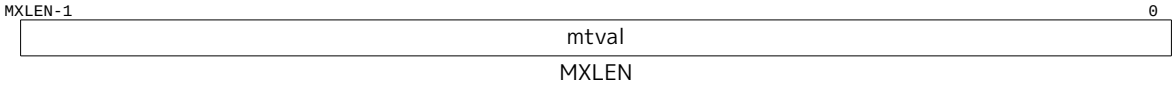


Figure 12. Machine trap value register

4.7.11. Machine Trap Value Register 2 (mtval2)

The mtval2 register is an MXLEN-bit read-write register, which is added as part of the Hypervisor extension (RISC-V, 2023). Zcheripurecap also requires the implementation of this CSR.

When a CHERI fault is taken into M-mode, mtval2 is written with additional CHERI-specific exception information with the format shown in Figure 13 to assist software in handling the trap.

If mtval is read-only zero for CHERI exceptions then mtval2 is also read-only zero for CHERI exceptions.

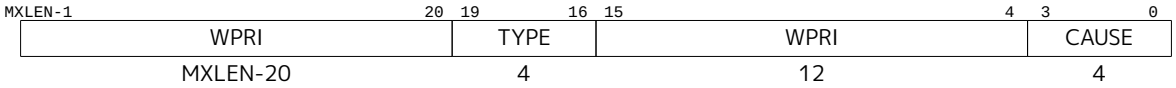


Figure 13. Machine trap value register 2 format for CHERI faults

mtval2 is also used for Hypervisor guest physical addresses, and so the implemented bits must also cover that use case. If Hypervisor is not implemented then all WPRI fields in Figure 13 are read-only-zero.

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 21 and Table 22 respectively.

Table 21. Encoding of TYPE field

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

Table 22. Encoding of CAUSE field

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

CHERI violations have the following order in priority:

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

4.8. Supervisor-Level CSRs

Zcheripurecap extends some of the existing RISC-V CSRs to be able to hold capabilities or with other new functions. pcc must grant ASR-permission to access S-mode CSRs regardless of the RISC-V privilege mode.

4.8.1. Supervisor Trap Vector Base Address Register (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.

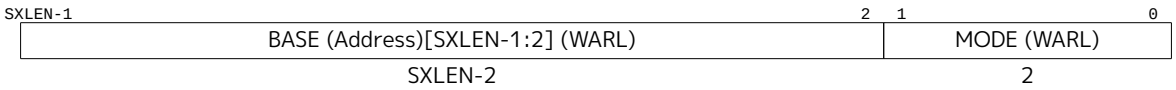


Figure 14. Supervisor trap-vector base-address register

4.8.2. Supervisor Trap Vector Base Address Capability Register (stvecc)

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

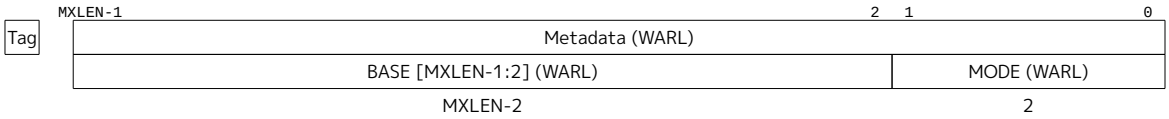


Figure 15. Supervisor trap-vector base-capability register

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

4.8.3. 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.

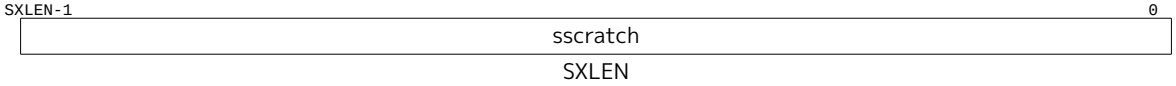


Figure 16. Supervisor-mode scratch register

4.8.4. Supervisor Scratch Capability Register (sscratchc)

The sscratchc register is a renamed extension of sscratch that is able to hold a capability.

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

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

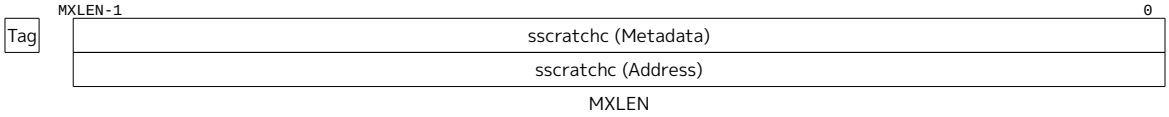


Figure 17. Supervisor scratch capability register

4.8.5. Supervisor Exception Program Counter (sepc)

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

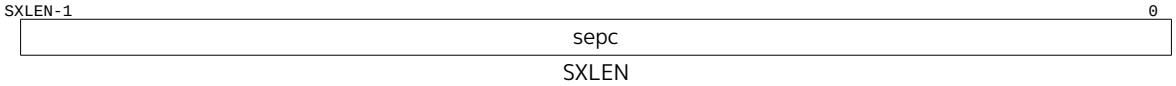


Figure 18. Supervisor exception program counter register

4.8.6. Supervisor Exception Program Counter Capability (sepcc)

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

As shown in Table 50, 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.

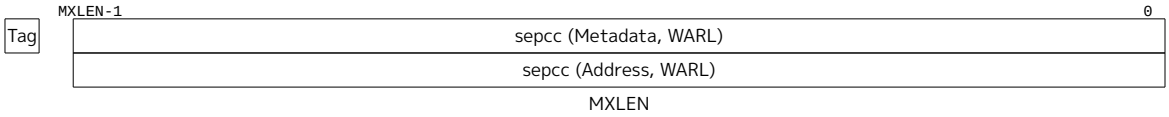


Figure 19. Supervisor exception program counter capability register

4.8.7. Supervisor Cause Register (scause)

Zcheripurecap 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 23 and Table 20 respectively. The behavior and usage of scause otherwise remains as described in (RISC-V, 2023).

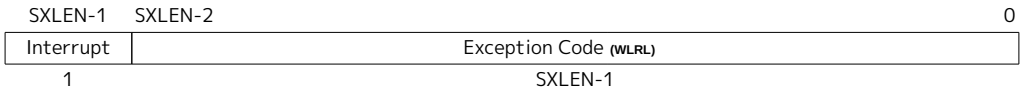


Figure 20. Supervisor cause register

Table 23. Supervisor cause register (scause) values after trap. Causes added in Zcheripurecap are in **bold**

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*

4.8.8. Supervisor Trap Value Register (stval)

The stval register is an SXLEN-bit read-write register formatted as shown in Figure 21.

stval is updated following the same rules as mtval for CHERI exceptions which are delegated to S-mode.

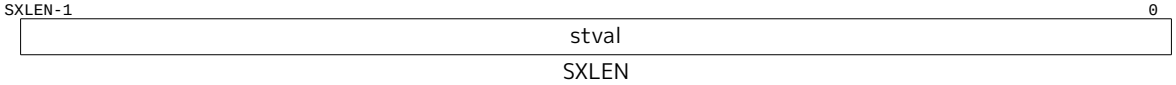


Figure 21. Supervisor trap value register

4.8.9. Supervisor Trap Value Register 2 (stval2)

The stval2 register is an SXLEN-bit read-write register, which is added as part of Zcheripurecap when the implementation supports S-mode. Its CSR address is 0x14b.

stval2 is updated following the same rules as mtval2 for CHERI exceptions which are delegated to S-mode.

The fields are identical to mtval2 for CHERI exceptions.

stval2 is not a standard RISC-V CSR, but mtval2 is.

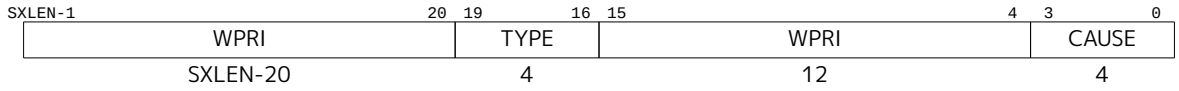


Figure 22. Supervisor trap value register 2

4.9. Unprivileged CSRs

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

4.10. CHERI Exception handling

`auth_cap` *is ddc for Integer Pointer Mode and* `cs1` *for Capability Pointer Mode*

Table 24. Valid CHERI exception combination description

Instructions	Xcause	Xtval. TYPE	Xtval. CAUSE	Description	Check
All instructions have these exception checks first					
All	28	0	0	pcc tag	not(pcc.tag)
All	28	0	1	pcc seal	isCapSealed(pcc)[1]
All	28	0	2	pcc permission	not(pcc.X-permission)
All	28	0	3	pcc invalid address	pcc holds an invalid address
All	28	0	4	pcc bounds	Any byte of current instruction out of pcc bounds
CSR/Xret additional exception check					
CSR*, MRET, SRET	28	0	2	pcc permission	not(pcc.ASR-permission) when required for CSR access or execution of MRET/SRET
direct jumps additional exception check					
JAL, Conditional branches (BEQ, BNE, BLT[U], BGE[U])	28	2	4	pcc bounds	any byte of minimum length instruction at target out of pcc bounds
indirect jumps additional exception checks					
indirect jumps	28	2	0	`cs1` tag	not(`cs1.tag`)
indirect jumps	28	2	1	`cs1` seal	isCapSealed(`cs1`) and imm12 != 0
indirect jumps	28	2	2	`cs1` permission	not(`cs1`.X-permission)

Instructions	Xcause	Xtval. TYPE	Xtval. CAUSE	Description	Check
indirect jumps	28	2	3	`cs1` invalid address	target address is an invalid address
indirect jumps	28	2	4	`cs1` bounds	any byte of minimum length 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)
all loads	28	1	3	`auth_cap` invalid address	Address is invalid (see Invalid address conversion)
all loads	28	1	4	`auth_cap` bounds	Any byte of load access out of `auth_cap` bounds
capability loads	4	N/A	N/A	load address misaligned	Misaligned capability load
Store/atomic/cache-block-operation additional exception checks					
all stores, all atomics, all cbos	28	1	0	`auth_cap` tag	not(`auth_cap.tag`)
all stores, all atomics, all cbos	28	1	1	`auth_cap` seal	isCapSealed(`auth_cap`)
all atomics, CBO.INVAL*	28	1	2	`auth_cap` permission	not(`auth_cap`.R-permission)
all stores, all atomics, CBO.INVAL*, CBO.ZERO*	28	1	2	`auth_cap` permission	not(`auth_cap`.W-permission)
CBO.CLEAN*, CBO.FLUSH*	28	1	2	`auth_cap` permission	not(`auth_cap`.R-permission) and not(`auth_cap`.W-permission)
all stores, all atomics, all cbos	28	1	3	`auth_cap` invalid address	Address is invalid (see Invalid address conversion)
all stores, all atomics	28	1	4	`auth_cap` bounds	any byte of access out of `auth_cap` bounds
CBO.ZERO*, CBO.INVAL*	28	1	4	`auth_cap` bounds	any byte of cache block out of `auth_cap` bounds
CBO.CLEAN*, CBO.FLUSH*	28	1	4	`auth_cap` bounds	all bytes of cache block out of `auth_cap` bounds
CBO.INVAL*	28	0	2	pcc permission	not(pcc.ASR-permission)

Instructions	Xcause	Xtval. TYPE	Xtval. CAUSE	Description	Check
capability stores	6	N/A	N/A	capability alignment	Misaligned capability store

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

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

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

4.11. CHERI Exceptions and speculative execution

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

4.12. Physical Memory Attributes (PMA)

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

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

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

4.13. Page-Based Virtual-Memory Systems

RISC-V's page-based virtual-memory management is generally orthogonal to CHERI. In Zcheripurecap, 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.

Zcheripurecap requires that the pcc grants the ASR-permission to change the page-table root **satp** and other virtual-memory parameters as described in Section 4.8.

4.13.1. Invalid Address Handling

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

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

RISC-V permits that CSRs holding addresses, such as mtvec and mepc (see Table 50) as well as pc, need not hold all possible invalid addresses. Implementations may convert an invalid address into some other invalid address that the register is capable of holding. Therefore, implementations often support area and power optimizations by compressing invalid addresses in a lossy fashion.

Where compressed addresses are implemented, there must be also sufficient address bits to represent all valid physical addresses. The following description is for both virtual and physical addresses.

Compressing invalid addresses allows implementations to reduce the number of flip-flops required to hold some CSRs, such as mtvec. In CHERI, invalid addresses may also be used to reduce the number of bits to compare during a bounds check, for example, to 40 bits if using Sv39, assuming that this also covers all valid physical addresses.

Care needs to be taken not to truncate physical addresses to the implemented number of physical addresses bits without also checking that the capability is still valid following the rules in this section, as the capability bounds and representable range always cover the entire MXLEN-bit address bits, but the address is likely not to.

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

Accessing CSRs

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

1. If A's address is invalid and A does not have infinite bounds (see Section 3.1), then A's tag is set to 0.
2. Write the final (potentially modified) version of capability A to the CSR e.g. mtvecc, mepcc, etc.

Branches and Jumps

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

- pcc for branches, direct jumps and any branch when in *Integer Pointer Mode* (see Chapter 7).
- The capability in the **c** input register of a jump when in *Capability Pointer Mode* (see Chapter 7).

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

1. Calculate the effective target address T of the jump or branch as required by the instruction's

behavior.

2. If T is invalid and A does not have infinite bounds (see Section 3.1), then the instruction gives rise to a CHERI fault; the *CHERI jump or branch* fault is reported in the TYPE field and invalid address violation is reported in the CAUSE field of mtval2 or stval2.
3. If T is invalid and A has infinite bounds (see Section 3.1), then A's tag is unchanged and T is written into A's address field. Attempting to execute the instruction at address T gives rise to an instruction access fault or page fault as is usual in RISC-V.
4. Otherwise T is valid and the instruction behaves as normal.

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

Memory Accesses

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

1. Calculate the effective address range R of the memory access as required by the instruction's behavior.
2. If any byte in R is invalid and A does not have infinite bounds (see Section 3.1), then the instruction gives rise to a CHERI fault; the *CHERI data* fault is reported in the TYPE field and invalid address violation is reported in the CAUSE field of mtval2 or stval2.
3. If any byte in R is invalid and A has infinite bounds (see Section 3.1), the hart will raise an access fault or page fault as is usual in RISC-V.
4. Otherwise all bytes in R are valid and the instruction behaves as normal.

4.14. Integrating Zcheripurecap with Sdext

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

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

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

These requirements allow a debugger to read and write capabilities in integer registers without disturbing other registers. These requirements may be relaxed if some other means of accessing capabilities in integer registers, such as an extension of the Access Register

abstract command, is added. The following sequences demonstrate how a debugger can read and write a capability in `c1` *if* `MXLEN` *is 64,* `hartinfo.dataaccess` *is 0,* `hartinfo.dataaddr` *is 0xBF0,* `hartinfo.datasize` *is 1,* `dmstatus.impebreak` *is 0, and* `abstractcs.progbufsize` *is 5:*

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

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

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

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

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

4.14.1. Debug Mode

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

4.14.2. Core Debug Registers

Zcheripurecap renames and extends debug CSRs that are designated to hold addresses to be able to hold capabilities. The renamed debug CSRs are listed in Table 14.

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).

4.14.3. Debug Program Counter (dpc)

The dpc register is as defined in (RISC-V, 2022). It is a DXLEN-bit register used as the PC saved when

entering debug mode. dpc is extended into dpcc.

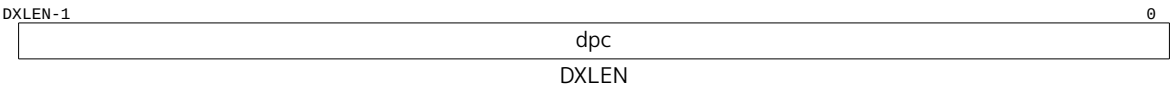


Figure 23. Debug program counter

4.14.4. Debug Program Counter Capability (dpcc)

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

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

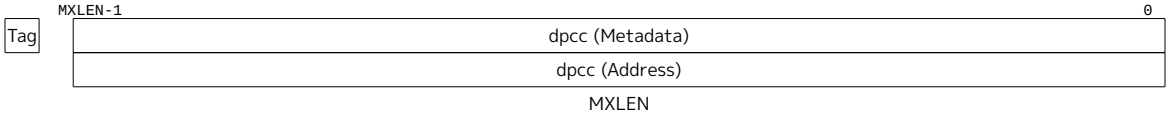


Figure 24. Debug program counter capability

Upon entry to debug mode, (RISC-V, 2022), does not specify how to update the PC, and says PC relative instructions may be illegal. This concept is extended to include any instruction which reads or updates pcc, which refers to all jumps, conditional branches and AUIPC. The exceptions are MODESW.CAP and MODESW.INT which *are* supported if Zcherihybrid is implemented, see dinfc for details.

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

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

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

The legalisation of dpcc follows the same rules described for mepcc.

4.14.5. Debug Scratch Register 0 (dscratch0)

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

Figure 25. Debug scratch 0 register

4.14.6. Debug Scratch Register 0 Capability (dscratch0c)

The dscratch0c register is a CLEN-bit plus tag bit renamed extension to dscratch0 that is able to hold a capability.

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

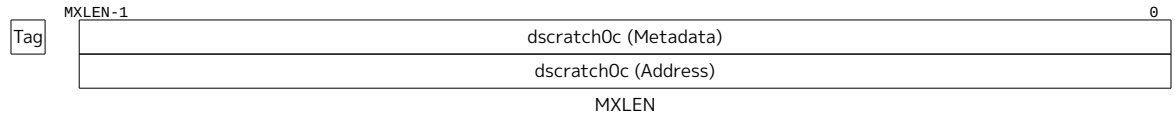


Figure 26. Debug scratch 0 capability register

4.14.7. Debug Scratch Register 1 (dscratch1)

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

Figure 27. Debug scratch 1 register

4.14.8. Debug Scratch Register 1 Capability (dscratch1c)

The dscratch1c register is a CLEN-bit plus tag bit renamed extension to dscratch1 that is able to hold a capability.

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

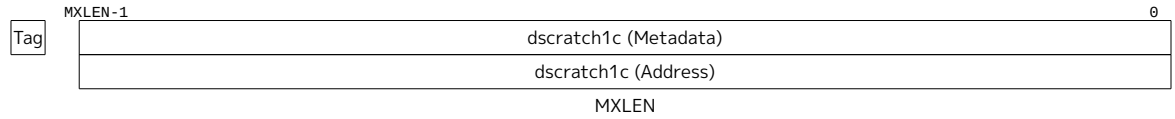


Figure 28. Debug scratch 1 capability register

4.14.9. Debug Infinite Capability Register (dinfc)

The dinfc register is a CLEN-bit plus tag bit CSR only accessible in debug mode.

The reset value is the Infinite capability.

If Zcherihybrid is implemented:

- The M-bit is reset to *Integer Pointer Mode* (1).
- The debugger can set the M-bit to *Capability Pointer Mode* (0) by executing MODESW.CAP from the program buffer.
  - Executing MODESW.CAP causes subsequent instructions execution from the program buffer, starting from the next instruction, to be executed in *Capability Pointer Mode*. It also sets the CHERI execution mode to *Capability Pointer Mode* on future entry into debug mode.
  - Therefore to enable use of a CHERI debugger, a single MODESW.CAP only needs to be executed once from the program buffer after resetting the core.
  - The debugger can also execute MODESW.INT to change the mode back to *Integer Pointer Mode*, which also affects the execution of the next instruction in the program buffer, updates the M-bit of dinfc and controls which CHERI execution mode to enter on the next entry into debug mode.

The M-bit of dinfc is *only* updated by executing MODESW.CAP or MODESW.INT from the program buffer.

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

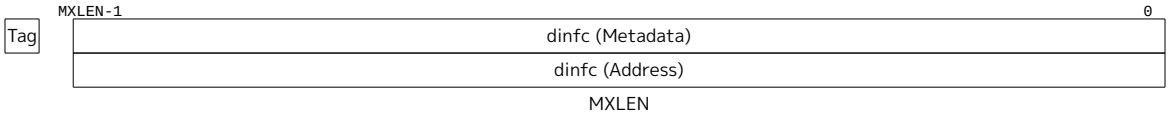


Figure 29. Debug infinite capability register

4.15. Integrating Zcheripurecap with Sdtrig

The Sdtrig extension is generally orthogonal to Zcheripurecap. However, the priority of synchronous exceptions and where triggers fit is adjusted as shown in Table 25.

Table 25. Synchronous exception priority (including triggers) in decreasing priority order. Entries added in Zcheripurecap are in **bold**

Priority	Exc. Code	Description	Trigger
Highest	3 3 3 3		etrigger icount itrigger mcontrol/mcontrol6 after (on previous instruction)
	3	Instruction address breakpoint	mcontrol/mcontrol6 execute address before
	28	**Prior to instruction address translation: CHERI fault due to PCC checks (tag, execute permission, invalid address and bounds)**	
	12, 1	During instruction address translation: First encountered page fault or access fault	
	1	With physical address for instruction: Instruction access fault	
	3		mcontrol/mcontrol6 execute data before
	2 0 8,9,11 3	Illegal instruction Instruction address misaligned Environment call Environment break	
	3	Load/store/AMO address breakpoint	mcontrol/mcontrol6 load/store address before
	3		mcontrol/mcontrol6 store data before
	28	**CHERI faults due to: PCC ASR-permission clear Branch/jump target address checks (tag, execute permissions, invalid address and bounds)**	

Priority	Exc. Code	Description	Trigger
	28	**Prior to address translation for an explicit memory access: Load/store/AMO capability address misaligned CHERI fault due to capability checks (tag, permissions, invalid address and bounds)**	
	4,6	Optionally: Load/store/AMO address misaligned	
	13, 15, 5, 7	During address translation for an explicit memory access: First encountered page fault or access fault	
	5,7	With physical address for an explicit memory access: Load/store/AMO access fault	
	4,6	If not higher priority: Load/store/AMO address misaligned	
Lowest	3		mcontrol/mcontrol6 load data before

Chapter 5. "Zcheripte" 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 facilitate capability revocation and control the flow of capabilities in memory at the page granularity. For this reason, the Zcheripte extension adds new bits to RISC-V's Page Table Entry (PTE) format.

Implementing any virtual memory translation scheme (*Sv39*, *Sv48* or *Sv57*) and Zcheripurecap requires Zcheripte to be implemented.

There is no explicit mechanism for enabling or disabling Zcheripte. A VM-enabled legacy (non-CHERI) OS running in Integer Pointer Mode will not load or store capabilities, and so the default state of CW=0 causing loaded capabilities to have their tags cleared, and stored capabilities with their tags set to cause a page fault, won't occur. A CHERI-aware OS running a VM-enabled OS is required to support Zcheripte, and the minimum level of support is to set CW to 1 in all PTEs and leave sstatus.CRG and CRG in all PTEs set to 0, which will allow capabilities with their tags set to be loaded and stored successfully.

5.1. Limiting Capability Propagation

Page table enforcement can allow the operating system to limit the flow of capabilities between processes. It is highly desirable that a process should only possess capabilities that have been issued for that address space by the operating system. Unix processes may share memory for efficient communication, but capability pointers must not be shared across these channels into a foreign address space. An operating system might defend against this by only issuing a capability to the shared region that does not grant the load/store capability permission. However, there are circumstances where portions of general-purpose, mmapped* memory become shared, and the operating system must prevent future capability communication through those pages. This is not possible without restructuring software, as the capability for the original allocation, which spans both shared memory and private memory, would need to be deleted and replaced with a list of distinct capabilities with appropriate permissions for each range. Such a change would not be transparent to the program. Such sharing through virtual memory is on the page granularity, so preventing capability writes with a PTE permission is a natural solution.

* *allocated using mmap*

5.2. Capability Revocation

Page table enforcement can accelerate concurrent capability revocation for temporal safety. Without page table capability protection, a concurrent capability revocation sweep must begin by visiting all PTEs to mark them unreadable, henceforth trapping on any read to a new page to sweep it clean before proceeding. With a page-granularity generational capability read permission, we can eliminate the initial permission change of all PTEs. In addition, a page-granularity capability write control can eliminate many pages from the sweep that are known to not contain capabilities.

5.3. Extending the Page Table Entry Format

The page table entry format remains unchanged for Sv32. However, two new bits, Capability Write (CW) and Capability Read Generation (CRG), are added to leaf PTEs in Sv39, Sv48 and Sv57 as shown in Figure 30, Figure 31 and Figure 32 respectively.

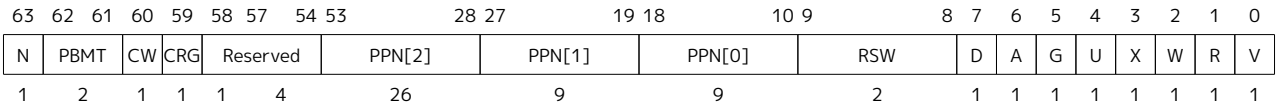


Figure 30. Sv39 page table entry

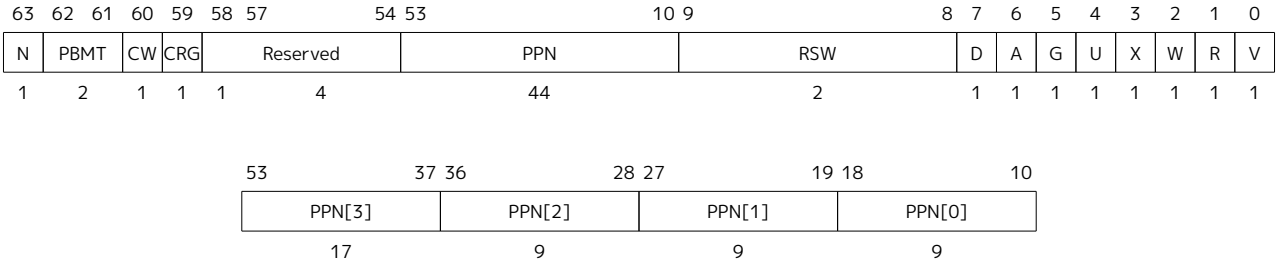


Figure 31. Sv48 page table entry

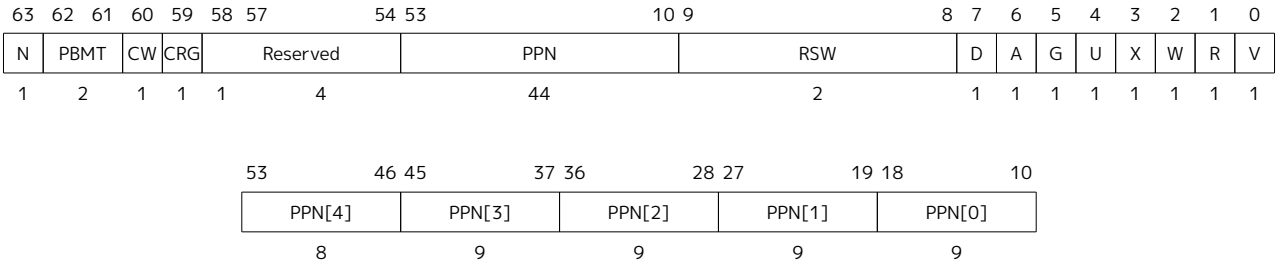


Figure 32. Sv57 page table entry

All behaviour related to Zcheripte requires the authorizing capability to have C-permission. If C-permission is not granted then all capability load/stores and AMOs always clear the tag and the behaviour in this section is not relevant.

The CW bit indicates whether reading or writing capabilities with the tag set to the virtual page is permitted. When the CW bit is set, capabilities are written as usual, and capability reads are controlled by the CRG bit.

If the CW bit is clear then:

- When a capability load or AMO instruction is executed, the implementation clears the tag bit of the capability read from the virtual page.
- When CRG is clear, the "no capability state", a store page fault exception is raised when a capability store or AMO instruction is executed and the tag bit of the capability being written is set.
- When CRG is set, the "pre-CW state", two schemes are permitted:
  - The same behavior as when CRG is clear, allowing software interpretation of this state.
  - When a capability store or AMO instruction is executed and the tag bit of the capability being written is set, the implementation sets the CW bit and assigns the CRG bit equal to sstatus.CRG. 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

CW bit and CRG 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.

When CW is set, the CRG bit indicates the current generation of the virtual memory page with regards to the ongoing capability revocation cycle. Two schemes are permitted:

- A load page fault exception is raised when a capability load or AMO instruction is executed and the virtual page's CRG bit does not equal sstatus.CRG.
- A load page fault exception is raised when a capability load or AMO instruction is executed, the capability read from memory has tag set and the virtual page's CRG bit does not equal sstatus.CRG.

An implementation may avoid a dependency on the tag bit value of the capability read. This will introduce additional traps during revocation sweeps.

Table 26. Summary of Load CW and CRG behavior in the PTEs

PTE.CW	PTE.CRG	Load/AMO
0	X	Clear loaded tag
1	≠ sstatus.CRG	Page fault, or page fault if tag is set[1]
1	= sstatus.CRG	Normal operation

Table 27. Summary of Store CW and CRG behavior in the PTEs

PTE.CW	PTE.CRG	Store/AMO
0	0	Page fault if stored tag is set
0	1	Page fault if stored tag is set, or hardware CW and CRG update[2]
1	X	Normal operation

[1] The choice here is whether to take data dependent exceptions on loads or atomic operations. It is legal for the implementation to fault even if the tag is not set since this behaviour is only an optimization for software. This means it is also legal to only check the tag under certain conditions and conservatively fault otherwise.

[2] The choice here follows the pattern of whether to implement the *Svade* extension to take page-fault exceptions for A and D PTE bit state changes, or whether to implement a hardware updating mechanism.

5.4. Extending the Supervisor (sstatus) and Virtual Supervisor (vsstatus) Status Registers

The sstatus and vsstatus CSRs are extended to include the new Capability Read Generation (CRG) bit

as shown.

When V=1 vsstatus.CRG is in effect.

mstatus.CRG also exists. Reading or writing it is equivalent to reading or writing sstatus.CRG.

As there is no M-mode translation available in RISC-V, there is no current software use for mstatus.CRG. It is only included not to break the rule that sstatus is required to be a subset of mstatus.

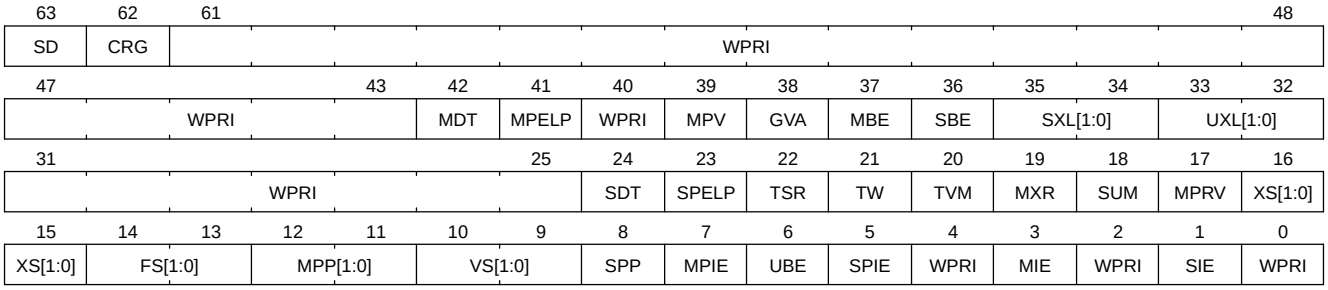


*Figure 33. Virtual Supervisor-mode status (**mstatus**) register when MXLEN=64*

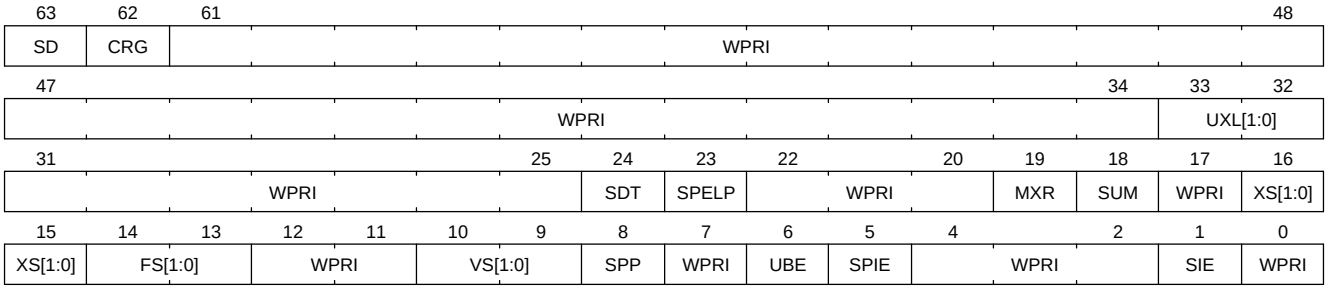


*Figure 34. Supervisor-mode status (**sstatus**) register when SXLEN=64*

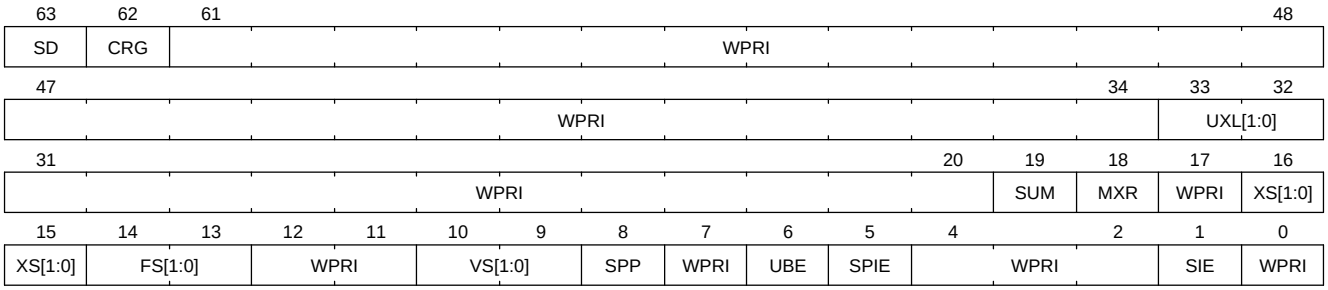


*Figure 35. Virtual Supervisor-mode status (**vsstatus**) register when VSXLEN=64*

Chapter 6. "Zcherilevels" Extension for Capability Levels

Zcherilevels is an extension to Zcheripurecap that adds support for associating a level with capabilities and limiting the flow of capabilities to specific memory region subsets. This extension allows assigning a level to capabilities, which in conjunction with two new permissions allows enforcing invariants on capability propagation. For example, this can be used to ensure that a callee can only write a copy of the passed-in argument capability to specific locations in memory (e.g. the callee's stack frame but not the heap). It can also be used to avoid sharing of compartment-local data (such as pointers to a stack object) between compartments.

This specification only defines the architectural mechanics of this feature, for further information on how this can be used by software please refer to (Watson et al., 2023).

The number of supported capability levels is implementation-defined, but this specification currently only requires supporting two levels (*local* and *global*).

6.1. Capability format changes

Zcherilevels adds a new `LVLBITS`-bit field to the capability encoding, the *Capability Level (CL)*. It also adds two new permission fields, EL-permission and SL-permission.

- For `MXLEN=64` capability encoding, the AP field is widened by `LVLBITS+1` bits (i.e. 2 bits for `LVLBITS=1`)

  The `MXLEN=64` capability encoding diagram shows the layout for `LVLBITS=1`

- For `MXLEN=32` the capability's AP field is able to encode these permissions without increasing in size (provided that LVLBITS≤2).

Zcherilevels requires that LVLBITS≥1 although LVLBITS>1 is considered an experimental enhancement of this extension. See Section 6.4 for the mechanics when LVLBITS>1.

6.1.1. Capability Level (CL)

The Capability Level (CL) is a new field added to the capability encoding, as shown in Section 3.1.

With `LVLBITS=1`, the *Capability Level* can hold two values: when set to 1 the capability is *global* (in general allowing it to be stored using any authorizing capability), and when set to 0 the capability is *local*, and can only be stored by authorizing capabilities with the SL-permission set. Furthermore, the EL-permission can be used to restrict loading of *global* capabilities — causing the hardware to automatically set the level of loaded capabilities to *local* instead.

The current specification only defines two levels, equivalent to local and global capabilities from CHERI v9, Morello and CHERIoT.

As with permissions, the capability level can only be decreased but never increased (without deriving from a capability with a higher level). Therefore, the capability level is adjusted using the ACPERM instruction (see Section 6.2) and are queried using GCPERM.

6.1.2. New capability permissions

Zcherilevels introduces two new capability permissions:

Store Level Permission (SL)

This is a `LVLBITS` wide field that allows limiting the propagation of capabilities using the following logic: capabilities with a Capability Level (CL) less than the inverse of the authorizing capability's SL-permission will be stored with the tag cleared. With `LVLBITS=1` there is a single bit comparison, so it behaves as follows:

- If this field (as well as C-permission and W-permission) is set to 1 then capabilities may be stored via this capability regardless of their associated Capability Level (CL).
- If this field is zero, then any capability with a Capability Level (CL) of zero (i.e. *local*), will be stored with the tag cleared.

For `LVLBITS=1` this permission is equivalent to StoreLocal in CHERI v9, Morello and CHERIoT.

Elevate Level Permission (EL)

Any capability with its tag set to 1 that is loaded from memory has its EL-permission cleared and its Capability Level (CL) restricted to the authorizing capability's Capability Level (CL) if the authorizing capability does not grant EL-permission. This permission is similar to the existing LM-permission, but instead of applying to the W-permission on the loaded capability it restricts the CL field.

Table 28. Encoding of architectural permissions for MXLEN=32 when Zcherilevels is implemented

Bits[4:3]	R	W	C	LM	EL	SL	X	ASR	Mode[1]	Notes
Quadrant 0: Non-capability data read/write										
bit[2] - write, bit[1] - reserved (0), bit[0] - read										
Reserved bits for future extensions are 0 so new permissions are not implicitly granted										
0									N/A	No permissions
1	✔								N/A	Data RO
2-3	reserved									
4		✔							N/A	Data WO
5	✔	✔							N/A	Data RW
6-7	reserved									
Quadrant 1: Executable capabilities										
bit[0] - M-bit (0-*Capability Pointer Mode*, 1-*Integer Pointer Mode*)										
Bits[4:3]	R	W	C	LM	EL	SL	X	ASR	Mode[1]	
0-1	✔	✔	✔	✔	✔	∞	✔	✔	Mode[1]	Execute + ASR (see Infinite)
2-3	✔		✔	✔	✔	∞[1]	✔		Mode[1]	Execute + Data & Cap RO
4-5	✔	✔	✔	✔	✔	∞	✔		Mode[1]	Execute + Data & Cap RW

Bits[4:3]	R	W	C	LM	EL	SL	X	ASR	Mode[1]	Notes
6-7	✔	✔				0[1]	✔		Mode[1]	Execute + Data RW
Quadrant 2: Restricted capability data read/write										
bit[2] = write, bit[1:0] = store level. R and C implicitly granted, LM dependent on W permission.										
Bits[4:3]	R	W	C	LM	EL	SL	X	ASR	Mode[1]	
0-2	reserved									
3	✔		✔			0[1]			N/A	Data & Cap RO (without LM-permission)
4	✔	✔	✔	✔		*(3)*			N/A	*Reserved* when `LVLBITS=1` [2]
5	✔	✔	✔	✔		*(2)*			N/A	*Reserved* when `LVLBITS=1` [2]
6	✔	✔	✔	✔		1			N/A	Data & Cap RW (with store *local*, no EL-permission)
7	✔	✔	✔	✔		0			N/A	Data & Cap RW (no store *local*, no EL-permission)
Quadrant 3: Capability data read/write										
bit[2] = write, bit[1:0] = store level. R and C implicitly granted.										
Reserved bits for future extensions must be 1 so they are implicitly granted										
Bits[4:3]	R	W	C	LM	EL	SL	X	ASR	Mode[1]	
0-2	reserved									
3	✔		✔	✔	✔	0[1]			N/A	Data & Cap RO
4	✔	✔	✔	✔	✔	*(3)*			N/A	*Reserved* when `LVLBITS=1` [2]
5	✔	✔	✔	✔	✔	*(2)*			N/A	*Reserved* when `LVLBITS=1` [2]
6	✔	✔	✔	✔	✔	1			N/A	Data & Cap RW (with store *local*)
7	✔	✔	✔	✔	✔	0			N/A	Data & Cap RW (no store *local*)

[1] SL isn't applicable in these cases, but this value is reported by GCPERM to simplify the rules followed by ACPERM

[2] These entries are reserved when `LVLBITS=1` and in use when `LVLBITS=2`

6.2. Changing capability levels and permissions

While capability levels (CL) are conceptually a label on the capability rather than a permission granted by the capability, they are adjusted using the ACPERM instruction. This avoids the need for a dedicated instruction and allows reducing the level and removing EL-permission in a single

instruction.

6.3. Capability level summary table

A capability with CL=1 is global and with CL=0 is local.

Table 29. Zcherilevels `LVLBITS=1` *summary table for stored capabilities*

Auth cap field			Data cap field	
W	C	SL	CL	Notes
1	1	1	X	Store data capability unmodified
		0	1	Store data capability unmodified (CL ≥ ~SL)
			0	Store data capability with tag cleared (CL < ~SL)

if W=0 or C=0 then SL is irrelevant

Table 30. Zcherilevels additional rules for loading capabilities

Auth cap field				Data cap field		
R	C	EL	CL	Tag	Sealed	Action
1	1	0	X	1	Yes	Load data capability with `CL=min(auth.CL, data.CL)`, EL unchanged
					No	Load data capability with `EL=0, CL=min(auth.CL, data.CL)`
All other cases						Load data capability with EL, CL unmodified

6.4. Extending Zcherilevels to more than two levels

When `LVLBITS>1`, the behaviour of ACPERM can no longer use masking to adjust the Capability Level (CL) or SL-permission, but instead must perform an integer minimum operation on those `LVLBITS`-wide fields. The CL field of the resulting capability is set to `min(rs2[CL], cs1[CL])` (equivalent to `rs2[CL] & cs1[CL]` for `LVLBITS=1`). Similarly, SL-permission is set to `min(rs2[SL], cs1[SL])` (equivalent to `rs2[SL] & cs1[SL]` for `LVLBITS=1`).

When storing capabilities, the SL-permission checks need to perform a `LVLBITS`-wide integer comparison instead of just testing a single bit. Considering for an example `LVLBITS=2`:

SL-permission	Permitted for levels	Resulting semantics
3	As low as `~0b11=0`	Authorizes stores of capabilities with any level
2	As low as `~0b10=1`	Strip tag for level 0 (most *local*), keep for 1,2,3
1	As low as `~0b01=2`	Strip tag for level 0&1, keep for 2&3
0	As low as `~0b00=3`	Strip tag for level 0,1,2, i.e. only the most *global* can be stored

While this extra negation is non-intuitive, it is required such that ACPERM can use a monotonically decreasing operation for both CL SL-permission.

The layout of the ACPERM input / GCPERM result is not yet defined, but existing bits will not be moved around so the SL/CL fields will be non-contiguous.

Chapter 7. "Zcherihybrid" Extension for CHERI *Integer Pointer Mode*

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

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

7.1. CHERI Execution Mode

Zcherihybrid 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 Pointer Mode* and *Integer Pointer Mode*. Additionally, there is a new unprivileged register: the default data capability, ddc, that is used to authorise all data memory accesses when in *Integer Pointer Mode*.

The current CHERI execution mode is given by the M-bit field of pcc that is encoded as described in Section 7.1.

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

- The authorising capability used to execute memory access instructions. In *Integer Pointer Mode*, ddc is implicitly used. In *Capability Pointer 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 *Integer Pointer Mode* but not *Capability Pointer Mode* and vice-versa (see Chapter 12).

The implication is that the CHERI execution mode is always Capability Pointer Mode on implementations that support Zcheripurecap, but not Zcherihybrid.

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 Pointer Mode*). The mode is shown in the assembly syntax.

The CHERI execution mode is key in providing backwards compatibility with the base RISC-V ISA. RISC-V software is able to execute unchanged in implementations supporting both Zcheripurecap and Zcherihybrid provided that the Infinite capability is installed in ddc and pcc (with M=1, i.e. in *Integer Pointer Mode*). Setting both registers to Infinite ensures that:

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

7.2. CHERI Execution Mode Encoding

Zcherihybrid adds a new CHERI execution Mode field (M) to the capability format, which is only valid

for code capabilities, i.e. when the X-permission is set.

- When MXLEN=32, the Mode is encoded in bit 0 of quadrant 1 from the AP field *even though it is not a permission* as shown in Table 4.
  - Only quadrant 1 represents executable capabilities, and so it's the only one which encodes the Mode.
- When MXLEN=64, the Mode is encoded separately; a new M-bit field is added to the capability format as shown in Table 5. The M-bit is only valid for code capabilities, otherwise the field is reserved.

Mode is encoded with permissions for MXLEN=32, but is not a permission. It is orthogonal to permissions as it can vary arbitrarily using SCMODE.

In both encodings:

- Mode (M)=0 indicates *Capability Pointer Mode*.
- Mode (M)=1 indicates *Integer Pointer Mode*.

The current CHERI execution mode is given by the M-bit of the pcc and the CHERI register access settings as follows:

- The Mode is *Capability Pointer Mode* when the M-bit of the pcc is 0, **and** CHERI register access is enabled for the current privilege.
- Otherwise the Mode is *Integer Pointer Mode*.

When the M-bit can be set, the rules defined by ACPERM must be followed.

7.2.1. Observing the CHERI Execution Mode

The effective CHERI execution mode is given by the values of some CSRs and the M-bit from the PCC. The following code sequences demonstrate how a program can observe the current, effective CHERI execution mode depending on the machine's privilege mode.

In debug mode, the following sequence executed from the program buffer will write 0 for *Capability Pointer Mode* and 1 for *Integer Pointer Mode* to `x1`:

```
csrr   c1, dinfc
gcmode x1, c1
```

In any other privilege mode, the following sequence will write 0 for *Capability Pointer Mode* and 1 for *Integer Pointer Mode* to `x1`:

```
auipc c1, 0
gctag x1, c1
```

7.3. Zcherihybrid Instructions

Zcherihybrid introduces a small number of new mode-switching and capability manipulation instructions to the base RISC-V integer ISA, as shown in Table 46. Additionally, the behavior of some existing instructions changes depending on the current CHERI execution mode.

7.3.1. Capability Load and Store Instructions

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

The load capability instruction is LC. When the CHERI execution mode is *Capability Pointer Mode*; the instruction behaves as described in Section 4.3. In *Integer Pointer Mode*, the capability authorising the memory access is ddc, so the effective address is obtained by adding the **x** register to the sign-extended offset.

The store capability instruction is SC. When the CHERI execution mode is *Capability Pointer Mode*; the instruction behaves as described in Section 4.3. In *Integer Pointer Mode*, the capability authorising the memory access is ddc, so the effective address is obtained by adding the **x** register to the sign-extended offset.

7.3.2. Capability Manipulation Instructions

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

A new GCMODE instruction allows decoding the CHERI execution mode from an arbitrary capability held in an **x** register. The output is written to an unprivileged **x** register.

7.3.3. Mode Change Instructions

New CHERI execution mode switch instructions, MODESW.CAP and MODESW.INT, allow software to change the hart's current M-bit in pcc. If the current mode in the pcc is *Integer Pointer Mode*, then the mode after executing MODESW.CAP is *Capability Pointer Mode* and similarly for MODESW.INT when in *Capability Pointer Mode*. This instruction effectively writes the CHERI execution mode M-bit of the capability currently installed in the pcc.

7.4. Existing RISC-V Instructions

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

7.4.1. Control Transfer Instructions

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

The jump and link instruction JAL when the CHERI execution mode is Capability; behaves as described in Section 4.4. When the mode is *Integer Pointer Mode*. In this case, the address of the instruction following the jump (**pc** + 4) is written to an **x** register; that register's tag and capability metadata are zeroed.

The jump and link register instruction is JALR when the CHERI execution mode is *Capability Pointer Mode*; behaves as described in Section 4.4. When the mode is *Integer Pointer Mode*. 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.

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

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

7.4.2. Conditional Branches

The behaviour is as shown in Section 4.4.2.2.

7.4.3. Load and Store Instructions

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

Loads and stores behave as described in Section 4.4 when in *Capability Pointer Mode*. In *Integer Pointer Mode*, the instructions behave as described in the RISC-V base ISA 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 4.4 regardless of the CHERI execution mode.

7.4.4. CSR Instructions

Zcherihybrid adds the concept of CSRs which contain a capability where the address field is visible in *Integer Pointer Mode* (e.g. mtvec) and the full capability is visible in *Capability Pointer Mode* through a different name (e.g. mtvecc). These are referred to as *extended CSRs*. Also, Zcherihybrid adds the new capability CSRs listed in Table 31.

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

When CSRRW is executed on an extended CSR in *Integer Pointer Mode*:

- The register operand is an **x** register.
- Only XLEN bits from the **x** source are written to the capability address field.
  - The tag and metadata are updated as specified in Table 48.
- Only XLEN bits are read from the capability address field, which are extended to MXLEN bits according to (RISC-V, 2023) *(3.1.6.2. Base ISA Control in mstatus Register)* and are then written to the destination **x** register.

When CSRRW is executed on an extended CSR in *Capability Pointer Mode*, or on a new capability CSR regardless of the CHERI execution mode:

- The register operand is a **c** register.
- The full capability in the **c** register source is written to the CSR.
  - The capability may require modification before the final written value is determined (see Table 48).

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

When an extended CSR or a new capability CSR is used with 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 48 for an XLEN write.
- When accessing an extended CSR:
  - In *Integer Pointer Mode*, XLEN bits are read from the capability address field and written to an output **x** register.
  - In *Capability Pointer Mode*, CLEN bits are read from the CSR and written to an output **c** register.
- When accessing a new capability CSR:
  - CLEN bits are read from the CSR and written to an output **c** register.

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

7.5. Integrating Zcherihybrid with Sdext

A new debug default data capability (dddc) CSR is added at the CSR number shown in Table 31.

Zcherihybrid allows MODESW.CAP and MODESW.INT to execute in debug mode.

When entering debug mode, whether the core enters *Integer Pointer Mode* or *Capability Pointer Mode* is controlled by the M-bit in dinfc.

The current mode can be read from dinfc.

7.6. Debug Default Data Capability (dddc)

dddc is a register that is able to hold a capability. The address is shown in Table 31.

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

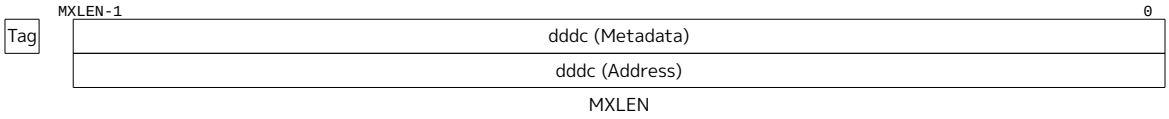


Figure 36. Debug default data capability

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

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

As shown in Table 50, dddc is a data pointer, so it does not need to be able to hold all possible invalid addresses.

7.7. Disabling CHERI Registers

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

- The CHERI instructions in Section 4.3 and Appendix D cause illegal instruction exceptions
- Executing CSR instructions accessing any CSR added by Zcherihybrid (see Table 31) causes an illegal instruction exception
- Executing CSR instructions accessing any extended CSR (see Section 4.6) only allows XLEN access.
- All allowed instructions execute as if the CHERI execution mode is *Integer Pointer Mode*. The mode bit in pcc is treated as if it was zero while CHERI register access is disabled.

CHERI register access is disabled if

- XLEN in the current mode is less than MXLEN, or
- the endianness in the current mode is not the reset value of mstatus.MBE, or
- the effective CRE for the current privilege is 0.

The effective CRE for the current privilege is:

- Machine: `mseccfg.CRE`
- Supervisor: `mseccfg.CRE & menvcfg.CRE`
- User: `mseccfg.CRE & menvcfg.CRE & senvcfg.CRE`

The effective CRE is always 1 in debug mode.

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

Disabling CHERI register access prevents low-privileged Integer Pointer Mode software from interfering with the correct operation of higher-privileged Integer Pointer Mode software that do not perform ddc switches on trap entry and return.

Disable CHERI register access also allows harts supporting CHERI to be fully compatible with standard RISC-V, so CHERI instructions, such as CRAM, that do not change the state of CHERI CSRs raise exceptions when CRE=0.

Table 59 summarizes the behavior of a hart in connection with the CRE and the CHERI execution mode.

7.8. Added CLEN-wide CSRs

Zcherihybrid adds the CLEN-wide CSRs shown in Table 31.

Table 31. CLEN-wide CSRs added in Zcherihybrid

CLEN CSR	Address	Prerequisites	Permissions	Description
dddc	0x7bc	Zcherihybrid, Sdext	DRW	Debug Default Data Capability (saved/restored on debug mode entry/exit)
mtdc	0x74c	Zcherihybrid, M-mode	MRW, ASR-permission	Machine Trap Data Capability (scratch register)
stdc	0x163	Zcherihybrid, S-mode	SRW, ASR-permission	Supervisor Trap Data Capability (scratch register)
vstdc	0x245	Zcherihybrid, H	HRW, ASR-permission	Virtual Supervisor Trap Data Capability (scratch register)
ddc	0x416	Zcherihybrid	URW	User Default Data Capability

7.8.1. Machine Status Registers (mstatus and mstatush)

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

Setting the SXL or UXL field to a value that is not MXLEN disables most CHERI features and instructions, as described in Section 7.7, while in that privilege mode.

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

Whenever XLEN in any mode is set to a value less than MXLEN, standard RISC-V rules from (RISC-V, 2023) are followed. This means that all operations must ignore source operand register bits above the configured XLEN, and must sign-extend results to fill all MXLEN bits in the destination register. Similarly, **pc** bits above XLEN are ignored, and when the **pc** is written, it is sign-extended to fill MXLEN. The integer writing rule from CHERI is followed, so that every register write also zeroes the metadata and tag of the destination register.

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

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

Setting the MBE, SBE, or UBE field to a value that is not the reset value of MBE disables most CHERI features and instructions, as described in Section 7.7, while in that privilege mode.

7.8.2. Machine Trap Default Capability Register (mtdc)

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

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

Access to this CSR is illegal if CHERI register access is disabled for the current privilege.

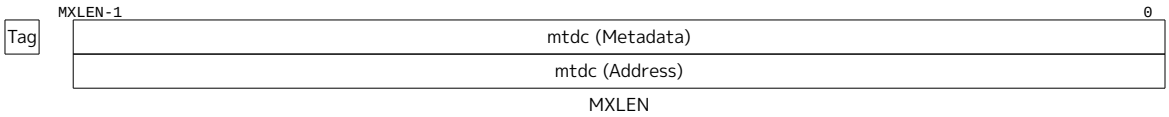


Figure 37. Machine-mode trap data capability register

7.8.3. Machine Security Configuration Register (mseccfg)

Zcherihybrid adds a new enable bit to mseccfg as shown in Figure 38.

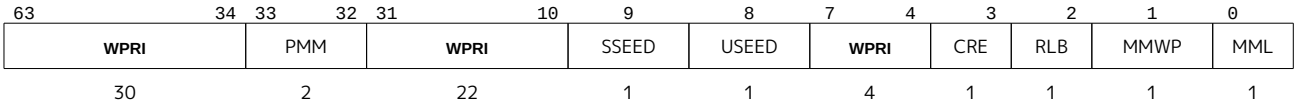


*Figure 38. Machine security configuration register (**mseccfg**)*

The CHERI Register Enable (CRE) bit controls whether M-mode and lower privilege levels have access to capability registers and instructions. When mseccfg.CRE=1, all CHERI instructions and registers can be accessed. When mseccfg.CRE=0, CHERI register and instruction access is prohibited for M-mode and lower privilege levels as described in Section 7.7.

The reset value is 0.

7.8.4. Machine Environment Configuration Register (menvcfg)

Zcherihybrid adds a new enable bit to menvcfg as shown in Figure 39.

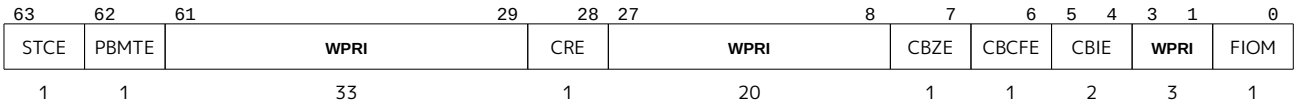


*Figure 39. Machine environment configuration register (**menvcfg**)*

The CHERI Register Enable (CRE) bit controls whether less privileged levels can perform explicit accesses to CHERI registers. When menvcfg.CRE=1 and mseccfg.CRE=1, CHERI registers can be read and written by less privileged levels. When menvcfg.CRE=0, CHERI registers are disabled in less privileged levels as described in Section 7.7.

The reset value is 0.

7.8.5. Supervisor Trap Default Capability Register (stdc)

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

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

Access to this CSR is illegal if CHERI register access is disabled for the current privilege.

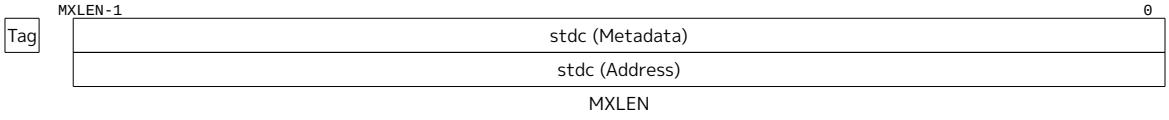


*Figure 40. Supervisor trap data capability register (**stdc**)*

7.8.6. Supervisor Environment Configuration Register (senvcfg)

The **senvcfg** register operates as described in the RISC-V Privileged Specification. Zcherihybrid adds a new enable bit as shown in Figure 41.

*Figure 41. Supervisor environment configuration register (**senvcfg**)*

The CHERI Register Enable (CRE) bit controls whether U-mode can perform explicit accesses to CHERI registers. When senvcfg.CRE=1 and menvcfg.CRE=1 and mseccfg.CRE=1 CHERI registers can be read and written by U-mode. When senvcfg.CRE=0, CHERI registers are disabled in U-mode as described in Section 7.7.

The reset value is 0.

7.8.7. Default Data Capability (ddc)

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

Access to this CSR is illegal if CHERI register access is disabled for the current privilege.

CRE is not required for the implicit access required by checking memory accesses against ddc

As shown in Table 50, ddc is a data pointer, so it does not need to be able to hold all possible invalid addresses.

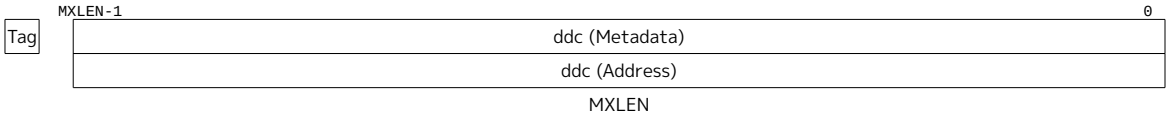


Figure 42. Unprivileged default data capability register

Chapter 8. Integrating Zcheripurecap and Zcherihybrid with the Hypervisor Extension

The RISC-V hypervisor (H) extension virtualizes the supervisor-level architecture to support the efficient hosting of guest operating systems atop a type-1 or type-2 hypervisor (RISC-V, 2023). The hypervisor extension is generally orthogonal to CHERI; the main requirements, when integrating with Zcheripurecap and Zcherihybrid, is that address CSRs added for hypervisors are extended to CLEN size so that they are able to hold capabilities. The remainder of this chapter describes these changes in detail.

8.1. Hypervisor Status Register (hstatus)

The hstatus register operates as described in (RISC-V, 2023) except for the VSXL field that controls the value of XLEN for VS-mode (known as VSXLEN).

The encoding of the VSXL field is the same as the MXL field of **misa**. Only 1 and 2 are supported values for VSXL. When the implementation supports Zcheripurecap (but not Zcherihybrid), then hstatus's VSXL must be read-only as described in mstatus for mstatus.SXL. When the implementation supports both Zcheripurecap and Zcherihybrid, then VSXL behaves as described in Section 7.8.1 for mstatus.SXL.

The VSBE field determines controls the endianness of explicit memory accesses from VS-mode and implicit memory accesses to VS-level memory management data structures. VSBE=0 indicates little endian and VSBE=1 is big endian. VSBE must be read-only and equal to MBE when the implementation only supports Zcheripurecap. VSBE is optionally writeable when Zcherihybrid is also supported.

8.2. Hypervisor Environment Configuration Register (henvcfg)

The henvcfg register operates as described in the RISC-V Privileged Specification. A new enable bit is added to henvcfg when the implementation supports Zcherihybrid as shown in Figure 43.

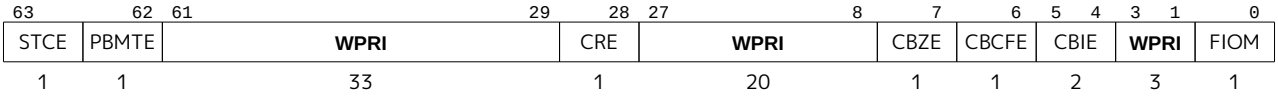


*Figure 43. Hypervisor environment configuration register (***henvcfg***)*

The CHERI Register Enable (CRE) bit controls whether explicit access to CHERI registers is permitted when V=1. When henvcfg.CRE=1 and menvcfg.CRE=1 and mseccfg.CRE=1, CHERI registers can be read and written by VS-mode and VU-mode. When henvcfg.CRE=0, CHERI registers are disabled in VS-mode and VU-mode as described in Section 7.7.

The reset value is 0.

8.3. Hypervisor Trap Value Register (htval)

The htval register operates as described in (RISC-V, 2023).

htval is updated following the same rules as mtval for CHERI exceptions which are taken in HS-mode.

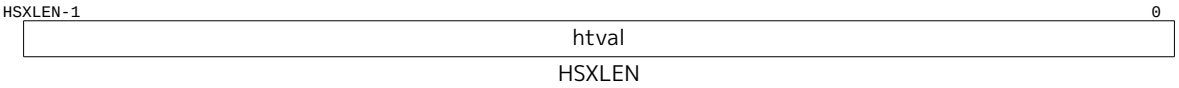


Figure 44. Hypervisor trap value register

8.4. Hypervisor Trap Value Register 2 (htval2)

The htval2 register is an HSXLEN-bit read-write register, which is added as part of Zcheripurecap when the hypervisor extension is supported. Its CSR address is 0x64b.

htval2 is updated following the same rules as mtval2 for CHERI exceptions which are taken in HS-mode.

The fields are identical to mtval2 for CHERI exceptions.

htval2 is not a standard RISC-V CSR, but mtval2 is.

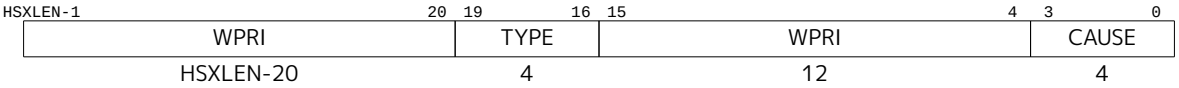


Figure 45. Hypervisor trap value register 2

8.5. Virtual Supervisor Status Register (vsstatus)

The vsstatus register operates as described in (RISC-V, 2023) except for the UXL field that controls the value of XLEN for VU-mode.

The encoding of the UXL field is the same as the MXL field of **misa**. Only 1 and 2 are supported values for UXL. When the implementation supports Zcheripurecap (but not Zcherihybrid), then vsstatus.UXL must be read-only as described in mstatus for mstatus.UXL. When the implementation supports both Zcheripurecap and Zcherihybrid, then UXL behaves as described in Section 7.8.1 for mstatus.UXL.

8.6. Virtual Supervisor Trap Vector Base Address Register (vstvec)

The vstvec register is as defined in (RISC-V, 2023). It is the VSXLEN-bit read/write register that is the VS mode's version of the supervisor register stvec.

Figure 46. Virtual supervisor trap vector base address register

8.7. Virtual Supervisor Trap Vector Base Address Capability Register (vstvecc)

The vstvecc register is a renamed extension of vstvec that is able to hold a capability. Its reset value is the Infinite capability.

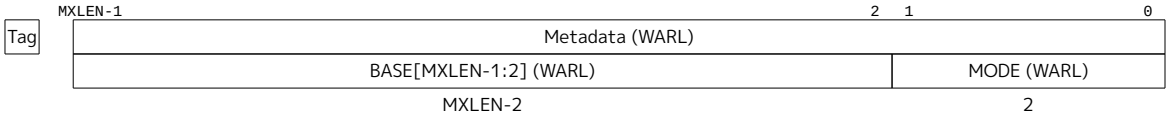


Figure 47. Virtual supervisor trap vector base address capability register

The handling of vstvecc is otherwise identical to mtvecc, but in virtual supervisor mode.

8.8. Virtual Supervisor Scratch Register (vsscratch)

The vsscratch register is as defined in (RISC-V, 2023). It is a VSXLEN read/write register that is VS-mode's version of supervisor register sscratch. vsscratch is extended into vsscratchc.

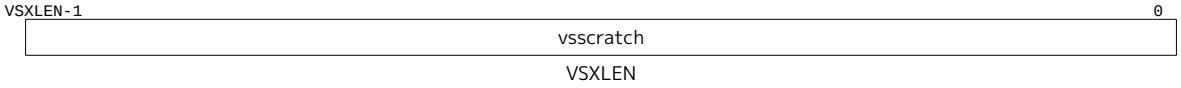


Figure 48. Virtual supervisor scratch register

8.9. Virtual Supervisor Scratch Register (vsscratchc)

The vsscratchc register is a renamed version of vsscratch that is able to hold a capability.

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

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

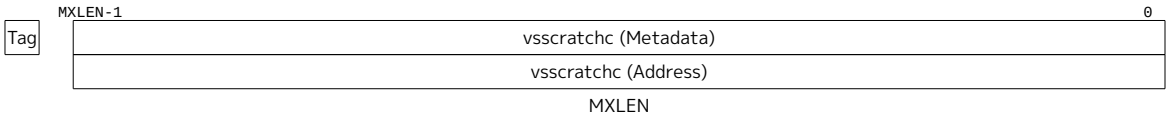


Figure 49. Virtual supervisor scratch capability register

8.10. Virtual Supervisor Exception Program Counter (vsepc)

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

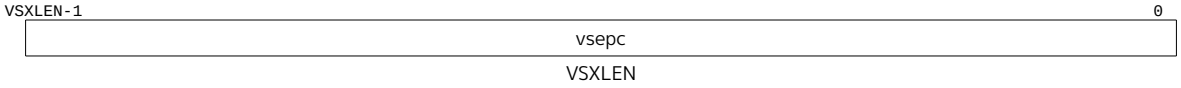


Figure 50. Virtual supervisor exception program counter

8.11. Virtual Supervisor Exception Program Counter Capability (vsepcc)

The vsepcc register is a renamed extension of vsepc that is able to hold a capability. Its reset value is the Infinite capability.

As shown in Table 50, vsepcc is an executable vector, so it need not be able to hold all possible invalid addresses. Additionally, the capability in vsepcc is unsealed when it is installed in pcc on execute of an SRET instruction. The handling of vsepcc is otherwise identical to mepcc, but in virtual supervisor mode.

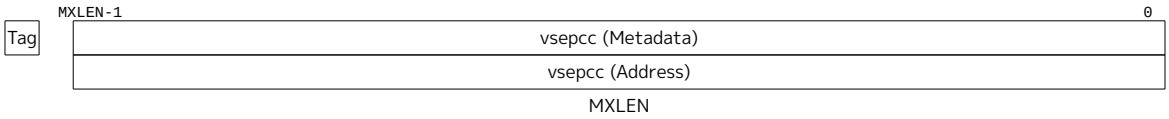


Figure 51. Virtual supervisor exception program counter capability

8.12. Virtual Supervisor Cause Register (vscause)

The vscause register is as defined in (RISC-V, 2023). It must additionally support the new exception code for CHERI exceptions that scause supports.

8.13. Virtual Supervisor Trap Default Capability Register (vstdc)

The vstdc register is a capability width read/write register that is VS-mode's version of supervisor register stdc. This register is only present when the implementation supports Zcherihybrid.

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

Access to this CSR is illegal if CHERI register access is disabled for the current privilege.

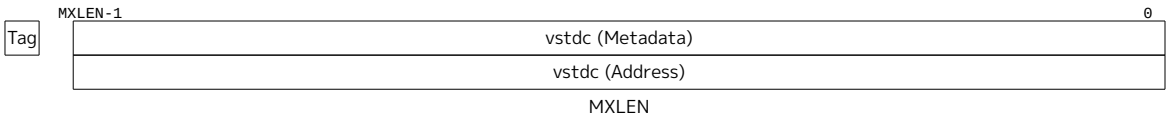


Figure 52. Virtual supervisor trap default capability register

8.14. Virtual Supervisor Trap Value Register (vstval)

The vstval register is a VSXLEN-bit read-write register.

vstval is updated following the same rules as mtval for CHERI exceptions which are taken in VS-mode.

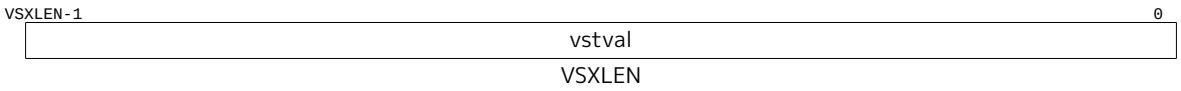


Figure 53. Virtual supervisor trap value register

8.15. Virtual Supervisor Trap Value Register 2 (vstval2)

The vstval2 register is a VSXLEN-bit read-write register, which is added as part of Zcheripurecap when the hypervisor extension is supported. Its CSR address is 0x24b.

vstval2 is updated following the same rules as mtval2 for CHERI exceptions which are taken in VS-mode.

The fields are identical to mtval2 for CHERI exceptions.

vstval2 is not a standard RISC-V CSR, but mtval2 is.

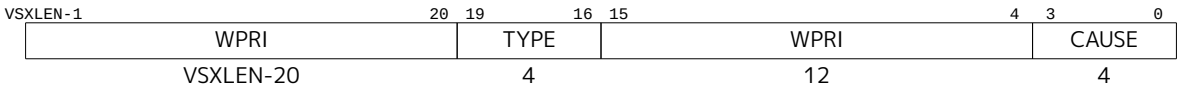


Figure 54. Virtual supervisor trap value register 2

8.16. Existing Hypervisor Load and Store Instructions

The hypervisor extension defines several integer load and store instructions (such as HLV.W, HSV.W and HLVX.WU) that transfer the amount of integer data described in (RISC-V, 2023) between the registers and memory as though V=1. These instructions change behaviour depending on the CHERI execution mode although the instruction's encoding remains unchanged.

When in *Capability Pointer Mode*, the hypervisor load and store instructions behave as described in Section 4.4. In *Integer Pointer Mode*, the instructions behave as described in (RISC-V, 2023) and rely on an **x** register operand providing the effective address for the memory access; the capability authorising the memory access is ddc.

The exception cases remain as described in Section 4.4 regardless of the CHERI execution mode.

8.17. Hypervisor Load and Store Capability Instructions

Hypervisor virtual-machine load (HLV.C) and store (HSV.C) capability instructions read or write CLEN bits from memory as though V=1. These instructions change behaviour depending on the CHERI execution mode although the instruction's encoding remains unchanged.

When in *Capability Pointer Mode*, the hypervisor load and store capability instructions behave as described in Section 4.4. In *Integer Pointer Mode*, the instructions behave as rely on an **x** register operand providing the effective address for the memory access and the capability authorising the memory access is ddc.

Chapter 9. Integrating Zcheripurecap and Zcherihybrid with the Vector Extension

The RISC-V vector (V) extension is orthogonal to CHERI because the vector registers only hold integer or floating-point data. The vector registers are *not* extended to hold capabilities.

A future extension may allow tags to be stored in vector registers. Until that time, vector load and store instructions must not be used to implement generic memory copying in software, such as the `memcpy()` *standard C library function, because the vector registers do not hold capabilities, so the tags of any copied capabilities will be set to 0 in the destination memory.*

Vector loads and stores all follow the behaviour as described in Section 4.4.3.

The assembly syntax of all vector loads and stores are updated in *Capability Pointer Mode*, so that the address operand becomes a **c** operand instead of an **x** operand.

According to the vector extension (RISC-V, 2021) only *active* elements are accessed or updated in memory. Therefore only *active* elements are subject to CHERI exception checks. If a vector load or store has no *active* elements then *no* CHERI fault will be taken.

This is consistent with other exceptions such as page faults which are only taken on *active* elements.

In the case of fault-only-first loads, only the first element will cause a CHERI length violation. If a later element causes a length violation, then it will be treated the same way as a page fault and vl will be reduced. All other CHERI exceptions, such as tag and permission violations are checked on the first element, and so will be taken as expected.

Indexed loads in Capability Pointer Mode check the bounds of every access against the authority capability in `cs1`*. Therefore the approach of having a zero base register and treating every element as an absolute address may not work well in this mode.*

Chapter 10. Integrating Zcheripurecap and Zcherihybrid with Pointer Masking

The pointer masking extensions Smmpm, Smnpm, SSnpm, Sspm and Supm are compatible with Zcherihybrid.

For instructions using integer addresses (e.g. loads/stores in *Integer Pointer Mode*), they are interpreted as being XLEN-wide, and may be subject to pointer masking. All data accesses are checked against ddc which is unaffected by pointer masking. Therefore no capability bounds encoding is affected.

For instructions using capabilities (e.g. loads/stores in *Capability Pointer Mode*), the final access address is subject to pointer masking, but the computed bounds are not. The entire address field, including any bits representing the pointer mask, are used for bounds calculation. When pointer masking is enabled, the dereferenced address has the masked bits replaced by sign extension before the bounds check.

This scheme doesn't seem very useful, but the problem is the dynamic configuration of pointer masking which can arbitrarily update the meaning of the address within the capability, so the full address field must be used to calculate bounds. There is future work required to determine a more useful way of applying pointer masking to capabilities.

Chapter 11. "Zstid" Extension for Thread Identification

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

11.1. Control and Status Registers (CSRs)

Zstid adds new CSRs to implement a trusted thread identifier (TID) used in compartmentalization. These CSRs are listed in Table 32, Table 33, Table 34 and Table 35.

Table 32. Added machine-mode CSRs in Zstid

Zstid CSR	Address	Prerequisites	Read-Permission	Write-Permission	Description
mtid	0x780	M-mode	M	M, ASR-permission	Machine Thread Identifier

Table 33. Added supervisor-mode CSRs in Zstid

Zstid CSR	Address	Prerequisites	Read-Permission	Write-Permission	Description
stid	0x580	S-mode	S	S, ASR-permission	Supervisor Thread Identifier

Table 34. Added virtual supervisor-mode CSRs in Zstid

Zstid CSR	Address	Prerequisites	Read-Permission	Write-Permission	Description
vstid	0xA80	VS-mode	S	H, ASR-permission	Virtual Supervisor Thread Identifier

Table 35. Added user-mode CSRs in Zstid

Zstid CSR	Address	Prerequisites	Read-Permission	Write-Permission	Description
utid	0x480	U-mode	U	U, ASR-permission	User Thread Identifier

11.2. Machine-Level, Supervisor-Level and Unprivileged CSRs

11.2.1. Machine Thread Identifier (mtid)

The mtid register is an MXLEN-bit read-write register. It is used to identify the current thread in machine mode. The reset value of this register is UNSPECIFIED.

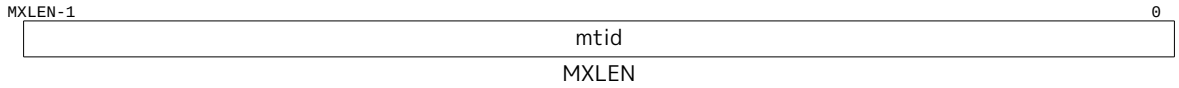


Figure 55. Supervisor thread identifier register

11.2.2. Supervisor Thread Identifier (stid)

The stid register is an SXLEN-bit read-write register. It is used to identify the current thread in supervisor mode. The reset value of this register is UNSPECIFIED.

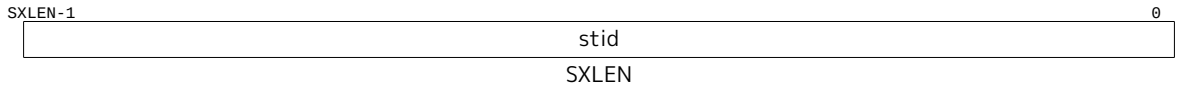


Figure 56. Supervisor thread identifier register

11.2.3. Virtual Supervisor Thread Identifier (vstid)

The vstid register is a VSLEN-bit read-write register. It is VS-mode's version of supervisor register stid used to identify the current thread in virtual supervisor mode. As other Virtual Supervisor registers when V=1, vstid substitutes for the usual stid, so that instructions that normally read or modify stid actually access vstid instead. When V=0, vstid does not directly affect the behaviour of the machine.

The reset value of this register is UNSPECIFIED.

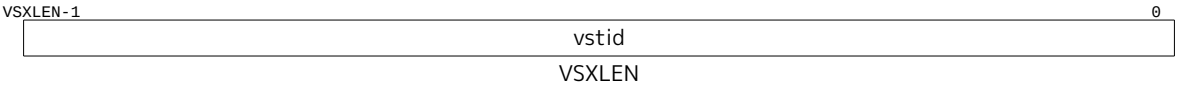


Figure 57. Virtual supervisor thread identifier register

11.2.4. User Thread Identifier (utid)

The utid register is an UXLEN-bit read-write register. It is used to identify the current thread in user mode. The reset value of this register is UNSPECIFIED.

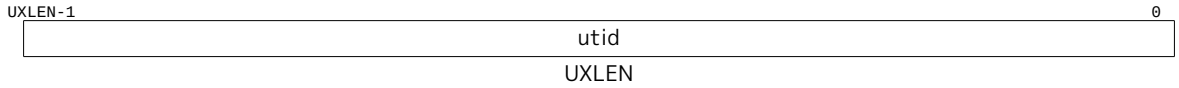


Figure 58. User thread identifier register

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

11.2.5. Machine Thread Identifier Capability (mtidc)

The mtidc register is an CLEN-bit read-write capability register. It is the capability extension of the mtid register. It is used to identify the current thread in machine mode. On reset the tag of mtidc will be set to 0 and the remainder of the data is UNSPECIFIED.

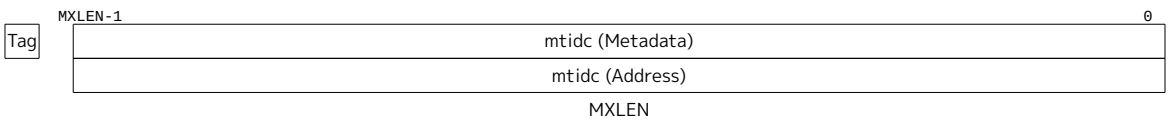


Figure 59. Machine thread identifier capability register

11.2.6. Supervisor Thread Identifier Capability (stidc)

The stidc register is an CLEN-bit read-write capability register. It is the capability extension of the stid register. It is used to identify the current thread in supervisor mode. On reset the tag of stidc will be set to 0 and the remainder of the data is UNSPECIFIED.

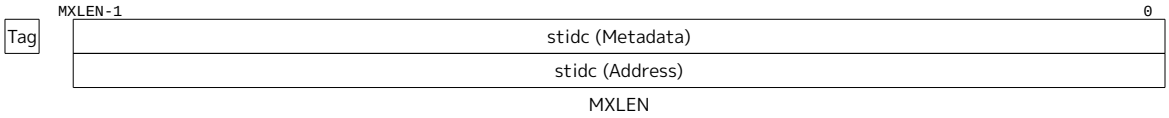


Figure 60. Supervisor thread identifier capability register

11.2.7. Virtual Supervisor Thread Identifier Capability (vstidc)

The vstidc register is a CLEN-bit read-write capability register. It is the capability extension of the stidc register used to identify the current thread in virtual supervisor mode. As other Virtual Supervisor registers when V=1, vstidc substitutes for the usual stidc, so that instructions that normally read or modify stidc actually access vstidc instead. When V=0, vstidc does not directly affect the behaviour of the machine. On reset the tag of vstidc will be set to 0 and the remainder of the data is UNSPECIFIED.

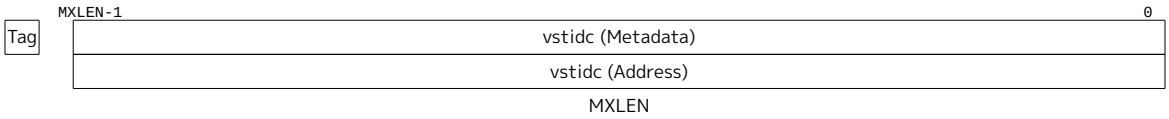


Figure 61. Virtual supervisor thread identifier capability register

11.2.8. User Thread Identifier Capability (utidc)

The utidc register is an CLEN-bit read-write capability register. It is the capability extension of the utid register. It is used to identify the current thread in user mode. On reset the tag of utidc will be set to 0 and the remainder of the data is UNSPECIFIED.

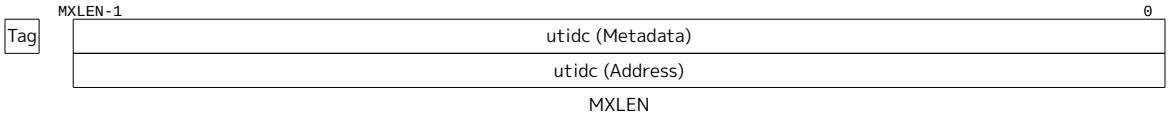


Figure 62. User thread identifier capability register

11.3. "Smstateen/Ssstateen" Integration

The TID bit controls access to the CSRs in Table 33, Table 34 and Table 35 provided by the Zstid extension.

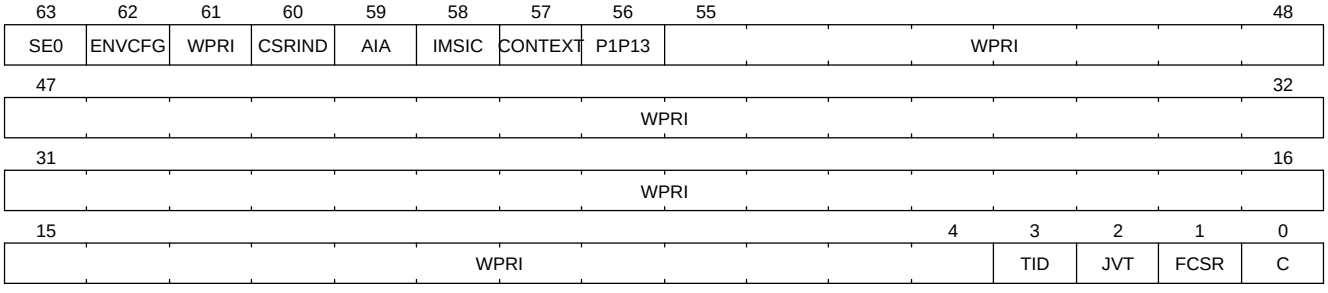


Figure 63. Machine State Enable 0 Register (`mstateen0`)

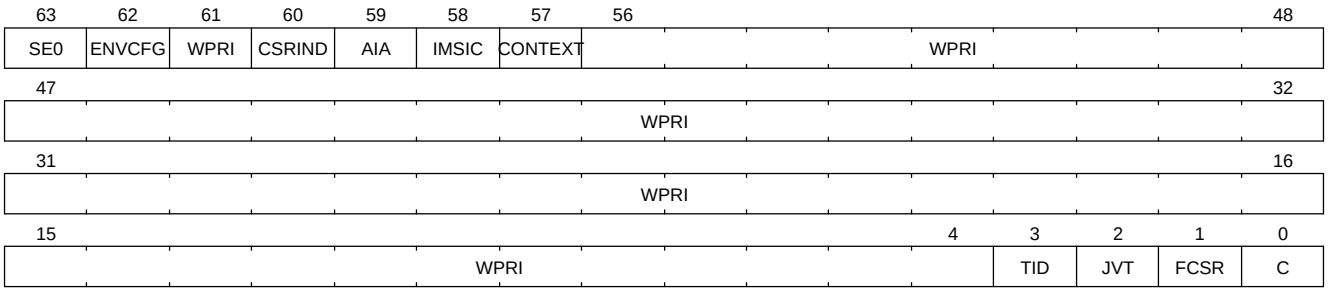


Figure 64. Hypervisor State Enable 0 Register (`hstateen0`)

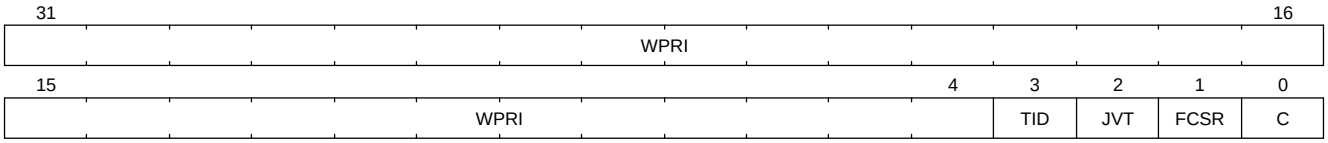


Figure 65. Supervisor State Enable 0 Register (`sstateen0`)

11.4. CHERI Compartmentalization

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

The RISC-V ABI includes a *thread pointer (tp)* register, which is not usable for the purpose of reliably identifying the current thread because the tp register is a general purpose register and can be changed arbitrarily by untrusted code. Therefore, this specification offers three additional CSRs that facilitate a trusted source for the thread ID. All registers are readable from their respective privilege levels and writeable with ASR-permission.

This extension extends mtid, stid, vstid and utid to their respective capability variants mtidc, stidc, vstidc and utidc. This presents software with the freedom to still use these registers with capabilities or leave the metadata untouched and only use the registers to storage integers.

Chapter 12. 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 *Integer Pointer Mode*, every byte of each memory access is bounds checked against ddc
2. In *Integer Pointer Mode*, a minimum length instruction at the target of all indirect jumps is bounds checked against pcc
3. In *Capability Pointer Mode* a minimum length instruction at the target of all indirect jumps is bounds checked against `cs1` (e.g. JALR)
4. A minimum length instruction at the taken target of all direct jumps and conditional branches is bounds checked against pcc regardless of CHERI execution mode

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

12.1. "Zcheripurecap" and "Zcherihybrid" Extensions for CHERI

12.1.1. CMV

Synopsis

Capability move

Mnemonic

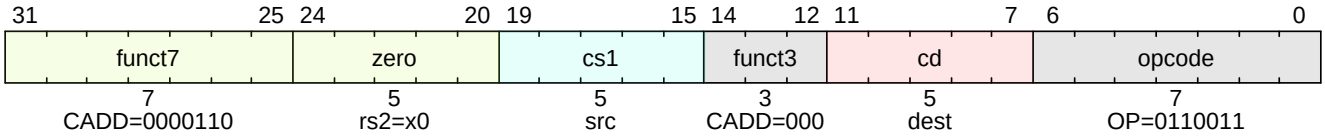
`cmv cd, cs1`

Suggested assembly syntax

`mv cd, cs1`

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

Encoding



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

Description

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

This instruction can propagate tagged capabilities which have malformed bounds, have reserved bits set or have a permission field which cannot be produced by ACPERM.

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.2. MODESW.INT

See MODESW.CAP.

12.1.3. MODESW.CAP

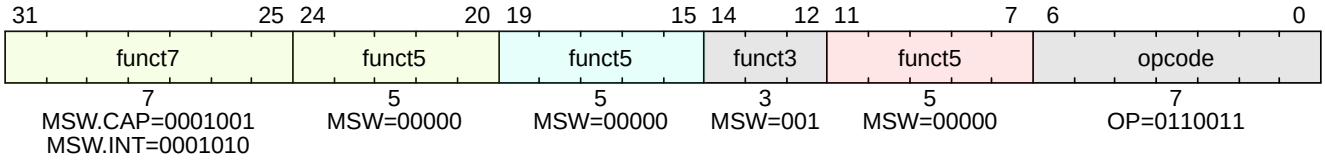
Synopsis

Switch execution mode to *Capability Pointer Mode* (MODESW.CAP), or *Integer Pointer Mode* (MODESW.INT), 32-bit encodings

Mnemonic

```
modesw.cap
modesw.int
```

Encoding



Description

Set the hart's current CHERI execution mode in pcc.

- MODESW.CAP: If the current mode in pcc is *Integer Pointer Mode* (1), then the M-bit in pcc is set to *Capability Pointer Mode* (0). Otherwise no effect.
- MODESW.INT: If the current mode in pcc is *Capability Pointer Mode* (0), then the M-bit in pcc is set to *Integer Pointer Mode* (1). Otherwise no effect.

Executing MODESW.CAP or MODESW.INT from the program buffer in debug mode updates the M-bit of dinfc. The M-bit of dinfc sets the CHERI execution mode for the execution of the next instruction from the program buffer, and is used to control which CHERI execution mode to enter next time debug mode is entered. The CHERI execution mode is **only** *controlled by the M-bit of dinfc in debug mode.*

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcherihybrid

Operation

```
TODO
```

12.1.4. CADDI

See CADD.

12.1.5. CADD

Synopsis

Capability pointer increment

Mnemonic

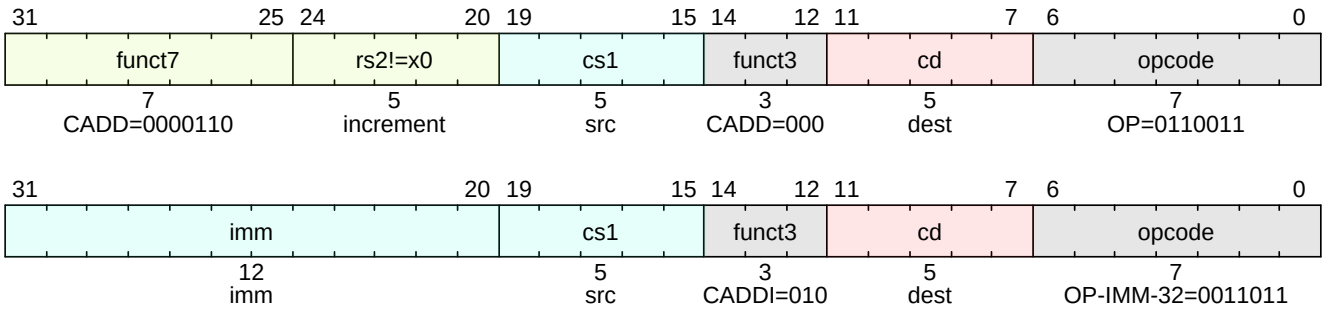
```
cadd cd, cs1, rs2
caddi cd, cs1, imm
```

Suggested assembly syntax

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

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

Encoding



CADD with `rs2=x0` *is decoded as CMV instead, the key difference being that tagged capabilities cannot have their tag cleared by CMV.*

Description

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

For CADD, the address is incremented by the value in `rs2` .
For CADDI, the address is incremented by the immediate value `imm`.

This instruction sets `cd.tag=0` *if* `cs1` *'s bounds are malformed, or if any of the reserved fields are set.*

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation (CADD)

TODO

Operation (CADDI)

TODO

12.1.6. SCADDR

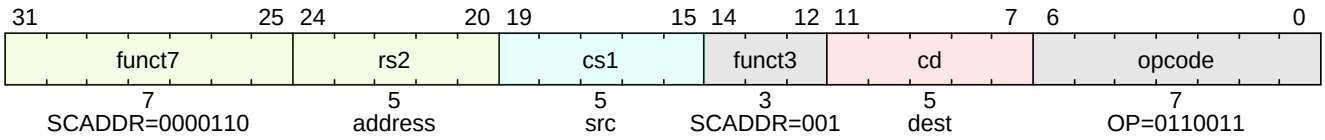
Synopsis

Capability set address

Mnemonic

`scaddr cd, cs1, rs2`

Encoding



Description

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

This instruction sets `cd.tag=0` *if* `cs1` *'s bounds are malformed, or if any of the reserved fields are set.*

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.7. ACPERM

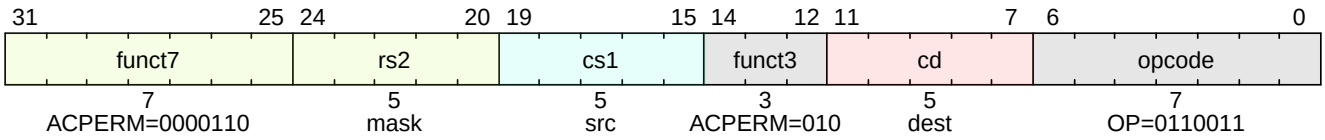
Synopsis

Mask capability permissions

Mnemonics

```
acperm cd, cs1, rs2
```

Encoding



Description

ACPERM performs the following operations:

1. Convert the AP and SDP fields of capability `cs1` into a bit field with the format shown in Figure 66.
2. Calculate the bitwise AND of the bit field with the mask `rs2`.
3. If the AP and M-bit field in `cs1` could not have been produced by ACPERM then clear all AP permissions, and the M-bit to 0. Skip the next step.
4. Clear AP permissions as required to meet the rules below.
5. Encode the AP permissions for RV32 according to Table 4.
6. Copy `cs1` to `cd`, and update the AP and SDP fields with the newly calculated versions.
7. Set `cd.tag=0` if `cs1` is sealed or if any reserved fields of `cs1` are set.

Some combinations of permissions cannot be encoded for MXLEN=32, and are not useful when MXLEN=64. These cases are defined to return useful minimal sets of permissions, which may be no permissions.

Future extensions may allow more combinations of permissions, especially for MXLEN=64. The rules from Table 36 must be followed when removing permissions.

Table 36. ACPERM common rules

Rule	Permission	Only valid if
1 (RV32 only)	ASR-permission	All other permissions are set.
2	C-permission	R-permission or W-permission
3 (RV32 only)	C-permission	R-permission
4 (RV32 only)	X-permission	R-permission
5 (RV32 only)	W-permission	not(C-permission) or LM-permission
6 (RV32 only)	X-permission	W-permission or C-permission
7	EL-permission	C-permission and R-permission
8 (RV32 only)	EL-permission	LM-permission
9	LM-permission	C-permission and R-permission

Rule	Permission	Only valid if
10 (RV32 only)	LM-permission	(W-permission or EL-permission)
11	SL-permission	C-permission
12 (RV32 only)	SL-permission	(LM-permission and (X-permission or W-permission))
13 (RV32 only)	X-permission	(C-permission and LM-permission and EL-permission and (SL-permission == ∞)) or (not(C-permission and not(LM-permission) and not(EL-permission) and (SL-permission==0)))[1]
14	ASR-permission	X-permission
15	M-bit	X-permission

[1] All the listed permissions in the set are either minimum or maximum.

The behaviour of currently illegal combinations from Table 36 is to clear the permission if invalid (or in the case of SL-permission set it to 0 (*local*)).

- For RV64 all such combinations may be redefined by future extensions.
- The RV32 only rules are added because they remove combinations which do not meet the encoding requirements for Table 4, or Table 28 if Zcherilevels is implemented.

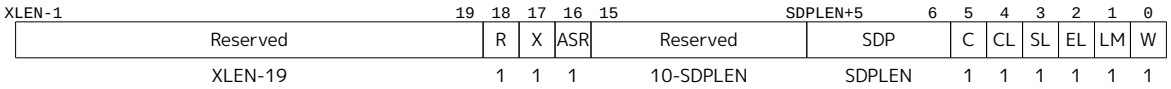


Figure 66. Capability permissions bit field

The EL, SL and CL fields are only defined if the implementation supports Zcherilevels.

Even though being included here CL is not considered an architectural permission.

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

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

12.1.8. SCMODE

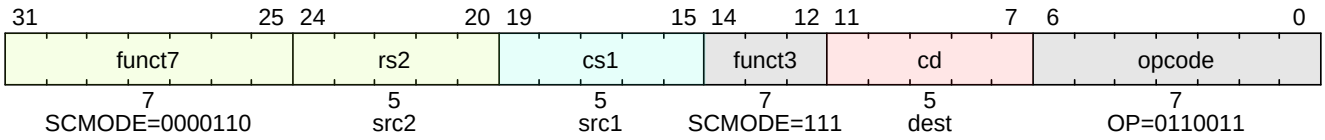
Synopsis

Capability set CHERI execution mode

Mnemonic

```
scmode cd, cs1, rs2
```

Encoding



Description

Copy `cs1` to `cd`. Clear `cd.tag` if `cs1` is sealed. Update the M-bit of `cd` to *Capability Pointer Mode* if the least significant bit of `rs2` is 0 and to *Integer Pointer Mode* if the bit is 1 provided that the following conditions are met, otherwise do not update the M-bit:

1. X-permission is set
2. The existing permissions can be produced by ACPERM

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcherihybrid

Operation

```
TODO
```

12.1.9. SCHI

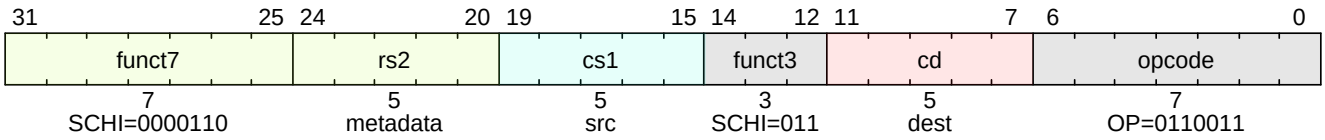
Synopsis

Capability set metadata

Mnemonic

```
schi cd, cs1, rs2
```

Encoding



Description

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

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.10. SCEQ

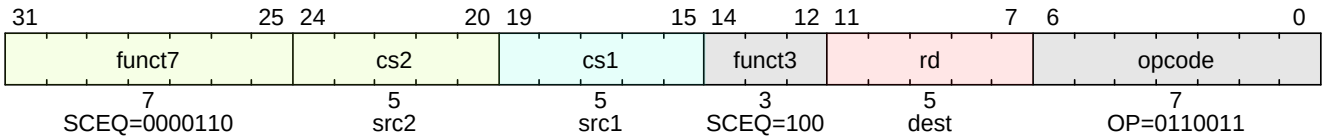
Synopsis

Set if Capabilities are EQual

Mnemonic

`sceq rd, cs1, cs2`

Encoding



Description

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

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.11. SENTRY

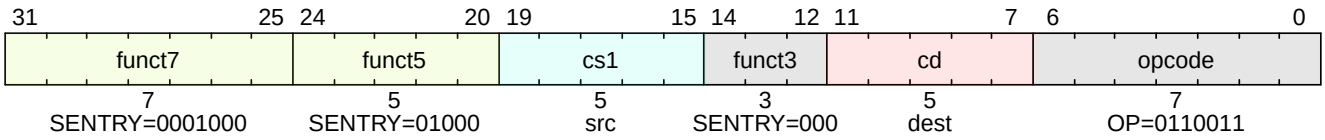
Synopsis

Seal capability as sealed entry.

Mnemonic

```
sentry cd, cs1
```

Encoding



Description

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

The SENTRY instruction may give rise to an illegal instruction fault when the implementation does not support capability type 1 (unrestricted sentry; see Section 3.2.5). This is not the case when the implementation supports the capability encoding described in Chapter 3.

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.12. SCSS

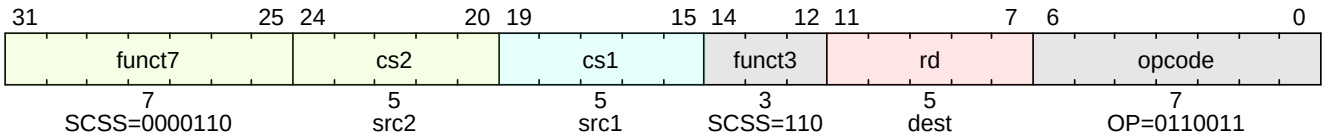
Synopsis

Set Capability Subset

Mnemonic

`scss rd, cs1, cs2`

Encoding



Description

`rd` is set to 1 if:

1. the tag of capabilities `cs1` and `cs2` are equal, and
2. the bounds and permissions of `cs2` are a subset of those of `cs1`, and
3. `cs2` 's Capability Level (CL) is equal to or lower than `cs1` 's
    a. *This is only relevant if Zcherilevels is implemented.*
4. neither `cs1` or `cs2` have bounds which are malformed, and
5. neither `cs1` or `cs2` have any bits set in reserved fields, and
6. neither `cs1` or `cs2` have permissions that could not have been legally produced by ACPERM

Otherwise set `rd` to 0.

The implementation of this instruction is similar to CBLD, although SCSS does not include the sealed bit in the check.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.13. CBLD

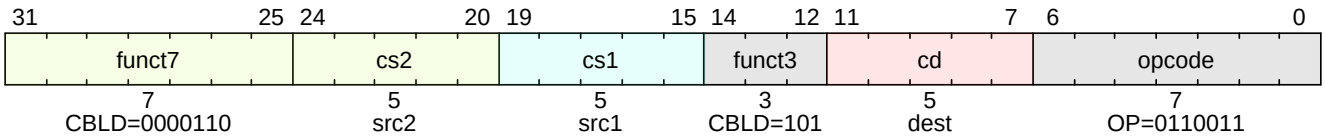
Synopsis

Capability build

Mnemonic

`cbld cd, cs1, cs2`

Encoding



Description

Copy `cs2` to `cd` and set `cd.tag` to 1 if

1. `cs1.tag` is set, and
2. `cs1` 's bounds are not malformed, and all reserved fields are zero, and
3. `cs1` 's permissions could have been legally produced by ACPERM, and
4. `cs1` is not sealed, and
5. `cs2` 's permissions and bounds are equal to or a subset of `cs1` 's, and
6. `cs2` 's Capability Level (CL) is equal to or lower than `cs1` 's, and
    a. *This is only relevant if Zcherilevels is implemented.*
7. `cs2` 's bounds are not malformed, and all reserved fields are zero, and
8. `cs2` 's permissions could have been legally produced by ACPERM, and
9. All reserved bits in `cs2` 's metadata are 0;

Otherwise, copy `cs2` to `cd` and clear `cd` 's tag.

CBLD is typically used alongside SCHI to build capabilities from integer values.

When `cs1` is `c0` this will copy `cs2` to `cd` and clear `cd.tag`. However this may change in future extensions, and so software should not assume `cs1==0` to be a pseudo instruction for tag clearing.

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

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

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

```
//cs1 is untagged or sealed, or if either is underivable
C(cd)       = clearTagIf(cs2_val, not(subset) |
                                  not(cs1_val.tag) |
                                  isCapSealed(cs1_val));
RETIRE_SUCCESS
```

Operation

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

12.1.14. GCTAG

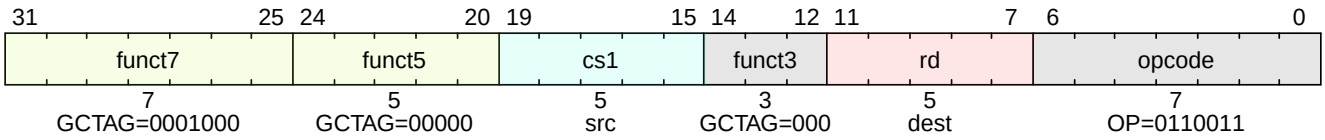
Synopsis

Capability get tag

Mnemonic

```
gctag rd, cs1
```

Encoding



Description

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

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.15. GCPERM

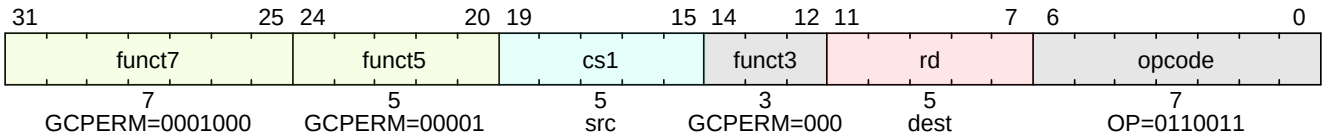
Synopsis

Capability get permissions

Mnemonic

```
gcperm rd, cs1
```

Encoding



Description

If MXLEN=32 unpack permissions from the format in Table 4.

Convert the unpacked AP permissions as well as the SDP fields of capability `cs1` into a bit field, with the format shown in Figure 67, and write the result to `rd`. A bit set to 1 in the bit field indicates that `cs1` grants the corresponding permission.

If the AP field cannot be produced by ACPERM then all architectural permission bits in `rd` are set to 0.

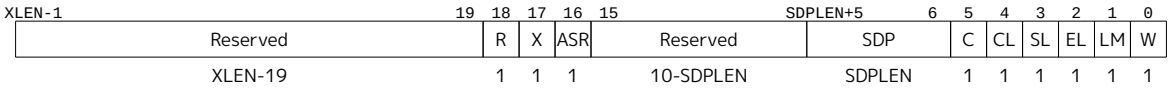


Figure 67. Capability permissions bit field

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO: The encoding of permissions changed.

12.1.16. GCHI

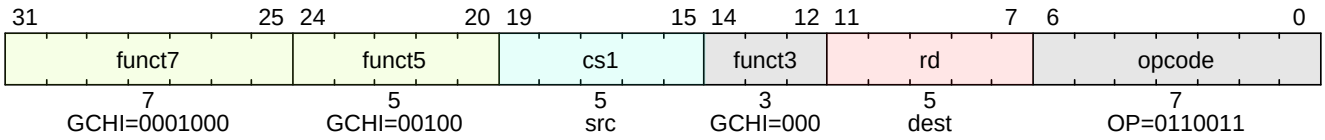
Synopsis

Capability get metadata

Mnemonic

`gchi rd, cs1`

Encoding



Description

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

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.17. GCBASE

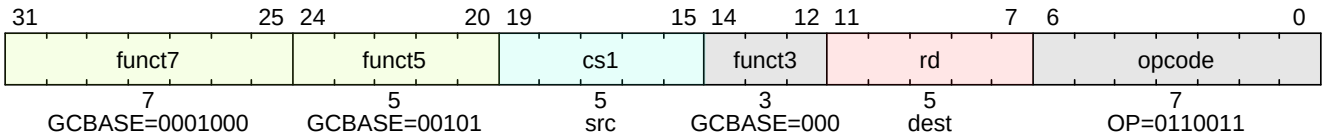
Synopsis

Capability get base address

Mnemonic

```
gcbase rd, cs1
```

Encoding



Description

Decode the base integer address from `cs1` 's bounds and write the result to `rd`. It is not required that the input capability `cs1` has its tag set to 1.

If `cs1` 's bounds are malformed then the bounds decode as zero, which causes this instruction to return zero.

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.18. GCLEN

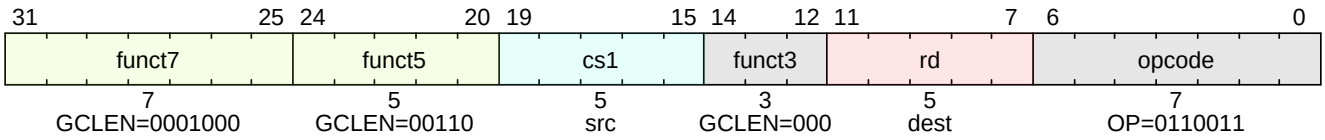
Synopsis

Capability get length

Mnemonic

```
gclen rd, cs1
```

Encoding



Description

Calculate the length of `cs1` 's bounds and write the result in `rd`. The length is defined as the difference between the decoded bounds' top and base addresses i.e. `top - base`. It is not required that the input capability `cs1` has its tag set to 1. GCLEN outputs 0 if `cs1` 's bounds are malformed (see Section 3.2.6.3), and 2^{MXLEN}-1 if the length of `cs1` is 2^{MXLEN}.

If `cs1` 's bounds are malformed then the bounds decode as zero, which causes this instruction to return zero.

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.19. GCMODE

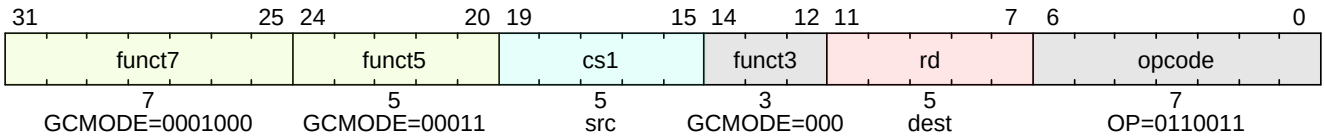
Synopsis

Capability get CHERI execution mode

Mnemonic

```
gcmode rd, cs1
```

Encoding



Description

Decode the CHERI execution mode from the capability in `cs1` and write the result to `rd`. It is not required that `cs1` has its tag set to 1. The output in `rd` is 0 if the capability in `cs1` does not have X-permission set or the AP field cannot be produced by ACPERM; otherwise, the output is 0 if `cs1` 's CHERI execution mode is *Capability Pointer Mode* or 1 if the mode is *Integer Pointer Mode*.

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcherihybrid

Operation

```
TODO
```

12.1.20. GCTYPE

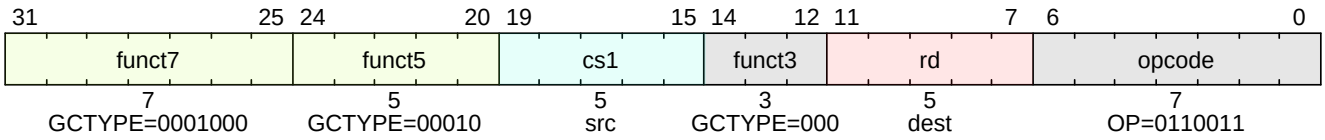
Synopsis

Capability get type

Mnemonic

```
gctype rd, cs1
```

Encoding



Description

Decode the architectural capability type from `cs1` and write the result to `rd`. It is not required that the input capability `cs1` has its tag set to 1.

While the architectural capability type maps directly to the value of the `CT` capability bit in Zcheripurecap, future extensions may define an alternate mapping. Therefore, software should always use GCTYPE to obtain the capability type rather than directly reading the high bits of the capability using GCHI.

Table 37. Capability types in Zcheripurecap

Type	Hardware interpretation
0	Unsealed capability
1	Sentry capability

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.21. SCBNDSI

See SCBNDS.

12.1.22. SCBNDS

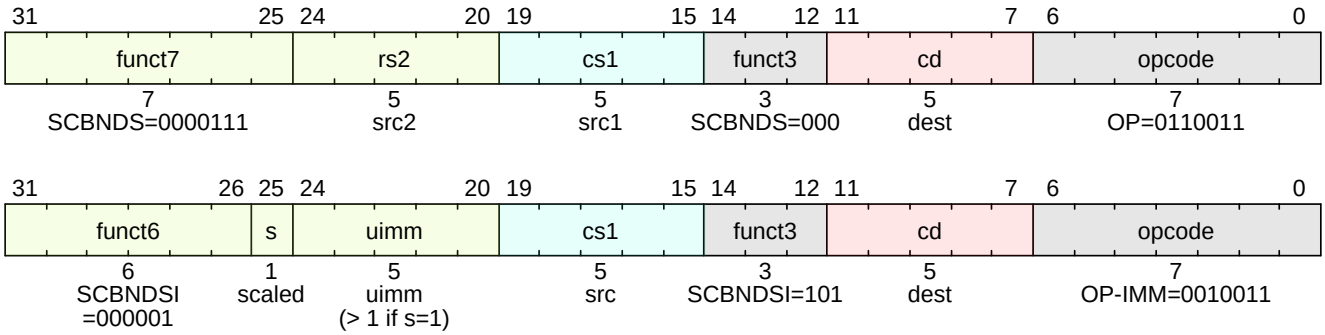
Synopsis

Capability set bounds

Mnemonics

```
scbnds cd, cs1, rs2
scbndsi cd, cs1, uimm
```

Encoding



Description

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

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

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

The SCBNDSI encoding with `s=1` *and* `uimm ≤ 1` *is RESERVED since these immediates can also be encoded with* `s=0`.

This instruction sets `cd.tag=0` *if* `cs1` *'s bounds are malformed, or if any of the reserved fields are set.*

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation for SCBNDS

TODO

Operation for SCBNDSI

TODO

12.1.23. SCBNDSR

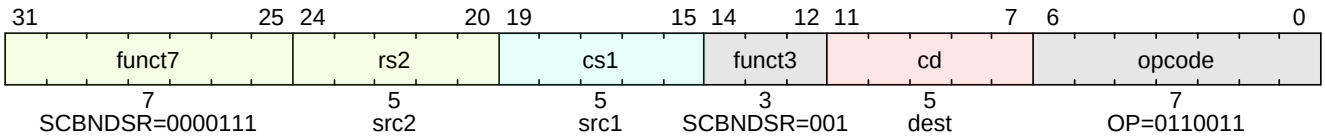
Synopsis

Capability set bounds, rounding up if necessary

Mnemonic

```
scbndsr cd, cs1, rs2
```

Encoding



Description

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

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

This instruction sets `cd.tag=0` if `cs1` 's bounds are malformed, or if any of the reserved fields are set.

Prerequisites

Zcheripurecap

Operation for SCBNDSR

TODO

12.1.24. CRAM

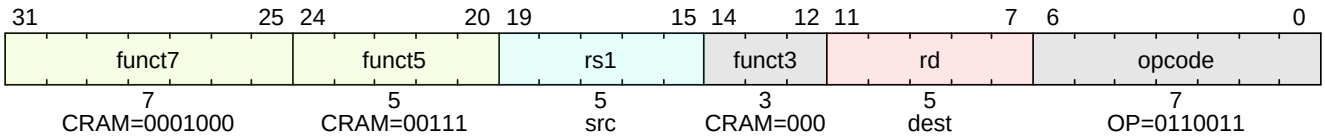
Synopsis

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`.

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites

Zcheripurecap

Operation

TODO

12.1.25. LC

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

Synopsis

Load capability

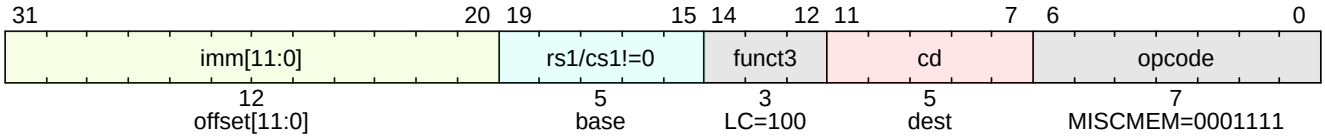
Capability Pointer Mode Mnemonic

```
lc cd, offset(cs1)
```

Integer Pointer Mode Mnemonic

```
lc cd, offset(rs1)
```

Encoding



Capability Pointer 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.

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer 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.

Resulting value of **`cd`**

The tag value written to `cd` is 0 if the tag of the memory location loaded is 0 or the authorizing capability (ddc or `cs1`) does not grant C-permission.

If the authorizing capability does not grant LM-permission, and the tag of `cd` is 1 and `cd` is not sealed, then an implicit ACPERM clearing W-permission and LM-permission is performed to obtain the intermediate permissions on `cd`.

If the authorizing capability does not grant EL-permission, and the tag of `cd` is 1, then an implicit ACPERM clearing EL-permission and restricting the Capability Level (CL) to the level of the authorizing capability is performed to obtain the final permissions on `cd`.

If the authorizing capability does not grant EL-permission, and the tag of `cd` is 1, then an implicit ACPERM restricting the Capability Level (CL) to the level of the authorizing capability is performed. If `cd` is not sealed, this implicit ACPERM also clears EL-permission to obtain the final permissions on `cd` (see Table 30).

Missing LM-permission does not affect untagged values since this could result in surprising bit patterns when copying non-capability data. Similarly, sealed capabilities are

not modified as they are not directly dereferenceable.

Missing EL-permission also affects the level of sealed capabilities since notionally the Capability Level (CL) of a capability is not a permission but rather a data flow label attached to the loaded value. However, untagged values are not affected by EL-permission.

While the implicit ACPERM introduces a dependency on the loaded data, microarchitectures can avoid this by deferring the actual masking of permissions until the loaded capability is dereferenced or the metadata bits are inspected using GCPERM or GCHI.

This instruction can propagate tagged capabilities which have malformed bounds, have reserved bits set or have a permission field which cannot be produced by ACPERM.

Exceptions

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

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites for *Capability Pointer Mode*

Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zcherihybrid

LC Operation

TODO

12.1.26. SC

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

Synopsis

Store capability

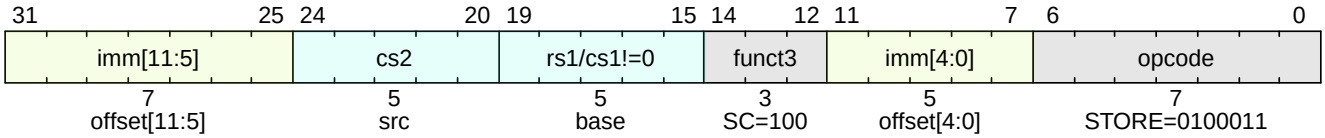
Capability Pointer Mode Mnemonic

```
sc cs2, offset(cs1)
```

Integer Pointer Mode Mnemonic

```
sc cs2, offset(rs1)
```

Encoding



Capability Pointer 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.

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer 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.

Tag of the written capability value

The capability written to memory has the tag set to 0 if the tag of `cs2` is 0 or if the authorizing capability (ddc or `cs1`) does not grant C-permission.

The stored tag is also set to zero if the authorizing capability does not have SL-permission set but the stored data has a Capability Level (CL) of 0 (*local*).

This instruction can propagate tagged capabilities which have malformed bounds, have reserved bits set or have a permission field which cannot be produced by ACPERM.

Exceptions

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

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set

CAUSE	Reason
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites for *Capability Pointer Mode*

Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zcherihybrid

SC Operation

TODO

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

12.2.1. AUIPC

Synopsis

Add upper immediate to **pc**/pcc

CHERI extensions which use an alternative capability format may choose to redefine the handling of the immediate operand for this instruction in Capability Pointer Mode.

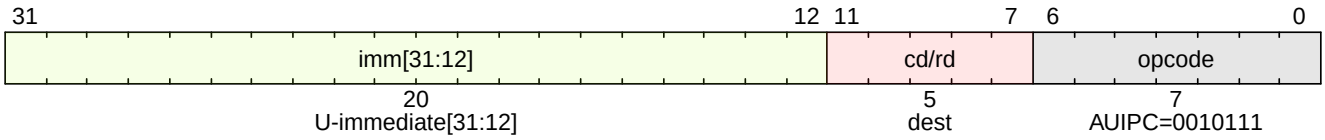
Capability Pointer Mode Mnemonic

```
auipc cd, imm
```

Integer Pointer Mode Mnemonic

```
auipc rd, imm
```

Encoding



Capability Pointer Mode Description

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

Integer Pointer Mode Description

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

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

Prerequisites for *Capability Pointer Mode*

Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zcherihybrid

Operation for AUIPC

TODO

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

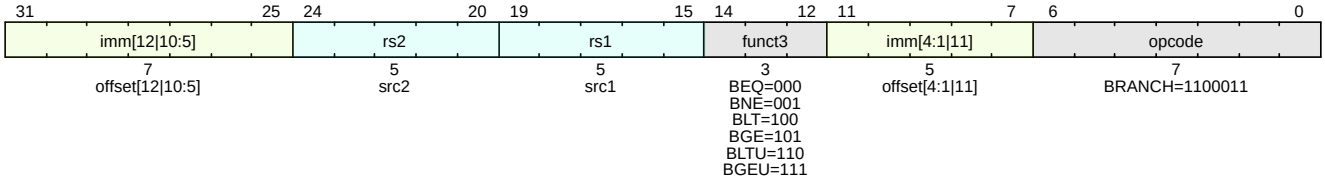
Synopsis

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

Mnemonics

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

Encoding



Description

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

Exceptions

When the target address is not within the pcc's bounds, and the branch is taken, a *CHERI jump or branch fault* is reported in the TYPE field and Bounds violation is reported in the CAUSE field of mtval2 or stval2:

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

Operation

TODO

12.2.3. JR

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

12.2.4. JALR

Synopsis

Jump and link register

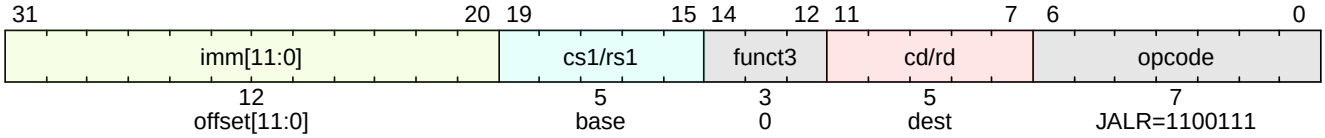
Capability Pointer Mode Mnemonic

```
jalr cd, cs1, offset
```

Integer Pointer Mode Mnemonic

```
jalr rd, rs1, offset
```

Encoding



Capability Pointer Mode Description

JALR allows unconditional, indirect jumps to a target capability. The target capability is unsealed if the `offset` is zero. The target address is obtained by adding the sign-extended 12-bit `offset` to `cs1.address`, 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 is sealed and written to `cd`.

Integer Pointer 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 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 mtval2 or stval2:

CAUSE	*Integer Pointer Mode*	*Capability Pointer Mode*	Reason
Tag violation		✔	`cs1` has tag set to 0, or has any reserved bits set
Seal violation		✔	`cs1` is sealed and the immediate is not 0
Permission violation		✔	`cs1` does not grant X-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	✔	✔	The target address is invalid according to Invalid address conversion

CAUSE	*Integer Pointer Mode*	*Capability Pointer Mode*	Reason
Bounds violation	✔	✔	Minimum length instruction is not within the target capability's bounds, which will fail if `cs1` has malformed bounds in *Capability Pointer Mode*.

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

Prerequisites *Capability Pointer Mode*

Zcheripurecap

Prerequisites *Integer Pointer Mode*

Zcherihybrid

Operation

TBD

12.2.5. J

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

12.2.6. JAL

Synopsis

Jump and link

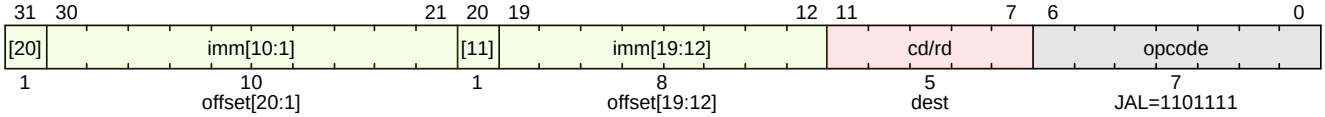
Capability Pointer Mode Mnemonic

```
jal cd, offset
```

Integer Pointer Mode Mnemonic

```
jal rd, offset
```

Encoding



Capability Pointer Mode Description

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

Integer Pointer 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 is written to `rd`.

Exceptions

CAUSE	*Integer Pointer Mode*	*Capability Pointer Mode*	Reason
Invalid address violation	✔	✔	The target address is invalid according to Invalid address conversion
Bounds 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. If they are supported then the value of the pcc in debug mode is UNSPECIFIED by this document.

Prerequisites for *Capability Pointer Mode*

Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zcherihybrid

Operation

TODO

12.2.7. LD

See LB.

12.2.8. LWU

See LB.

12.2.9. LW

See LB.

12.2.10. LHU

See LB.

12.2.11. LH

See LB.

12.2.12. LBU

See LB.

12.2.13. LB

Synopsis

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

Capability Pointer Mode Mnemonics (RV64)

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

Integer Pointer Mode Mnemonics (RV64)

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

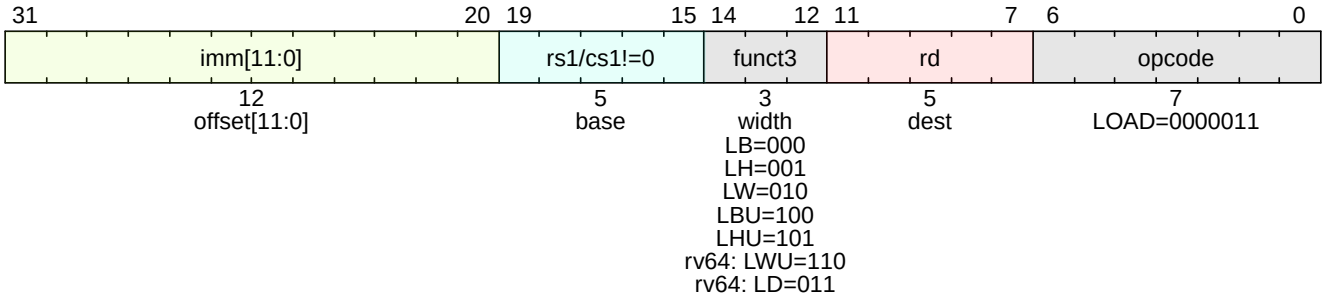
Capability Pointer Mode Mnemonics (RV32)

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

Integer Pointer Mode Mnemonics (RV32)

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

Encoding



Capability Pointer 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`.

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer 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 exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* LD

RV64, Zcheripurecap

Prerequisites for *Integer Pointer Mode* LD

RV64, Zcherihybrid

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

Zcheripurecap, OR
Zcherihybrid

Capability Pointer Mode Operation

TBD

Integer Pointer Mode Operation

TODO

12.2.14. SD

See SB

12.2.15. SW

See SB

12.2.16. SH

See SB

12.2.17. SB

Synopsis

Stores (SD, SW, SH, SB)

Capability Pointer Mode Mnemonics (RV64)

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

Integer Pointer Mode Mnemonics (RV64)

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

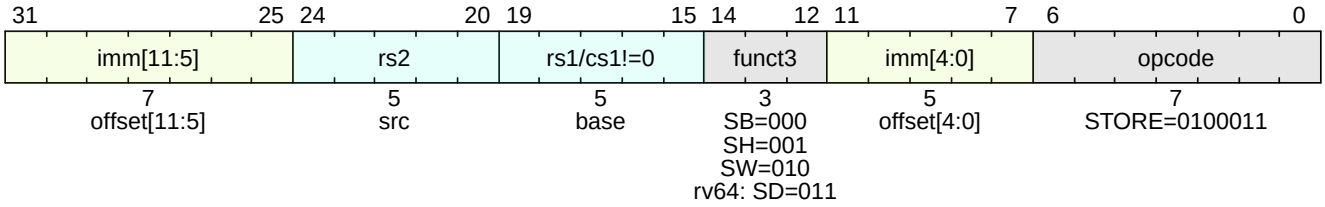
Capability Pointer Mode Mnemonics (RV32)

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

Integer Pointer Mode Mnemonics (RV32)

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

Encoding



Capability Pointer 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.

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer 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 exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* SD

RV64, Zcheripurecap

Prerequisites for *Integer Pointer Mode* SD

RV64, Zcherihybrid

Prerequisites for *Capability Pointer Mode* SW, SH, SB

Zcheripurecap

Prerequisites for *Integer Pointer Mode* SW, SH, SB

Zcherihybrid

Operation

```
TBD
```

12.2.18. SRET

See MRET.

12.2.19. MRET

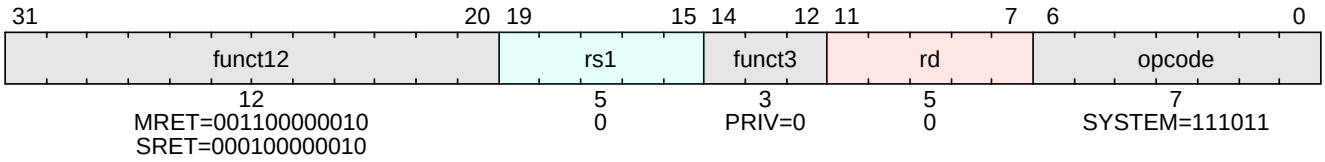
Synopsis

Trap Return (MRET, SRET)

Mnemonics

```
mret
sret
```

Encoding



Description

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

Exceptions

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

Operation

```
TBD
```

12.2.20. DRET

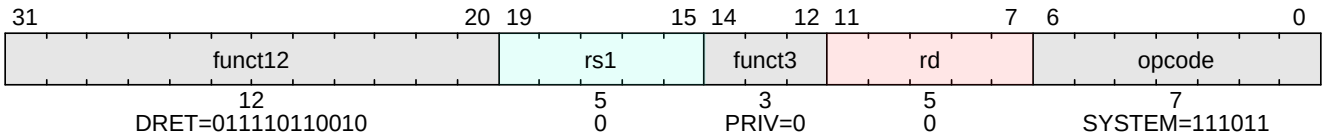
Synopsis

Debug Return (DRET)

Mnemonic

`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
```

12.3. "A" Standard Extension for Atomic Instructions

12.3.1. AMO<OP>.W

See AMO<OP>.D.

12.3.2. AMO<OP>.D

Synopsis

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

Capability Pointer Mode Mnemonics (RV64)

```
amo<op>.[w|d] rd, rs2, offset(cs1)
```

Capability Pointer Mode Mnemonics (RV32)

```
amo<op>.w rd, rs2, offset(cs1)
```

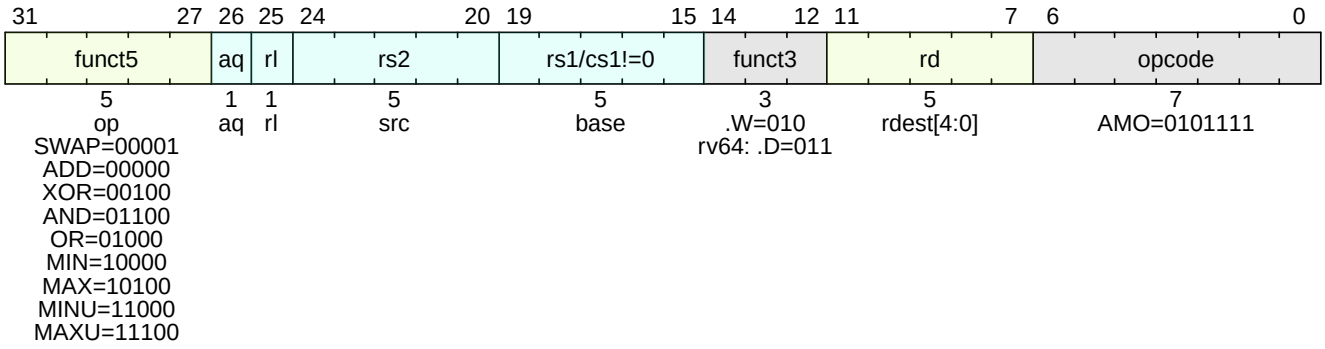
Integer Pointer Mode Mnemonics (RV64)

```
amo<op>.[w|d] rd, rs2, offset(rs1)
```

Integer Pointer Mode Mnemonics (RV32)

```
amo<op>.w rd, rs2, offset(rs1)
```

Encoding



Capability Pointer Mode Description

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

> *Any instance of this instruction with a* `cs1` *of* `c0` *would certainly trap (with a CHERI tag violation), as* `c0` *is defined to always hold a NULL capability. As such, the encodings with a* `cs1` *of* `c0` *are RESERVED for use by future extensions.*

Integer Pointer 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 mtval2 or stval2:

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission or W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

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

Zcheripurecap, and A

Prerequisites for *Integer Pointer Mode* AMO<OP>.W, AMO<OP>.D

Zcherihybrid, and A

Capability Pointer Mode Operation

```
TBD
```

Integer Pointer Mode Operation

TODO

12.3.3. AMOSWAP.C

Synopsis

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

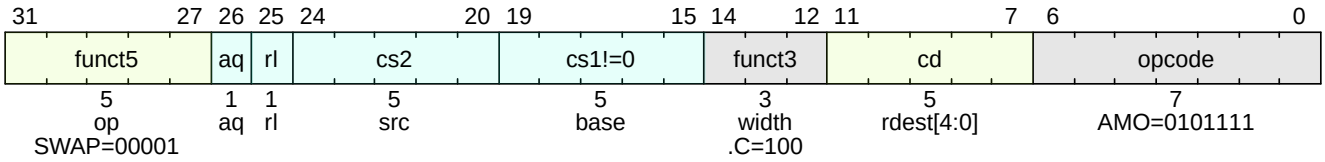
Capability Pointer Mode Mnemonic

```
amoswap.c cd, cs2, offset(cs1)
```

Integer Pointer Mode Mnemonic

```
amoswap.c cd, cs2, offset(rs1)
```

Encoding



Capability Pointer Mode Description

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

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer Mode Description

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

This instruction can propagate tagged capabilities which have malformed bounds, have reserved bits set or have a permission field which cannot be produced by ACPERM.

Permissions

Requires the authorizing 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.

If the authorizing capability does not grant LM-permission, and the tag of `cd` is 1 and `cd` is not sealed, then an implicit ACPERM clearing W-permission and LM-permission is performed to obtain the intermediate permissions on `cd` (see LC).

If the authorizing capability does not grant EL-permission, and the tag of `cd` is 1, then an implicit ACPERM restricting the Capability Level (CL) to the level of the authorizing capability is performed. If `cd` is not sealed, this implicit ACPERM also clears EL-permission to obtain the final permissions on `cd` (see Table 30 and LC).

The stored tag is also set to zero if the authorizing capability does not have SL-permission set but the stored data has a Capability Level (CL) of 0 (see SC).

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 mtval2 or stval2:

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission or W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Exceptions

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites for *Capability Pointer Mode* AMOSWAP.C

Zcheripurecap, and A

Prerequisites for *Integer Pointer Mode* AMOSWAP.C

Zcherihybrid, and A

Operation

TODO

12.3.4. LR.D

See LR.B.

12.3.5. LR.W

See LR.B.

12.3.6. LR.H

See LR.B.

12.3.7. LR.B

Synopsis

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

Capability Pointer Mode Mnemonics (RV64)

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

Capability Pointer Mode Mnemonics (RV32)

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

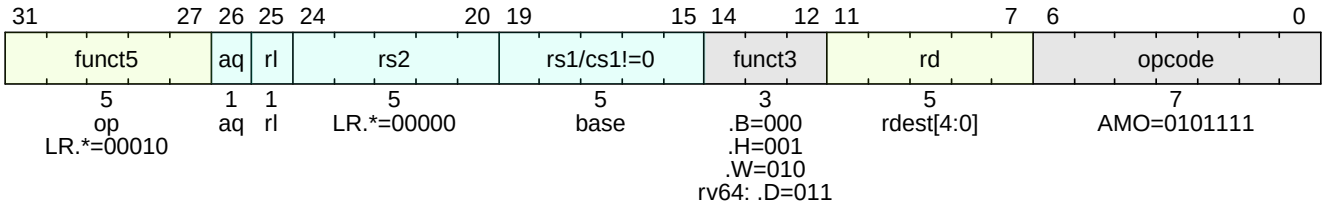
Integer Pointer Mode Mnemonics (RV64)

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

Integer Pointer Mode Mnemonics (RV32)

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

Encoding



Capability Pointer Mode Description

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

> *Any instance of this instruction with a* `cs1` *of* `c0` *would certainly trap (with a CHERI tag violation), as* `c0` *is defined to always hold a NULL capability. As such, the encodings with a* `cs1` *of* `c0` *are RESERVED for use by future extensions.*

Integer Pointer 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 exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM

CAUSE	Reason
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* LR.D

RV64, Zcheripurecap, and A

Prerequisites for *Capability Pointer Mode* LR.W

Zcheripurecap, and A

Prerequisites for *Capability Pointer Mode* LR.H, LR.B

Zabhlrsc, and Zcheripurecap

Prerequisites for LR.D

RV64, Zcherihybrid, and A

Prerequisites for LR.W

Zcherihybrid, and A

Prerequisites for LR.H, LR.B

Zabhlrsc, Zcherihybrid

Operation

```
TBD
```

12.3.8. LR.C

Synopsis

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

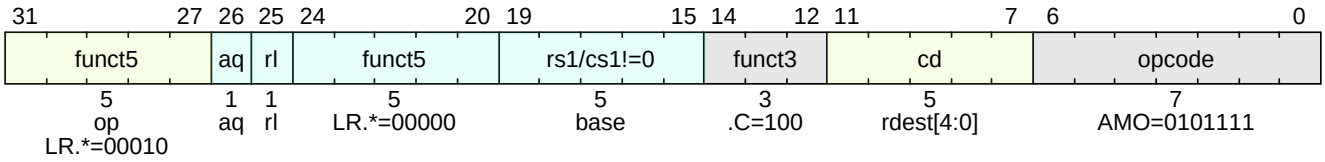
Capability Pointer Mode Mnemonic

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

Integer Pointer Mode Mnemonic

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

Encoding



Capability Pointer 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)).

Any instance of this instruction with a `cs1` *of* `c0` *would certainly trap (with a CHERI tag violation), as* `c0` *is defined to always hold a NULL capability. As such, the encodings with a* `cs1` *of* `c0` *are RESERVED for use by future extensions.*

Integer Pointer 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)).

Resulting value of **`cd`**

The tag value written to `cd` is 0 if the tag of the memory location loaded is 0 or the authorizing capability (ddc or `cs1`) does not grant C-permission.

If the authorizing capability does not grant LM-permission, and the tag of `cd` is 1 and `cd` is not sealed, then an implicit ACPERM clearing W-permission and LM-permission is performed to obtain the intermediate permissions on `cd`.

If the authorizing capability does not grant EL-permission, and the tag of `cd` is 1, then an implicit ACPERM clearing EL-permission and restricting the Capability Level (CL) to the level of the authorizing capability is performed to obtain the final permissions on `cd`.

If the authorizing capability does not grant EL-permission, and the tag of `cd` is 1, then an implicit ACPERM restricting the Capability Level (CL) to the level of the authorizing capability is performed. If `cd` is not sealed, this implicit ACPERM also clears EL-permission to obtain the final permissions on `cd` (see Table 30).

Missing LM-permission does not affect untagged values since this could result in surprising bit patterns when copying non-capability data. Similarly, sealed capabilities are not modified as they are not directly dereferenceable.

Missing EL-permission also affects the level of sealed capabilities since notionally the Capability Level (CL) of a capability is not a permission but rather a data flow label attached to the loaded value. However, untagged values are not affected by EL-permission.

While the implicit ACPERM introduces a dependency on the loaded data, microarchitectures can avoid this by deferring the actual masking of permissions until the loaded capability is dereferenced or the metadata bits are inspected using GCPERM or GCHI.

This instruction can propagate tagged capabilities which have malformed bounds, have reserved bits set or have a permission field which cannot be produced by ACPERM.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites for *Capability Pointer Mode*

Zcheripurecap, and A

Prerequisites for *Integer Pointer Mode*

Zcherihybrid, and A

Operation

```
TBD
```

12.3.9. SC.D

See SC.B.

12.3.10. SC.W

See SC.B.

12.3.11. SC.H

See SC.B.

12.3.12. SC.B

Synopsis

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

Capability Pointer Mode Mnemonics (RV64)

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

Capability Pointer Mode Mnemonics (RV32)

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

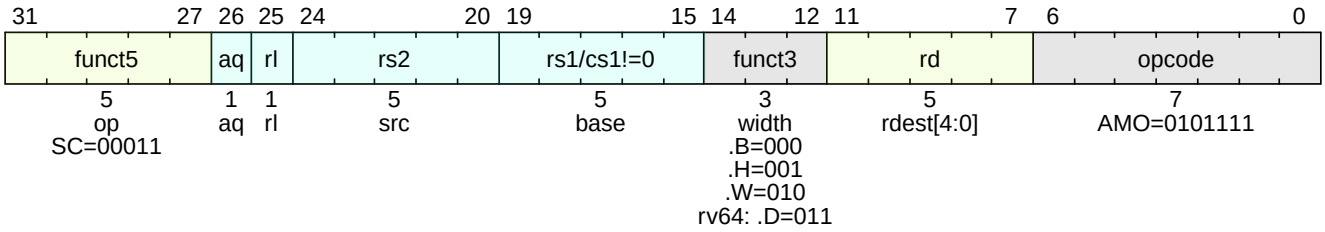
Integer Pointer Mode Mnemonics (RV64)

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

Integer Pointer Mode Mnemonics (RV32)

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

Encoding



Capability Pointer Mode Description

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

Any instance of this instruction with a `cs1` *of* `c0` *would certainly trap (with a CHERI tag violation), as* `c0` *is defined to always hold a NULL capability. As such, the encodings with a* `cs1` *of* `c0` *are RESERVED for use by future extensions.*

Integer Pointer 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 exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM

CAUSE	Reason
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* SC.D
: RV64, and Zcheripurecap, and A

Prerequisites for *Integer Pointer Mode* SC.D
: RV64, and Zcherihybrid, and A

Prerequisites for *Capability Pointer Mode* SC.W
: Zcheripurecap, and A

Prerequisites for *Integer Pointer Mode* SC.W
: Zcherihybrid, and A

Prerequisites for *Capability Pointer Mode* SC.H, SC.B
: Zcheripurecap, and Zabhlrsc

Prerequisites for *Integer Pointer Mode* SC.H, SC.B
: Zcherihybrid, and Zabhlrsc

Operation

```
TBD
```

12.3.13. SC.C

Synopsis

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

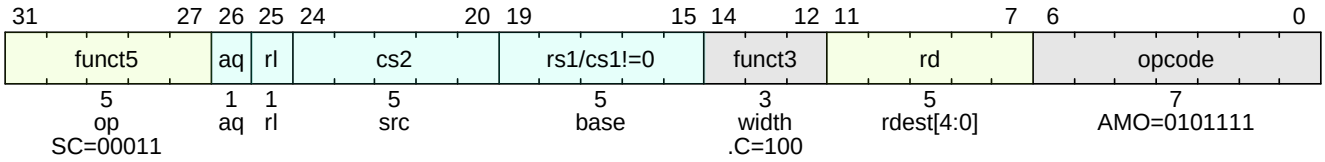
Capability Pointer Mode Mnemonic

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

Integer Pointer Mode Mnemonic

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

Encoding



Capability Pointer 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)).

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer 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)).

Tag of the written capability value

The capability written to memory has the tag set to 0 if the tag of `cs2` is 0 or if the authorizing capability (ddc or `cs1`) does not grant C-permission.

The stored tag is also set to zero if the authorizing capability does not have SL-permission set but the stored data has a Capability Level (CL) of 0 (*local*).

This instruction can propagate tagged capabilities which have malformed bounds, have reserved bits set or have a permission field which cannot be produced by ACPERM.

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 exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites for *Capability Pointer Mode*

Zcheripurecap, and A

Prerequisites for *Integer Pointer Mode*

Zcherihybrid, and A

Operation

```
TBD
```

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

12.4.1. CSRRW

Synopsis

CSR access (CSRRW) 32-bit encodings

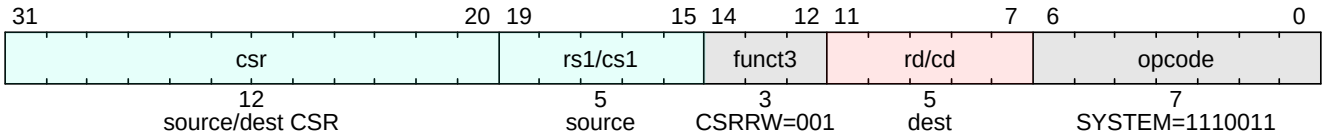
Mnemonic for accessing capability CSRs in *Capability Pointer Mode*

```
csrrw cd, csr, cs1
```

Mnemonic for accessing XLEN-wide CSRs or extended CSRs in *Integer Pointer Mode*

```
csrrw rd, csr, rs1
```

Encoding



Description

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

See Table 47 for a list of CLEN-wide CSRs and Table 48 for the action taken on writing each one.

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

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

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

The assembler pseudo-instruction to write a capability CSR in *Capability Pointer Mode*, `csrw csr, cs1`, is encoded as `csrrw c0, csr, cs1`.

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

When writing `cs1`, if the bounds are malformed, any reserved bits are set, or the permission could not have been produced by ACPERM then clear the tag before writing to the CSR.

Permissions

Accessing privileged CSRs require ASR-permission, including existing RISC-V CSRs, as described in Section 4.5.1. The list of privileged and unprivileged CSRs is shown in (RISC-V, 2023).

Prerequisites for *Capability Pointer Mode*

Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zcherihybrid

Operation

```
TBD
```

12.4.2. CSRRWI

See CSRRCI.

12.4.3. CSRRS

See CSRRCI.

12.4.4. CSRRSI

See CSRRCI.

12.4.5. CSRRC

See CSRRCI.

12.4.6. CSRRCI

Synopsis

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

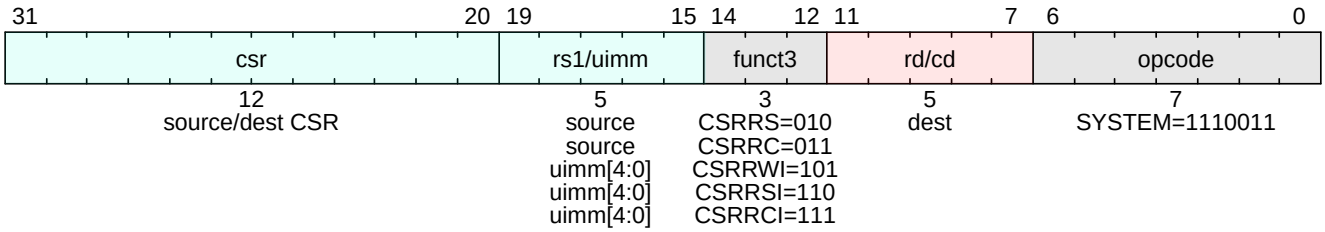
Mnemonics for accessing capability CSRs in *Capability Pointer Mode*

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

Mnemonics for accessing XLEN-wide CSRs or extended CSRs in *Integer Pointer Mode*

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

Encoding



Description

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

For capability CSRs, the full capability is read into `cd` in *Capability Pointer Mode*. In *Integer Pointer Mode*, the address field is instead read into `rd`.

Unlike CSRRW, these instructions only update the address field and the tag as defined in Table 48 when writing capability CSRs regardless of the execution mode. The final address to write to the capability CSR is determined as defined by RISC-V for these instructions.

See Table 47 for a list of capability CSRs and Table 48 for the action taken on writing an XLEN-wide value to each one.

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

The assembler pseudoinstruction to read a capability CSR in Capability Mode, `csrr cd, csr`, is encoded as `csrrs cd, csr, x0`.

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

If the CSR accessed is a capability, and `rs1` is `x0` for CSRRS and CSRRC, or `imm` is 0 for

> *CSRRSI and CSRRCI, then the CSR is not written so no representability check is needed in this case.*

Permissions

Accessing privileged CSRs requires ASR-permission, including existing RISC-V CSRs, as described in Section 4.5.1. The list of privileged and unprivileged CSRs is shown in (RISC-V, 2023).

Prerequisites for *Capability Pointer Mode*

Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zcherihybrid

Operation

```
TBD
```

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

12.5.1. FLD

See FLH.

12.5.2. FLW

See FLH.

12.5.3. FLH

Synopsis

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

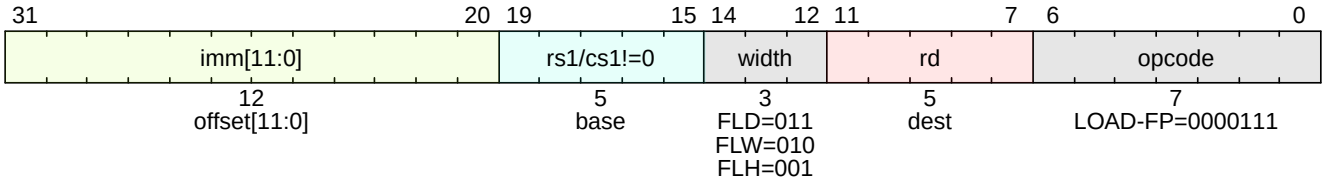
Capability Pointer Mode Mnemonics

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

Integer Pointer Mode Mnemonics

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

Encoding



Capability Pointer Mode Description

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

Any instance of this instruction with a `cs1` *of* `c0` *would certainly trap (with a CHERI tag violation), as* `c0` *is defined to always hold a NULL capability. As such, the encodings with a* `cs1` *of* `c0` *are RESERVED for use by future extensions.*

Integer Pointer Mode Description

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

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* FLD

Zcheripurecap, and D

Prerequisites for *Integer Pointer Mode* FLD
: Zcherihybrid, and D

Prerequisites for *Capability Pointer Mode* FLW
: Zcheripurecap, and F

Prerequisites for *Integer Pointer Mode* FLW
: Zcherihybrid, and F

Prerequisites for *Capability Pointer Mode* FLH
: Zcheripurecap, and Zfhmin or Zfh

Prerequisites for *Integer Pointer Mode* FLH
: Zcherihybrid, and Zfhmin or Zfh

Operation
: TODO

12.5.4. FSD

See FSH.

12.5.5. FSW

See FSH.

12.5.6. FSH

Synopsis

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

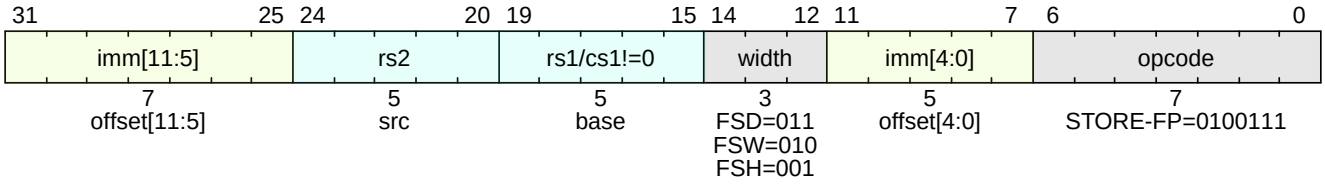
Capability Pointer Mode Mnemonics

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

Integer Pointer Mode Mnemonics

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

Encoding



Capability Pointer Mode Description

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

Any instance of this instruction with a `cs1` *of* `c0` *would certainly trap (with a CHERI tag violation), as* `c0` *is defined to always hold a NULL capability. As such, the encodings with a* `cs1` *of* `c0` *are RESERVED for use by future extensions.*

Integer Pointer Mode Description

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

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* FSD

Zcheripurecap, and D

Prerequisites for *Integer Pointer Mode* FSD
: Zcherihybrid, and D

Prerequisites for *Capability Pointer Mode* FSW
: Zcheripurecap, and F

Prerequisites for *Integer Pointer Mode* FSW
: Zcherihybrid, and F

Prerequisites for *Capability Pointer Mode* FSH
: Zcheripurecap, and Zfh or Zfhmin

Prerequisites for *Integer Pointer Mode* FSH
: Zcherihybrid, and Zfh or Zfhmin

Operation

```
TBD
```

12.6. "C" Standard Extension for Compressed Instructions

One group of 16-bit encodings are remapped to different instructions dependant upon the CHERI execution mode, MXLEN and which extensions are supported.

Zcf and Zilsd are incompatible

Zcd and Zcmp/Zcmt incompatible

12.6.1. RV32

Table 38. 16-bit instruction remapping in Integer Pointer Mode

Encoding		Supported Extensions				
[15:13]	[1:0]	Zca	Zcf	Zcd	Zcmp/ Zcmt	Zilsd
111	00	N/A	C.FSW	N/A	N/A	C.SD
011	00	N/A	C.FLW	N/A	N/A	C.LD
111	10	N/A	C.FSWSP	N/A	N/A	C.SDSP
011	10	N/A	C.FLWSP	N/A	N/A	C.LDSP
101	00	N/A	N/A	C.FSD	reserved[1]	N/A
001	00	N/A	N/A	C.FLD	reserved[1]	N/A
101	10	N/A	N/A	C.FSDSP	Zcmp/Zcmt	N/A
001	10	N/A	N/A	C.FLDSP	reserved[1]	N/A

[1] reserved for future standard Zcm extensions

Table 39. 16-bit instruction remapping in Capability Pointer Mode

Encoding		Supported Extensions				
[15:13]	[1:0]	Zca	Zcf	Zcd	Zcmp/ Zcmt	Zilsd
111	00	C.SC				
011	00	C.LC				
111	10	C.SCSP				
011	10	C.LCSP				
101	00	N/A	N/A	C.FSD	reserved[1]	N/A
001	00	N/A	N/A	C.FLD	reserved[1]	N/A
101	10	N/A	N/A	C.FSDSP	Zcmp/Zcmt	N/A
001	10	N/A	N/A	C.FLDSP	reserved[1]	N/A

[1] reserved for future standard Zcm extensions

12.6.2. RV64

Table 40. 16-bit instruction remapping in Integer Pointer Mode

Encoding		Supported Extensions				
[15:13]	[1:0]	Zca	Zcf	Zcd	Zcmp/ Zcmt	Zilsd
111	00	C.SD	N/A	N/A	N/A	N/A
011	00	C.LD	N/A	N/A	N/A	N/A
111	10	C.SDSP	N/A	N/A	N/A	N/A
011	10	C.LDSP	N/A	N/A	N/A	N/A
101	00	N/A	N/A	C.FSD	reserved[1]	N/A
001	00	N/A	N/A	C.FLD	reserved[1]	N/A
101	10	N/A	N/A	C.FSDSP	Zcmp/Zcmt	N/A
001	10	N/A	N/A	C.FLDSP	reserved[1]	N/A

Table 41. 16-bit instruction remapping in Capability Pointer Mode

Encoding		Supported Extensions				
[15:13]	[1:0]	Zca	Zcf	Zcd	Zcmp/ Zcmt	Zilsd
111	00	C.SD	N/A	N/A	N/A	N/A
011	00	C.LD	N/A	N/A	N/A	N/A
111	10	C.SDSP	N/A	N/A	N/A	N/A
011	10	C.LDSP	N/A	N/A	N/A	N/A
101	00	C.SC				
001	00	C.LC				
101	10	C.SCSP				
001	10	C.LCSP				

12.6.3. C.BEQZ, C.BNEZ

Synopsis

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

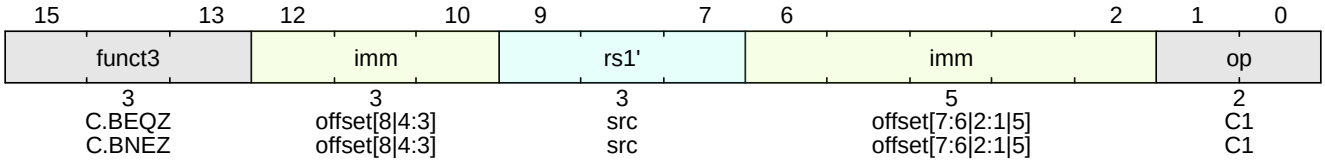
Mnemonics

```
c.beqz rs1', offset
c.bnez rs1', offset
```

Expansions

```
beq rs1', x0, offset
bne rs1', x0, offset
```

Encoding



Exceptions

When the target address is not within the pcc's bounds, and the branch is taken, a *CHERI jump or branch fault* is reported in the TYPE field and Bounds violation is reported in the CAUSE field of mtval2 or stval2:

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

Prerequisites

C or Zca

Operation (after expansion to 32-bit encodings)

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

12.6.4. C.MV

Synopsis

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

Capability Pointer Mode Mnemonic

```
c.mv cd, cs2
```

Capability Pointer Mode Expansion

```
cmv cd, cs2
```

Suggested assembly syntax

```
mv rd, rs2
mv cd, cs2
```

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

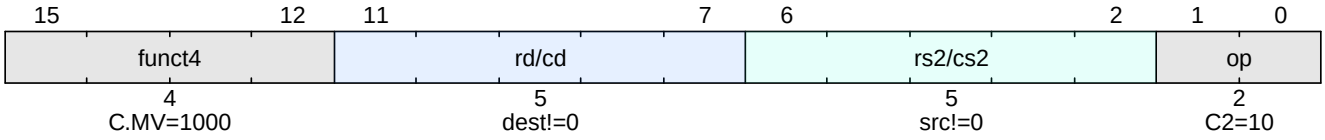
Integer Pointer Mode Mnemonic

```
c.mv rd, rs2
```

Integer Pointer Mode Expansion

```
add rd, x0, rs2
```

Encoding



Capability Pointer Mode Description

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

Integer Pointer Mode Description

Standard RISC-V C.MV instruction.

This instruction can propagate tagged capabilities which have malformed bounds, have reserved bits set or have a permission field which cannot be produced by ACPERM.

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid

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

See CMV

12.6.5. C.ADDI16SP

Synopsis

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

Capability Pointer Mode Mnemonic

```
c.addi16sp imm
```

Capability Pointer Mode Expansion

```
cadd csp, csp, imm
```

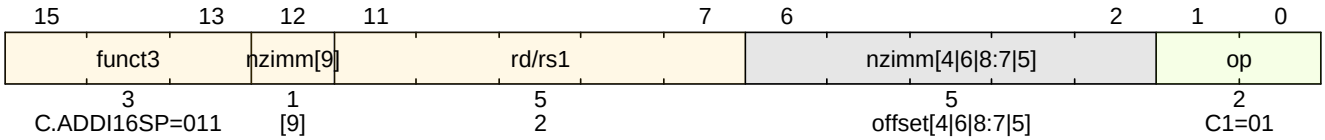
Integer Pointer Mode Mnemonic

```
c.addi16sp imm
```

Integer Pointer Mode Expansion

```
add sp, sp, imm
```

Encoding



Capability Pointer 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.

Integer Pointer Mode Description

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

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid

Capability Pointer Mode Operation

TODO

12.6.6. C.ADDI4SPN

Synopsis

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

Capability Pointer Mode Mnemonic

```
c.addi4spn cd', uimm
```

Capability Pointer Mode Expansion

```
cadd cd', csp, uimm
```

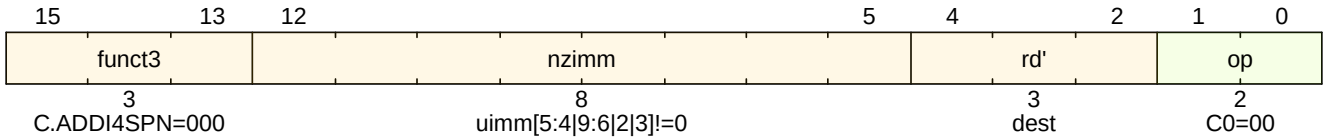
Integer Pointer Mode Mnemonic

```
c.addi4spn rd', uimm
```

Integer Pointer Mode Expansion

```
add rd', sp, uimm
```

Encoding



Capability Pointer Mode Description

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

Integer Pointer Mode Description

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

Prerequisites for C.ADDI4SPN

C or Zca, Zcheripurecap

Prerequisites for C.ADDI4SPN

C or Zca, Zcherihybrid

Capability Pointer Mode Operation

TODO

12.6.7. C.JALR

Synopsis

Jump register with link, 16-bit encodings

Capability Pointer Mode Mnemonic

```
c.jalr c1, cs1
```

Capability Pointer Mode Expansion

```
jalr c1, 0(cs1)
```

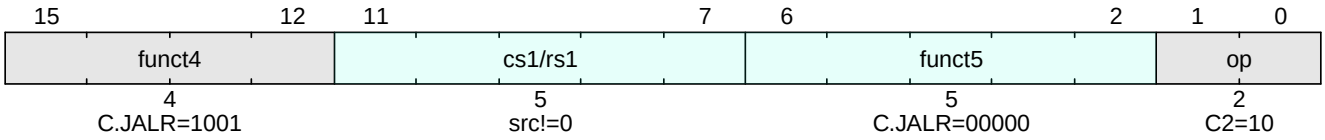
Integer Pointer Mode Mnemonic

```
c.jalr x1, rs1
```

Integer Pointer Mode Expansion

```
jalr x1, 0(rs1)
```

Encoding



Capability Pointer Mode Description

See JALR for execution of the expanded instruction as shown above. Note that the `offset` is zero in the expansion.

Integer Pointer Mode Description

See JALR for execution of the expanded instruction as shown above. Note that the `offset` is zero in the expansion.

Exceptions

See JALR

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

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid

Operation (after expansion to 32-bit encodings)

See JALR

12.6.8. C.JR

Synopsis

Jump register without link, 16-bit encodings

Capability Pointer Mode Mnemonic

```
c.jr cs1
```

Capability Pointer Mode Expansion

```
jalr c0, 0(cs1)
```

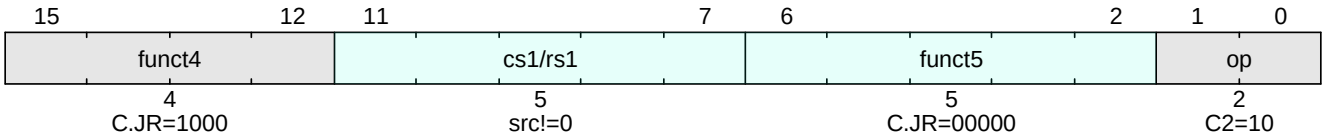
Integer Pointer Mode Mnemonic

```
c.jr rs1
```

Integer Pointer Mode Expansion

```
jalr x0, 0(rs1)
```

Encoding



Capability Pointer Mode Description

See JALR for execution of the expanded instruction as shown above. Note that the `offset` is zero in the expansion.

Integer Pointer Mode Description

See JALR for execution of the expanded instruction as shown above. Note that the `offset` is zero in the expansion.

Exceptions

See JALR

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

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid

Operation (after expansion to 32-bit encodings)

See JALR

12.6.9. C.JAL

Synopsis

Jump with link, 16-bit encodings

Capability Pointer Mode Mnemonic (RV32)

```
c.jal c1, offset
```

Capability Pointer Mode Expansion (RV32)

```
jal c1, offset
```

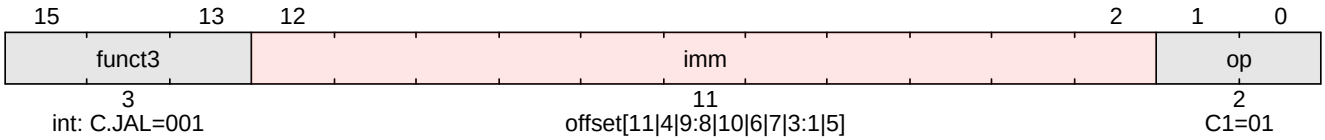
Integer Pointer Mode Mnemonic (RV32)

```
c.jal x1, offset
```

Integer Pointer Mode Expansion (RV32)

```
jal x1, offset
```

Encoding (RV32)



Capability Pointer Mode Description

Link the next linear pcc to `cd` and seal. Jump to pcc.address+offset.

Integer Pointer Mode Description

Set the next PC and link to `rd` according to the standard JAL definition.

Exceptions

See JAL

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

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid

Operation (after expansion to 32-bit encodings)

See JAL

12.6.10. C.J

Synopsis

Jump without link, 16-bit encodings

Mnemonic

```
c.j offset
```

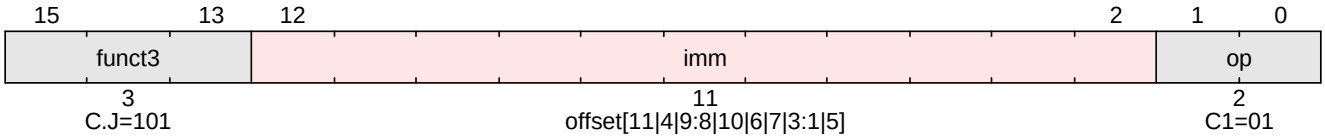
Capability Pointer Mode Expansion

```
jal c0, offset
```

Integer Pointer Mode Expansion

```
jal x0, offset
```

Encoding



Description

Set the next PC following the standard JAL definition.

There is no difference in *Capability Pointer Mode* or *Integer Pointer Mode* execution for this instruction.

Exceptions

See JAL

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

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid

Operation (after expansion to 32-bit encodings)

See JAL

12.6.11. C.LD

See C.LW.

12.6.12. C.LW

Synopsis

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

Capability Pointer Mode Mnemonics (RV64)

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

Capability Pointer Mode Expansions (RV64)

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

Integer Pointer Mode Mnemonics (RV64)

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

Integer Pointer Mode Expansions (RV64)

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

Capability Pointer Mode Mnemonic (RV32)

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

Capability Pointer Mode Expansion (RV32)

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

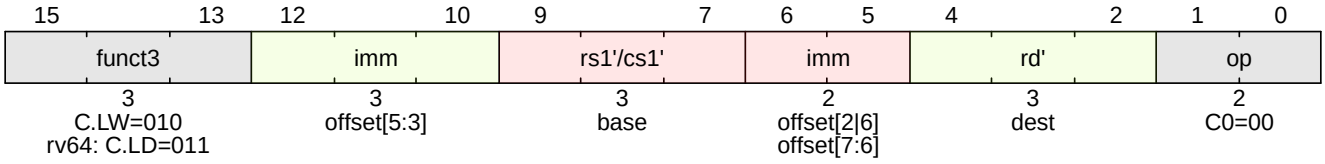
Integer Pointer Mode Mnemonic (RV32)

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

Integer Pointer Mode Expansion (RV32)

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

Encoding



Capability Pointer Mode Description

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

Integer Pointer Mode Description

Standard load instructions, authorised by the capability in ddc.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* C.LD

RV64, and C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode* C.LD

RV64, C or Zca, Zcherihybrid

Prerequisites *Capability Pointer Mode* C.LW

C or Zca, Zcheripurecap

Prerequisites *Integer Pointer Mode* C.LW

C or Zca, Zcherihybrid

Operation (after expansion to 32-bit encodings)

See LD, LW

12.6.13. C.LWSP

See C.LDSP.

12.6.14. C.LDSP

Synopsis

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

Capability Pointer Mode Mnemonics (RV64)

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

Capability Pointer Mode Expansions (RV64)

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

Integer Pointer Mode Mnemonics (RV64)

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

Integer Pointer Mode Expansions (RV64)

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

Capability Pointer Mode Mnemonic (RV32)

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

Capability Pointer Mode Expansion (RV32)

```
lw rd, offset(csp)
```

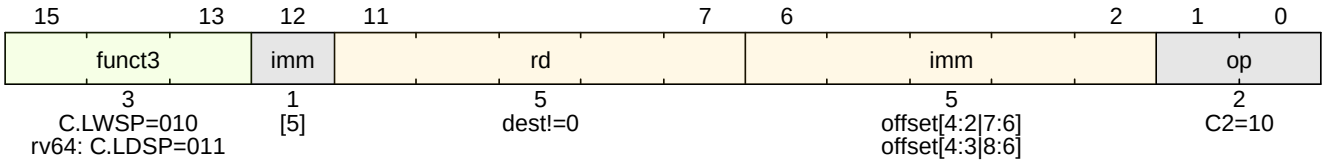
Integer Pointer Mode Mnemonic (RV32)

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

Integer Pointer Mode Expansion (RV32)

```
lw rd, offset(sp)
```

Encoding



Capability Pointer Mode Description

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

Integer Pointer Mode Description

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

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed

CAUSE	Reason
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* C.LDSP

RV64, and C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode* C.LDSP

RV64, and C or Zca, Zcherihybrid

Prerequisites for *Capability Pointer Mode* C.LWSP

C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode* C.LWSP

C or Zca, Zcherihybrid

Operation (after expansion to 32-bit encodings)

See LW, LD

12.6.15. C.FLW

See C.FLWSP.

12.6.16. C.FLWSP

Synopsis

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

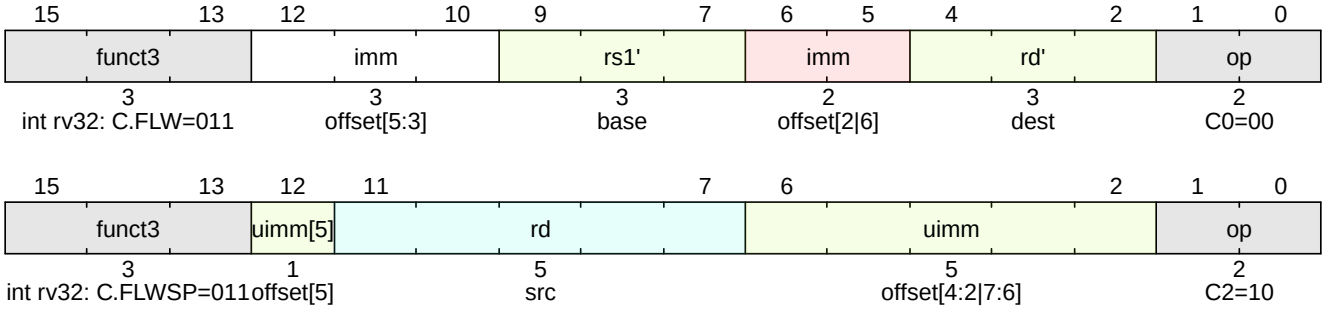
Integer Pointer Mode Mnemonics (RV32)

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

Integer Pointer Mode Expansions (RV32)

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

Encoding (RV32)



Integer Pointer Mode Description

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

These instructions are available in RV32 Integer Pointer Mode only. In Capability Pointer Mode they are remapped to C.LC/C.LCSP.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, and Zcf or F

Operation (after expansion to 32-bit encodings)

See FLW

12.6.17. C.FLD

See C.FLDSP

12.6.18. C.FLDSP

Synopsis

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

Capability Pointer Mode Mnemonic (RV32)

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

Capability Pointer Mode Expansion (RV32)

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

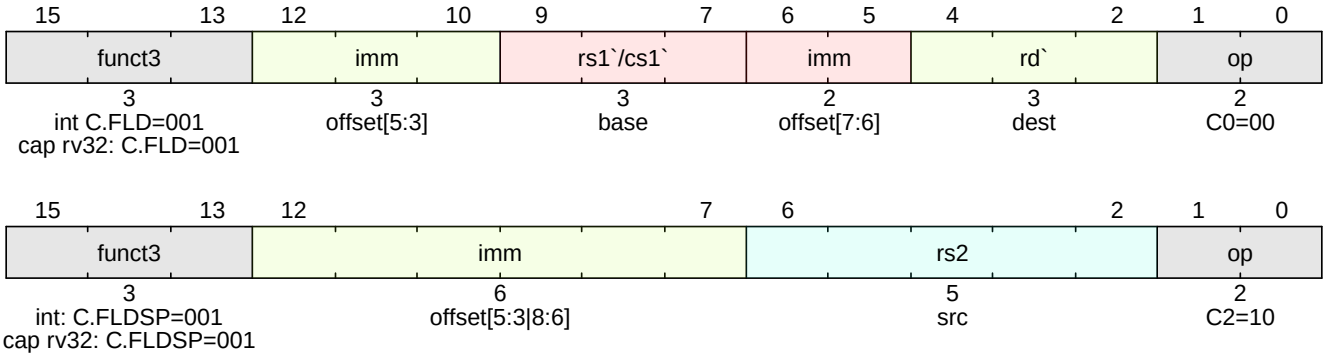
Integer Pointer Mode Mnemonic

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

Integer Pointer Mode Expansion

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

Encoding



Integer Pointer Mode Description

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

These instructions are available in RV64 Integer Pointer Mode only. In RV64 Capability Pointer Mode they are remapped to C.LC/C.LCSP.

These encodings may be remapped by future code-size Zcm standard extensions, similar to Zcmp and Zcmt. The rule is that in RV64 Capability Pointer Mode they are **always** *remapped to C.SC/C.SCSP.*

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set

CAUSE	Reason
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* (RV32 only)

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

Prerequisites for *Integer Pointer Mode*

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

Operation (after expansion to 32-bit encodings)

See FLD

12.6.19. C.LC

see C.LCSP.

12.6.20. C.LCSP

Synopsis

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

These instructions have different encodings for RV64 and RV32.

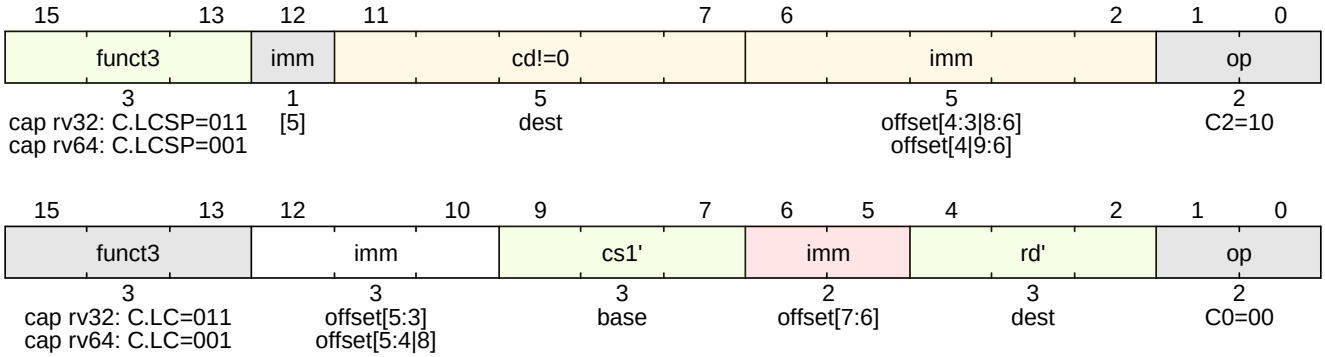
Capability Pointer Mode Mnemonics

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

Capability Pointer Mode Expansions

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

Encoding



Capability Pointer Mode Description

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

These mnemonics do not exist in Integer Pointer Mode.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion

CAUSE	Reason
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites

C or Zca, Zcheripurecap

Operation (after expansion to 32-bit encodings)

See LC

12.6.21. C.SD

See C.SW.

12.6.22. C.SW

Synopsis

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

Capability Pointer Mode Mnemonics (RV64)

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

Capability Pointer Mode Expansions (RV64)

```
sd rs2', offset(cs1')
sw rs2', offset(cs1')
```

Integer Pointer Mode Mnemonics (RV64)

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

Integer Pointer Mode Expansions (RV64)

```
sd rs2', offset(rs1')
sw rs2', offset(rs1')
```

Capability Pointer Mode Mnemonic (RV32)

```
c.sw rs2', offset(cs1')
```

Capability Pointer Mode Expansion (RV32)

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

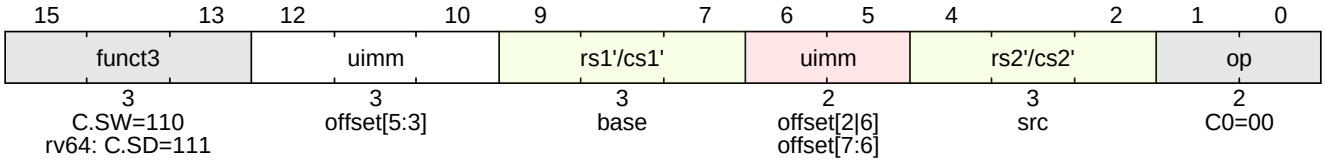
Integer Pointer Mode Mnemonic (RV32)

```
c.sw rs2', offset(rs1')
```

Integer Pointer Mode Expansion (RV32)

```
sw rs2', offset(rs1')
```

Encoding



Capability Pointer Mode Description

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

Integer Pointer Mode Description

Standard store instructions, authorised by the capability in ddc.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* C.SD

RV64, and C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode* C.SD

RV64, and C or Zca, Zcherihybrid

Prerequisites for *Capability Pointer Mode* C.SW

C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode* C.SW

C or Zca, Zcherihybrid

Operation (after expansion to 32-bit encodings)

See SD, SW

12.6.23. C.SWSP

See C.SDSP.

12.6.24. C.SDSP

Synopsis

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

Capability Pointer Mode Mnemonics (RV64)

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

Capability Pointer Mode Expansions (RV64)

```
sd rs2, offset(csp)
sw rs2, offset(csp)
```

Integer Pointer Mode Mnemonics (RV64)

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

Integer Pointer Mode Expansions (RV64)

```
sd rs2, offset(sp)
sw rs2, offset(sp)
```

Capability Pointer Mode Mnemonic (RV32)

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

Capability Pointer Mode Expansion (RV32)

```
sw rs2, offset(csp)
```

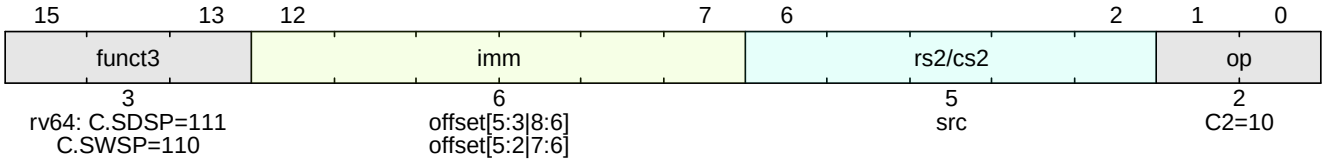
Integer Pointer Mode Mnemonic (RV32)

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

Integer Pointer Mode Expansion (RV32)

```
sw rs2, offset(sp)
```

Encoding



Capability Pointer Mode Description

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

Integer Pointer Mode Description

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

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* C.SDSP

RV64, and C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode* C.SDSP

RV64, and C or Zca, Zcherihybrid

Prerequisites for *Capability Pointer Mode* C.SWSP

C or Zca, Zcheripurecap

Prerequisites for *Integer Pointer Mode* C.SWSP

C or Zca, Zcherihybrid

Operation (after expansion to 32-bit encodings)

See SD, SW

12.6.25. C.FSW

See C.FSWSP.

12.6.26. C.FSWSP

Synopsis

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

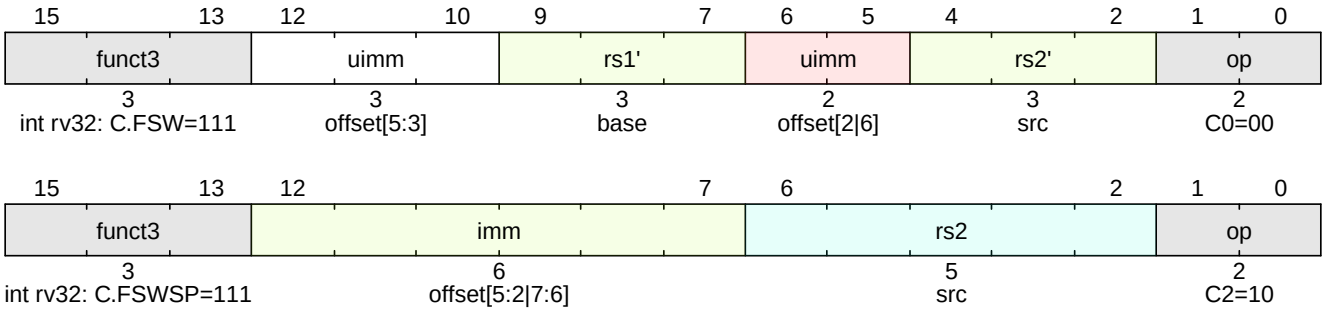
Integer Pointer Mode Mnemonics (RV32)

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

Integer Pointer Mode Expansions (RV32)

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

Encoding (RV32)



Integer Pointer Mode Description

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

These instructions are available in RV32 Integer Pointer Mode only. In Capability Pointer Mode they are remapped to C.SC/C.SCSP.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites

C or Zca, Zcherihybrid, Zcf or F

Operation (after expansion to 32-bit encodings)

See FSW

12.6.27. C.FSD

See C.FSDSP.

12.6.28. C.FSDSP

Synopsis

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

Capability Pointer Mode Mnemonics (RV32)

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

Capability Pointer Mode Expansions (RV32)

```
fsd fs2, offset(cs1')
fsd fs2, offset(csp)
```

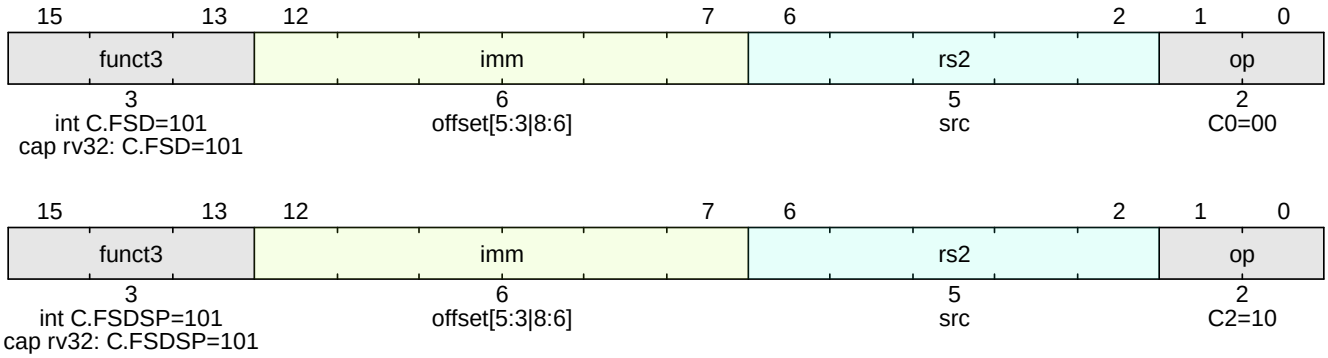
Integer Pointer Mode Mnemonics

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

Integer Pointer Mode Expansions

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

Encoding



Capability Pointer Mode Description

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

Integer Pointer Mode Description

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

These instructions are available in RV64 Integer Pointer Mode only. In RV64 Capability Pointer Mode they are remapped to C.SC/C.SCSP.

C.FSDSP may be remapped by the Zcmp, Zcmt standard extensions. C.FSD may be remapped by future code-size reduction extensions. The rule is that in RV64 Capability Pointer Mode they are **always** *remapped to C.LC/C.LCSP.*

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

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

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

Prerequisites for *Integer Pointer Mode* C.FSD, C.FSDSP

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

Operation (after expansion to 32-bit encodings)

See FSD

12.6.29. C.SC

see C.SCSP.

12.6.30. C.SCSP

Synopsis

Capability stores (C.SC, C.SCSP), 16-bit encodings

These instructions have different encodings for RV64 and RV32.

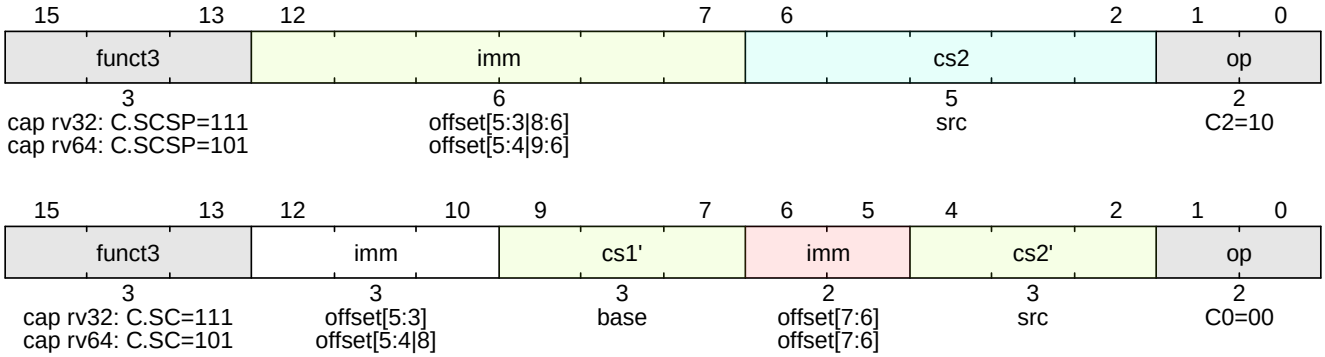
Capability Pointer Mode Mnemonics

```
c.sc cs2', offset(cs1')
c.sc cs2', offset(csp)
```

Capability Pointer Mode Expansions

```
sc cs2', offset(cs1')
sc cs2', offset(csp)
```

Encoding



Capability Pointer Mode Description

Store the CLEN+1 bit value in `cs2'` to memory. The capability in `cs1/csp` authorizes the operation. The effective address of the memory access is obtained by adding the address of `cs1/csp` to the sign-extended 12-bit offset.

These mnemonics do not exist in Integer Pointer Mode.

Tag of the written capability value

The capability written to memory has the tag set to 0 if the tag of `cs2'` is 0 or if the authorizing capability (`cs1/csp`) does not grant C-permission.

The stored tag is also set to zero if the authorizing capability does not have SL-permission set but the stored data has a Capability Level (CL) of 0 (*local*).

This instruction can propagate tagged capabilities which have malformed bounds, have reserved bits set or have a permission field which cannot be produced by ACPERM.

Exceptions

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

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below;

in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites

C or Zca, Zcheripurecap

Operation (after expansion to 32-bit encodings)

See SC

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

12.7.1. CBO.CLEAN

Synopsis

Perform a clean operation on a cache block

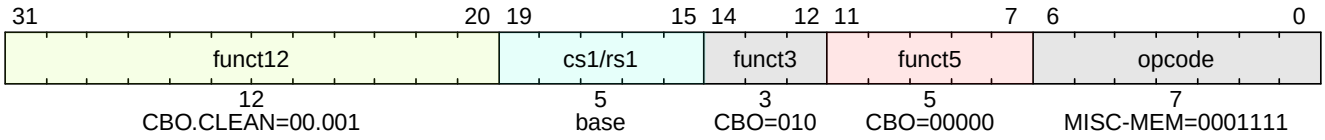
Capability Pointer Mode Mnemonic

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

Integer Pointer Mode Mnemonic

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

Encoding



Capability Pointer Mode Description

A CBO.CLEAN instruction performs a clean operation on the cache block whose effective address is the base address specified in `cs1`. The authorising capability for this operation is `cs1`.

Integer Pointer 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 occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission and R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	None of the bytes accessed are within the bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode*

Zicbom, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zicbom, Zcherihybrid

Operation

TBD

12.7.2. CBO.FLUSH

Synopsis

Perform a flush operation on a cache block

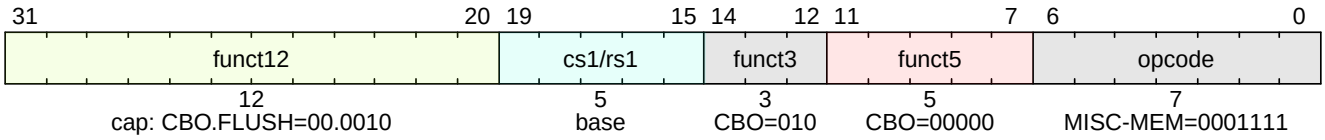
Capability Pointer Mode Mnemonic

```
cbo.flush 0(cs1)
```

Integer Pointer Mode Mnemonic

```
cbo.flush 0(rs1)
```

Encoding



Capability Pointer Mode Description

A CBO.FLUSH instruction performs a flush operation on the cache block whose effective address is the base address specified in `cs1`. The authorising capability for this operation is `cs1`.

Integer Pointer 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 occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission and R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	None of the bytes accessed are within the bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode*

Zicbom, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zicbom, Zcherihybrid

Operation

TBD

12.7.3. CBO.INVAL

Synopsis

Perform an invalidate operation on a cache block

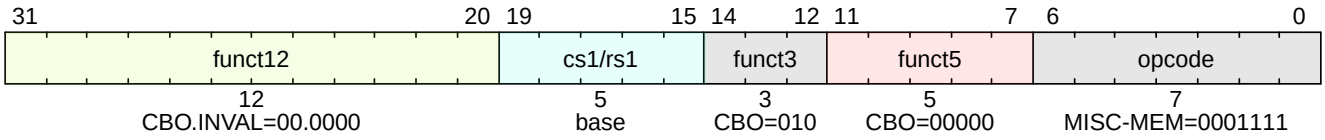
Capability Pointer Mode Mnemonic

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

Integer Pointer Mode Mnemonic

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

Encoding



Capability Pointer Mode Description

A CBO.INVAL 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`.

Integer Pointer 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 occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

The CBIE bit in menvcfg and senvcfg indicates whether CBO.INVAL performs cache block flushes instead of invalidations for less privileged modes. The instruction checks shown in the table below remain unchanged regardless of the value of CBIE and the privilege mode.

Invalidating a cache block can re-expose capabilities previously stored to it after the most recent flush, not just secret values. As such, CBO.INVAL has stricter checks on its use than CBO.FLUSH, and should only be made available to, and used by, sufficiently-trusted software. Untrusted software should use CBO.FLUSH instead.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, R-permission or ASR-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	None of the bytes accessed are within the bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode*

Zicbom, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zicbom, Zcherihybrid

Operation

```
TBD
```

12.7.4. CBO.ZERO

Synopsis

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

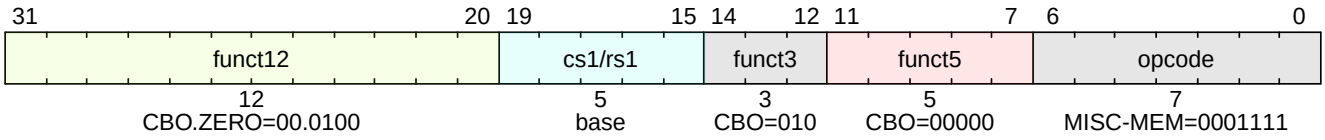
Capability Pointer Mode Mnemonic

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

Integer Pointer Mode Mnemonic

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

Encoding



Capability Pointer Mode Description

A `cbo.zero` instruction performs stores of zeros to the full set of bytes corresponding to the cache block whose effective address is the base address specified in `cs1`. An implementation may or may not update the entire set of bytes atomically although each individual write must atomically clear the tag bit of the corresponding aligned CLEN-bit location. The authorising capability for this operation is `cs1`.

Integer Pointer 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 exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode*

Zicboz, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zicboz, Zcherihybrid

Operation

```
TBD
```

12.7.5. PREFETCH.I

Synopsis

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

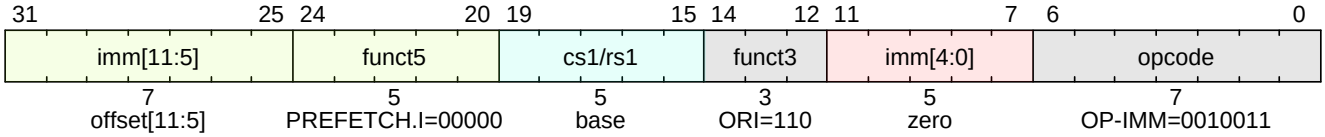
Capability Pointer Mode Mnemonic

```
prefetch.i offset(cs1)
```

Integer Pointer Mode Mnemonic

```
prefetch.i offset(rs1)
```

Encoding



Capability Pointer Mode Description

A PREFETCH.I instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in `cs1` and the sign-extended offset encoded in imm[11:0], where imm[4:0] equals 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`. This instruction does not throw any exceptions. However, following CHERI Exceptions and speculative execution, this instruction does not perform a prefetch if it is not authorized by `cs1`.

Integer Pointer 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 pcc.

In either mode, PREFETCH.I does not perform a memory access if one or more of the following conditions of the authorising capability are met:

- The tag is not set
- The sealed bit is set
- No bytes of the cache line requested is in bounds
- The X-permission is not set
- Any reserved bits are set
- The permissions could not have been produced by ACPERM
- The bounds are malformed

Prerequisites for *Capability Pointer Mode*

Zicbop, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zicbop, Zcherihybrid

Operation

```
TODO
```

12.7.6. PREFETCH.R

Synopsis

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

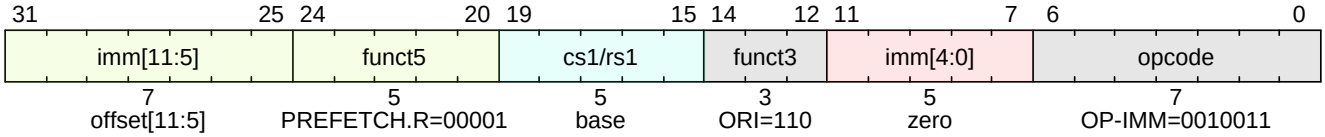
Capability Pointer Mode Mnemonic

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

Integer Pointer Mode Mnemonic

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

Encoding



Capability Pointer Mode Description

A PREFETCH.R instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in `cs1` and the sign-extended offset encoded in imm[11:0], where imm[4:0] equals 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`. This instruction does not throw any exceptions. However, in following CHERI Exceptions and speculative execution, this instruction does not perform a prefetch if it is not authorized by `cs1`.

Integer Pointer 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.

In either mode, PREFETCH.R does not perform a memory access if one or more of the following conditions of the authorising capability are met:

- The tag is not set
- The sealed bit is set
- No bytes of the cache line requested is in bounds
- The R-permission is not set
- Any reserved bits are set
- The permissions could not have been produced by ACPERM
- The bounds are malformed

Prerequisites for *Capability Pointer Mode*

Zicbop, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zicbop, Zcherihybrid

Operation

```
TODO
```

12.7.7. PREFETCH.W

Synopsis

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

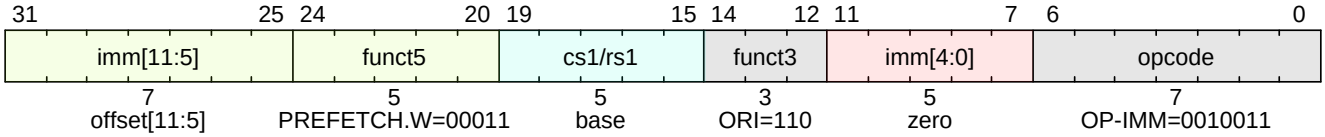
Capability Pointer Mode Mnemonic

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

Integer Pointer Mode Mnemonic

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

Encoding



Capability Pointer Mode Description

A PREFETCH.W instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in `cs1` and the sign-extended offset encoded in imm[11:0], where imm[4:0] equals 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`. This instruction does not throw any exceptions. However, following CHERI Exceptions and speculative execution, this instruction does not perform a prefetch if it is not authorized by `cs1`.

Integer Pointer 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.

In either mode, PREFETCH.W does not perform a memory access if one or more of the following conditions of the authorising capability are met:

- The tag is not set
- The sealed bit is set
- No bytes of the cache line requested is in bounds
- The W-permission is not set
- Any reserved bits are set
- The permissions could not have been produced by ACPERM
- The bounds are malformed

Prerequisites for *Capability Pointer Mode*

Zicbop, Zcheripurecap

Prerequisites for *Integer Pointer Mode*

Zicbop, Zcherihybrid

Operation

```
TODO
```

12.8. "Zba" Extension for Bit Manipulation Instructions

12.8.1. ADD.UW

Synopsis

Add unsigned word for address generation

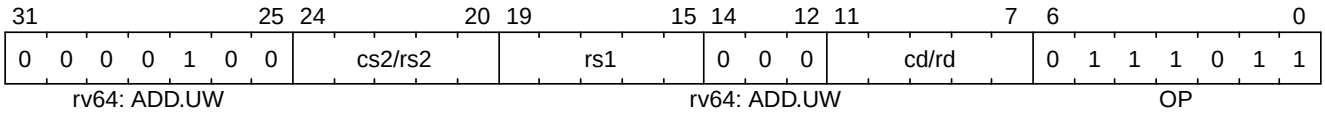
Capability Pointer Mode Mnemonic (RV64)

```
add.uw cd, rs1, cs2
```

Integer Pointer Mode Mnemonic (RV64)

```
add.uw rd, rs1, rs2
```

Encoding



Capability Pointer Mode Description

Increment the address field of `cs2` by the unsigned word in `rs1`. Clear the tag if the resulting capability is unrepresentable or `cs2` is sealed.

Integer Pointer Mode Description

Increment `rs2` by the unsigned word in `rs1`.

Prerequisites for *Capability Pointer Mode*

RV64, Zcheripurecap, Zba

Prerequisites for *Integer Pointer Mode*

RV64, Zcherihybrid, Zba

Capability Pointer Mode Operation

```
TBD
```

Integer Pointer Mode Operation

TODO

12.8.2. SH1ADD

See SH3ADD.

12.8.3. SH2ADD

See SH3ADD.

12.8.4. SH3ADD

Synopsis

Shift by *n* and add for address generation (SH1ADD, SH2ADD, SH3ADD)

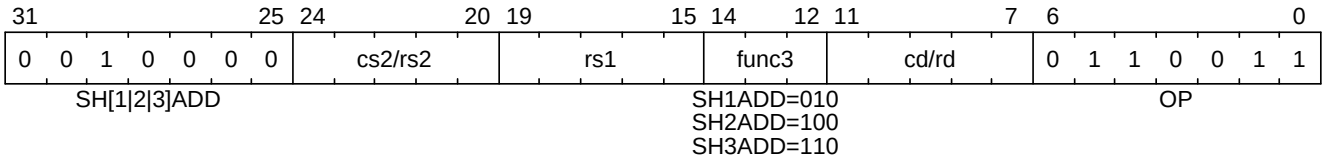
Capability Pointer Mode Mnemonics

```
sh[1|2|3]add cd, rs1, cs2
```

Integer Pointer Mode Mnemonics

```
sh[1|2|3]add rd, rs1, rs2
```

Encoding



Capability Pointer Mode Description

Increment the address field of `cs2` by `rs1` shifted left by *n* bit positions and write the result to `cd`. The tag bit of the output capability is 0 if `cs2` did not have its tag set to 1, the incremented address is outside `cs2` 's Representable Range or `cs2` is sealed.

This instruction sets `cd.tag=0` *if* `cs2` *'s bounds are malformed, or if any of the reserved fields are set.*

Integer Pointer Mode Description

Increment `rs2` by `rs1` shifted left by *n* bit positions and write the result to `rd`.

Exceptions

None

Prerequisites for *Capability Pointer Mode*

Zcheripurecap, Zba

Prerequisites for *Integer Pointer Mode*

Zcherihybrid, Zba

Capability Pointer Mode Operation

```
TODO
```

Integer Pointer Mode Operation

TODO

12.8.5. SH1ADD.UW

See SH3ADD.UW.

12.8.6. SH2ADD.UW

See SH3ADD.UW.

12.8.7. SH3ADD.UW

Synopsis

Shift by *n* and add unsigned word for address generation (SH1ADD.UW, SH2ADD.UW, SH3ADD.UW)

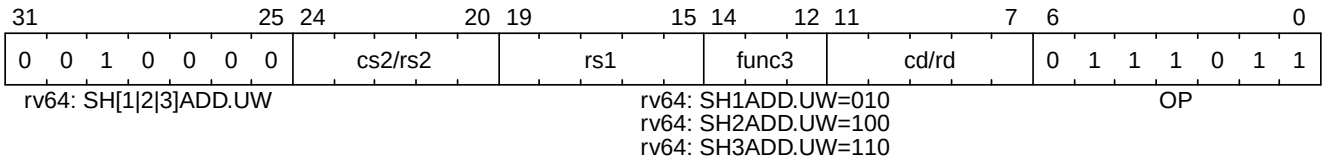
Capability Pointer Mode Mnemonics (RV64)

```
sh[1|2|3]add.uw cd, rs1, cs2
```

Integer Pointer Mode Mnemonics (RV64)

```
sh[1|2|3]add.uw rd, rs1, rs2
```

Encoding



Capability Pointer Mode Description

Increment the address field of `cs2` by the unsigned word in `rs1` shifted left by *n* bit positions and write the result to `cd`. The tag bit of the output capability is 0 if `cs2` did not have its tag set to 1, the incremented address is outside `cs2` 's Representable Range or `cs2` is sealed.

This instruction sets `cd.tag=0` *if* `cs2` *'s bounds are malformed, or if any of the reserved fields are set.*

Integer Pointer Mode Description

Increment `rs2` by the unsigned word in `rs1` shifted left by *n* bit positions and write the result to `rd`.

Exceptions

None

Prerequisites for *Capability Pointer Mode*

RV64, Zcheripurecap, Zba

Prerequisites for *Integer Pointer Mode*

RV64, Zcherihybrid, Zba

Capability Pointer Mode Operation

```
TODO
```

Integer Pointer Mode Operation

TODO

12.8.8. SH4ADD

Synopsis

Shift by 4 and add for address generation (SH4ADD)

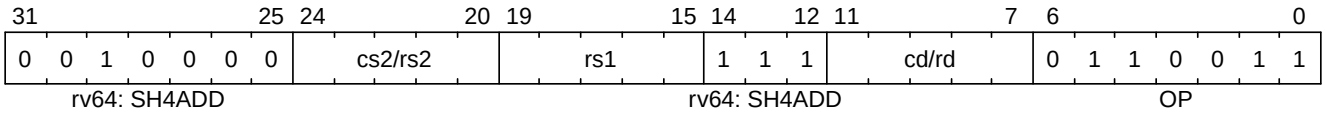
Capability Pointer Mode Mnemonic (RV64)

```
sh4add cd, rs1, cs2
```

Integer Pointer Mode Mnemonic (RV64)

```
sh4add rd, rs1, rs2
```

Encoding



Capability Pointer Mode Description

Increment the address field of `cs2` by `rs1` shifted left by 4 bit positions and write the result to `cd`. The tag bit of the output capability is 0 if `cs2` did not have its tag set to 1, the incremented address is outside `cs2`'s Representable Range or `cs2` is sealed.

This instruction sets `cd.tag=0` if `cs2`'s bounds are malformed, or if any of the reserved fields are set.

Integer Pointer Mode Description

Increment `rs2` by `rs1` shifted left by 4 bit positions and write the result to `rd`.

Exceptions

None

Prerequisites for *Capability Pointer Mode*

RV64, Zish4add

Prerequisites for *Integer Pointer Mode*

RV64, Zish4add

Capability Pointer Mode Operation

```
TBD
```

Integer Pointer Mode Operation

TBD

12.8.9. SH4ADD.UW

Synopsis

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

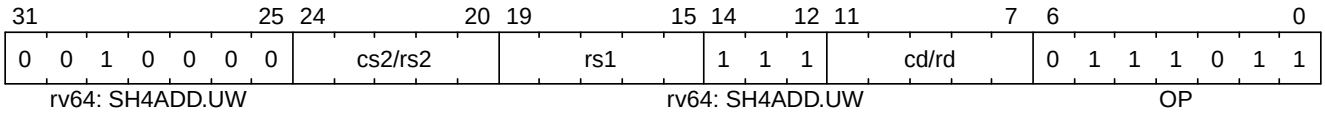
Capability Pointer Mode Mnemonic (RV64)

```
sh4add.uw cd, rs1, cs2
```

Integer Pointer Mode Mnemonic (RV64)

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

Encoding



Capability Pointer Mode Description

Increment the address field of `cs2` by the unsigned word in `rs1` shifted left by 4 bit positions and write the result to `cd`. The tag bit of the output capability is 0 if `cs2` did not have its tag set to 1, the incremented address is outside `cs2` 's Representable Range or `cs2` is sealed.

This instruction sets `cd.tag=0` if `cs2` 's bounds are malformed, or if any of the reserved fields are set.

Integer Pointer Mode Description

Increment `rs2` by the unsigned word in `rs1` shifted left by 4 bit positions and write the result to `rd`.

Exceptions

None

Prerequisites for *Capability Pointer Mode*

RV64, Zish4add

Prerequisites for *Integer Pointer Mode*

RV64, Zish4add

Capability Pointer Mode Operation

```
TBD
```

Integer Pointer Mode Operation

TBD

12.9. "Zcb" Standard Extension For Code-Size Reduction

12.9.1. C.LH

See C.LBU.

12.9.2. C.LHU

See C.LBU.

12.9.3. C.LBU

Synopsis

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

Capability Pointer Mode Mnemonics

```
c.lh rd', offset(cs1')
c.lhu rd', offset(cs1')
c.lbu rd', offset(cs1')
```

Capability Pointer Mode Expansions

```
lh rd, offset(cs1)
lhu rd, offset(cs1)
lbu rd, offset(cs1)
```

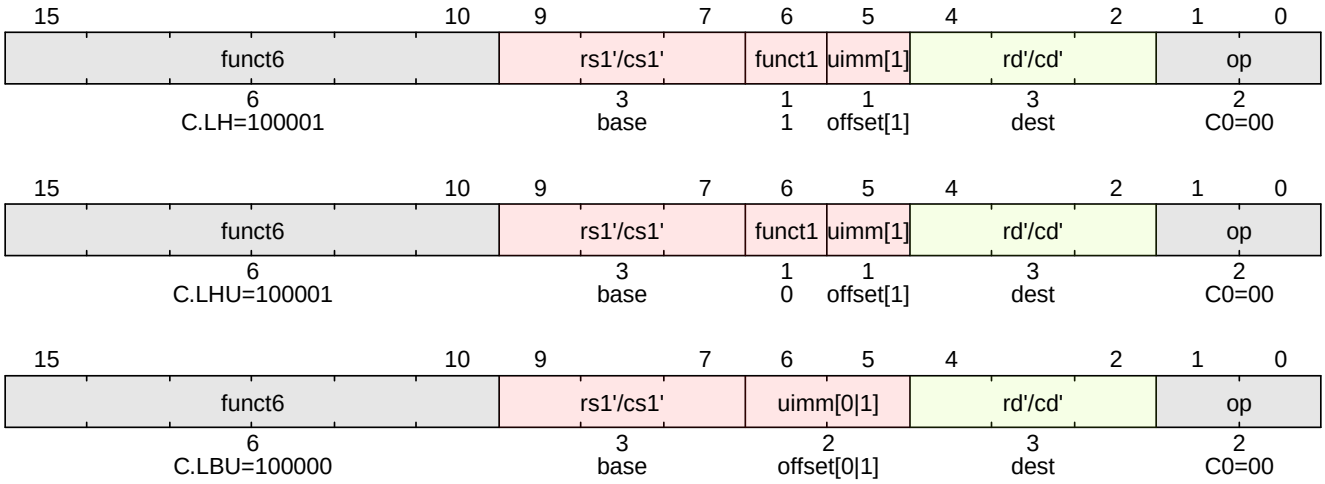
Integer Pointer Mode Mnemonics

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

Integer Pointer Mode Expansions

```
lh rd, offset(rs1)
lhu rd, offset(rs1)
lbu rd, offset(rs1)
```

Encoding



Capability Pointer Mode Description

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

Integer Pointer Mode Description

Subword load instructions, authorised by the capability in ddc.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap, and Zcb

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, and Zcb

Operation (after expansion to 32-bit encodings)

See LHU, LH, LBU

12.9.4. C.SH

See C.SB.

12.9.5. C.SB

Synopsis

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

Capability Pointer Mode Mnemonics

```
c.sh rs2', offset(cs1')
c.sb rs2', offset(cs1')
```

Capability Pointer Mode Expansions

```
sh rs2', offset(cs1')
sb rs2', offset(cs1')
```

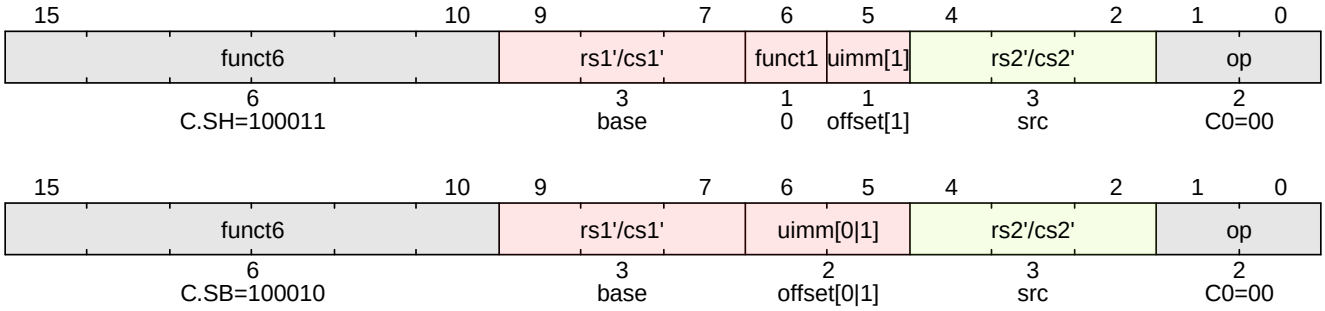
Integer Pointer Mode Mnemonics

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

Integer Pointer Mode Expansions

```
sh rs2', offset(rs1')
sb rs2', offset(rs1')
```

Encoding



Capability Pointer Mode Description

Subword store instructions, authorised by the capability in `cs1`.

Integer Pointer Mode Description

Subword store instructions, authorised by the capability in ddc.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion

CAUSE	Reason
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap, and Zcb

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, and Zcb

Operation (after expansion to 32-bit encodings)

See SH, SB

12.10. "Zcmp" Standard Extension For Code-Size Reduction

The push (CM.PUSH) and pop (CM.POP, CM.POPRET, CM.POPRETZ) instructions are redefined in *Capability Pointer Mode* to save/restore full capabilities.

The double move instructions (CM.MVSA01, CM.MVA01S) are redefined in *Capability Pointer Mode* to move full capabilities between registers. The saved register mapping is as shown in

Table 42. saved register mapping for Zcmp

saved register specifier	xreg	integer ABI	CHERI ABI
0	x8	s0	cs0
1	x9	s1	cs1
2	x18	s2	cs2
3	x19	s3	cs3
4	x20	s4	cs4
5	x21	s5	cs5
6	x22	s6	cs6
7	x23	s7	cs7

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

12.10.1. CM.PUSH

Synopsis

Create stack frame (CM.PUSH): store the return address register and 0 to 12 saved registers to the stack frame, optionally allocate additional stack space. 16-bit encodings.

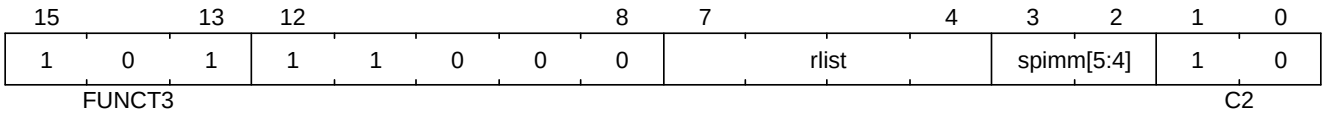
Capability Pointer Mode Mnemonic (RV32)

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

Integer Pointer Mode Mnemonic

```
cm.push {reg_list}, -stack_adj
```

Encoding



rlist values 0 to 3 are reserved for a future EABI variant

Capability Pointer 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`.

Integer Pointer 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.

This encoding conflicts with C.FSDSP which is remapped to C.SCSP in RV64 Capability Pointer Mode.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap, Zcmp

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, Zcmp

Operation

```
TBD
```

12.10.2. CM.POP

Synopsis

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

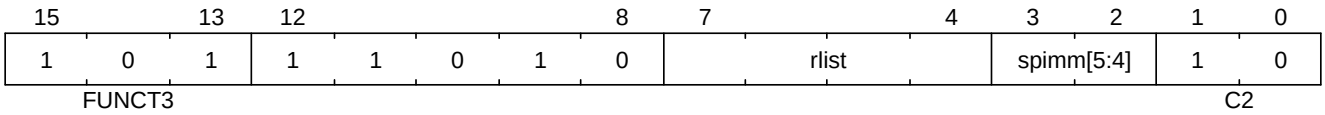
Capability Pointer Mode Mnemonic (RV32)

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

Integer Pointer Mode Mnemonic

```
cm.pop {reg_list}, -stack_adj
```

Encoding



rlist values 0 to 3 are reserved for a future EABI variant

Capability Pointer Mode Description

Load capability registers as specified in *creg_list*. Deallocate stack frame. All accesses are checked against `csp`.

Integer Pointer Mode Description

Load integer registers as specified in *reg_list*. Deallocate stack frame. All accesses are checked against ddc.

This encoding conflicts with C.FSDSP which is remapped to C.SCSP in RV64 Capability Pointer Mode.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap, Zcmp

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, Zcmp

Operation

```
TBD
```

12.10.3. CM.POPRET

Synopsis

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

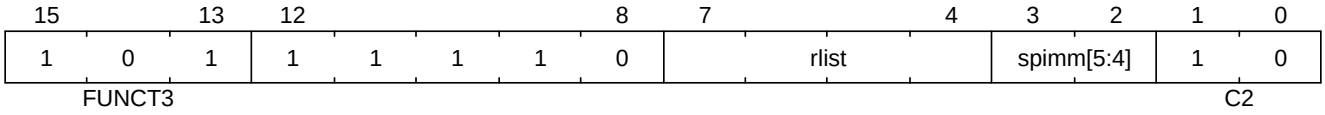
Capability Pointer Mode Mnemonic (RV32)

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

Integer Pointer Mode Mnemonic

```
cm.popret {reg_list}, -stack_adj
```

Encoding



rlist values 0 to 3 are reserved for a future EABI variant

Capability Pointer Mode Description

Load capability registers as specified in *creg_list*. Deallocate stack frame. Return by calling JALR to `cra`. All data accesses are checked against `csp`. The return destination is checked against `cra`.

Integer Pointer 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.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

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

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap, Zcmp

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, Zcmp

Operation

```
TBD
```

12.10.4. CM.POPRETZ

Synopsis

Destroy stack frame (CM.POPRETZ): load the return address register and register 0 to 12 saved registers from the stack frame, deallocate the stack frame. Move zero into argument register zero. Return through the return address register. 16-bit encodings.

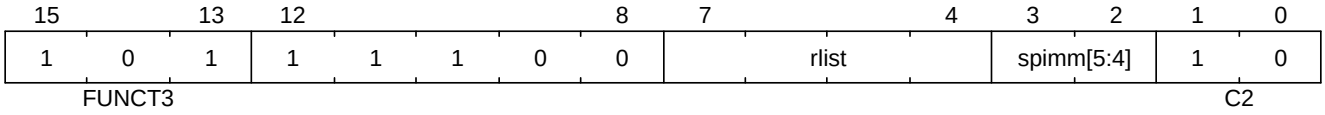
Capability Pointer Mode Mnemonic (RV32)

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

Integer Pointer Mode Mnemonic

```
cm.popretz {reg_list}, -stack_adj
```

Encoding



rlist values 0 to 3 are reserved for a future EABI variant

Capability Pointer Mode Description

Load capability registers as specified in *creg_list*. Deallocate stack frame. Move zero into `ca0`. Return by calling JALR to `cra`. All data accesses are checked against `csp`. The return destination is checked against `cra`.

Integer Pointer Mode Description

Load integer registers as specified in *reg_list*. Deallocate stack frame. Move zero into `a0`. Return by calling JALR to `ra`. All data accesses are checked against ddc. The return destination is checked against pcc.

Permissions

Loads are checked as for LC in both *Integer Pointer Mode* and *Capability Pointer Mode*.

The return is checked as for JALR in both *Integer Pointer Mode* and *Capability Pointer Mode*.

This encoding conflicts with C.FSDSP which is remapped to C.SCSP in RV64 Capability Pointer Mode.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM

CAUSE	Reason
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

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

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap, Zcmp

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, Zcmp

Operation

```
TBD
```

12.10.5. CM.MVSA01

Synopsis

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

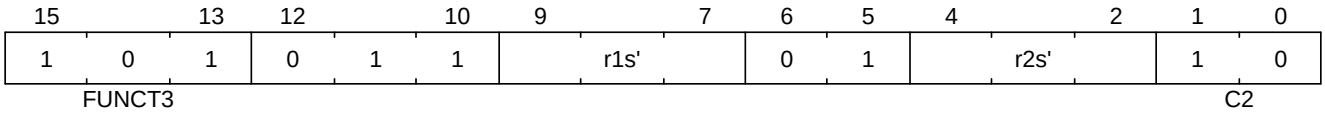
Capability Pointer Mode Mnemonic (RV32)

```
cm.mvsa01 cr1s', cr2s'
```

Integer Pointer Mode Mnemonic

```
cm.mvsa01 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 42.

Capability Pointer Mode Description

Atomically move two saved capability registers `cs0-cs7` into `ca0` and `ca1`.

Integer Pointer Mode Description

Atomically move two saved integer registers `s0-s7` into `a0` and `a1`.

This encoding conflicts with C.FSDSP which is remapped to C.SCSP in RV64 Capability Pointer Mode.

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap, Zcmp

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, Zcmp

Operation

```
TBD
```

12.10.6. CM.MVA01S

Synopsis

Move two saved registers into argument registers 0 and 1.

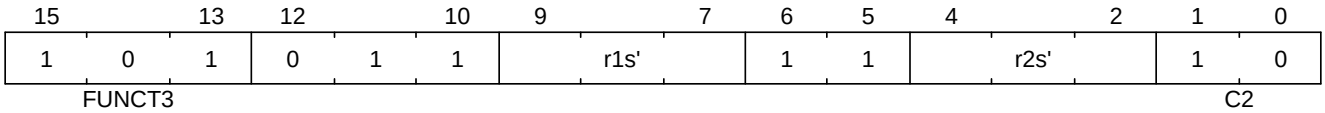
Capability Pointer Mode Mnemonic (RV32)

```
cm.mva01s cr1s', cr2s'
```

Integer Pointer Mode Mnemonic

```
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 42.

Capability Pointer Mode Description

Atomically move two capability registers `ca0` and `ca1` into `cs0-cs7`.

Integer Pointer Mode Description

Atomically move two integer registers `a0` and `a1` into `s0-s7`.

This encoding conflicts with C.FSDSP which is remapped to C.SCSP in RV64 Capability Pointer Mode.

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap, Zcmp

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, Zcmp

Operation

```
TBD
```

12.11. "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 Pointer Mode* to have capabilities in the jump table. This is to prevent the code-size growth caused by doubling the size of the jump table.

In the future, new jump table modes or new encodings can be added to have capabilities in the jump table.

The jump vector table CSR jvt has a capability alias jvtc so that it can only be configured to point to accessible memory. All accesses to the jump table are checked against jvtc in *Capability Pointer Mode*, and against pcc bounds in *Integer Pointer Mode*. This allows the jump table to be accessed when the pcc bounds are set narrowly to the local function only in *Capability Pointer Mode*.

Zcmt defines that the fetch from the jump table is from instruction memory. The overall instruction executed is effectively 48-bit, with 16-bits from CM.JALT/CM.JT, the other 32-bits (for RV32) from the table. Therefore pcc is used to authorise the fetch in Integer Pointer Mode, as the fetch is designated to be from instruction memory in (RISC-V, 2023).

In Capability Pointer Mode the implementation doesn't need to expand and bounds check against jvtc on every access, it is sufficient to decode the valid accessible range of entries after every write to jvtc, and then check that the accessed entry is in that range.

12.11.1. Jump Vector Table CSR (jvt)

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

12.11.2. Jump Vector Table CSR (jvtc)

jvtc extends jvt to be a capability width CSR, as shown in Table 18.

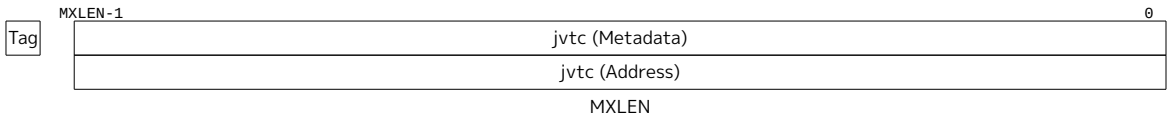


Figure 68. Jump Vector Table Capability register

All instruction fetches from the jump vector table are checked against jvtc in *Capability Pointer Mode*. In *Integer Pointer Mode* the address field gives the base address of the table, and the access is checked against pcc bounds.

See CM.JALT, CM.JT.

If the access to the jump table succeeds, then the instructions execute as follows:

- CM.JT executes as J or AUIPC+JR
- CM.JALT executes as JAL or AUIPC+JALR

As a result the capability metadata is retained in pcc during execution.

12.11.3. CM.JALT

Synopsis

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

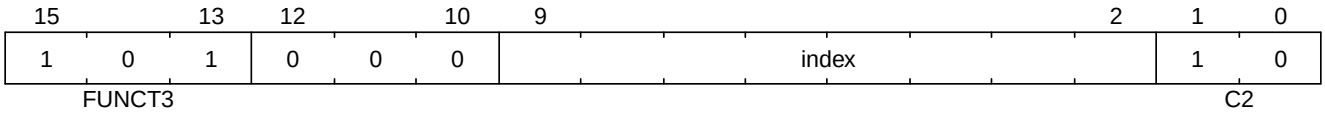
Capability Pointer Mode Mnemonic (RV32)

```
cm.jalt index
```

Integer Pointer Mode Mnemonic

```
cm.jalt index
```

Encoding



For this encoding to decode as CM.JALT, index≥32, otherwise it decodes as CM.JT.

Capability Pointer Mode Description

Redirect instruction fetch via the jump table defined by the indexing via `jvtc.address+index*XLEN/8`, checking every byte of the jump table access against jvtc bounds (not against pcc) and requiring X-permission. Link to `cra`.

Integer Pointer Mode Description

Redirect instruction fetch via the jump table defined by the indexing via `jvtc.address+index*XLEN/8`, checking every byte of the jump table access against pcc bounds and requiring X-permission. Link to `ra`.

This encoding conflicts with C.FSDSP which is remapped to C.SCSP in RV64 Capability Pointer Mode.

Capability Pointer Mode Permissions

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

Capability Pointer Mode Exceptions

When these instructions cause CHERI exceptions, *CHERI instruction fetch fault* is reported in the TYPE field and the following codes may be reported in the CAUSE field of mtval2 or stval2:

CAUSE	
Tag violation	✔
Seal violation	✔
Permission violation	✔
Invalid address violation	✔
Bounds violation	✔

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

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap, Zcmt

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, Zcmt

Operation

```
TBD
```

12.11.4. CM.JT

Synopsis

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

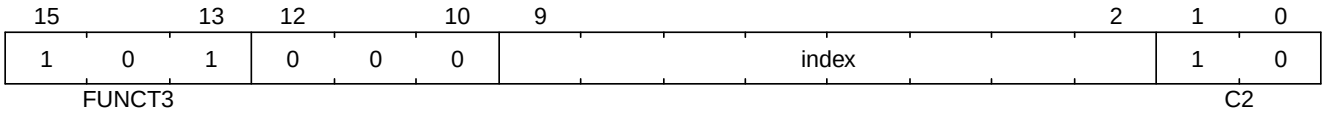
Capability Pointer Mode Mnemonic

`cm.jt index`

Integer Pointer Mode Mnemonic

`cm.jt index`

Encoding



For this encoding to decode as CM.JT, index<32, otherwise it decodes as CM.JALT.

Capability Pointer Mode Description

Redirect instruction fetch via the jump table defined by the indexing via `jvtc.address+index*XLEN/8`, checking every byte of the jump table access against jvtc bounds (not against pcc) and requiring X-permission.

Integer Pointer Mode Description

Redirect instruction fetch via the jump table defined by the indexing via `jvtc.address+index*XLEN/8`, checking every byte of the jump table access against pcc bounds and requiring X-permission.

This encoding conflicts with C.FSDSP which is remapped to C.SCSP in RV64 Capability Pointer Mode.

Capability Pointer Mode Permissions

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

Capability Pointer Mode Exceptions

When these instructions cause CHERI exceptions, *CHERI instruction fetch fault* is reported in the TYPE field and the following codes may be reported in the CAUSE field of mtval2 or stval2:

CAUSE	
Tag violation	✔
Seal violation	✔
Permission violation	✔
Invalid address violation	✔
Bounds violation	✔

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

Prerequisites for *Capability Pointer Mode*

C or Zca, Zcheripurecap, Zcmt

Prerequisites for *Integer Pointer Mode*

C or Zca, Zcherihybrid, Zcmt

Operation

```
TBD
```

12.12. "H" Extension for Hypervisor Support

12.12.1. HLV.B

See HLV.W.

12.12.2. HLV.BU

See HLV.W.

12.12.3. HLV.H

See HLV.W.

12.12.4. HLV.HU

See HLV.W.

12.12.5. HLV.WU

See HLV.W.

12.12.6. HLV.D

See HLV.W.

12.12.7. HLV.W

Synopsis

Hypervisor virtual-machine load

Capability Pointer Mode Mnemonics (RV64)

```
hlv.b[u] rd, cs1
hlv.h[u] rd, cs1
hlv.w[u] rd, cs1
hlv.d rd, cs1
```

Integer Pointer Mode Mnemonics (RV64)

```
hlv.b[u] rd, rs1
hlv.h[u] rd, rs1
hlv.w[u] rd, rs1
hlv.d rd, rs1
```

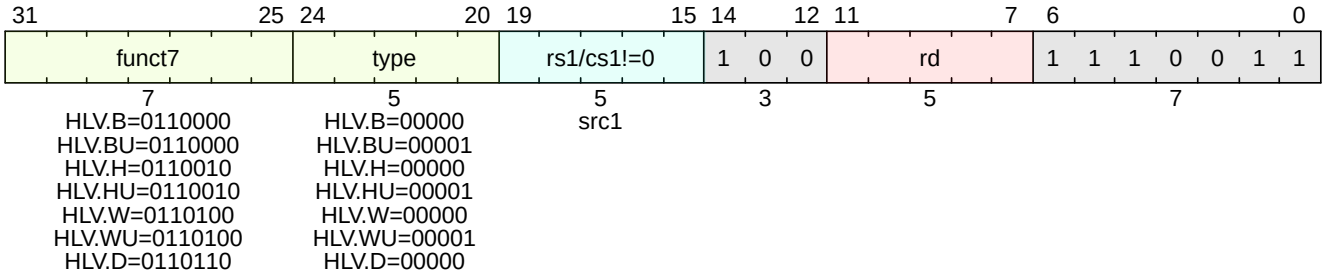
Capability Pointer Mode Mnemonics (RV32)

```
hlv.b[u] rd, cs1
hlv.h[u] rd, cs1
hlv.w rd, cs1
```

Integer Pointer Mode Mnemonics (RV32)

```
hlv.b[u] rd, rs1
hlv.h[u] rd, rs1
hlv.w rd, rs1
```

Encoding



Capability Pointer Mode Description

Performs a load as though V=1; i.e., with the address translation and protection, and endianness, that apply to memory accesses in either VS-mode or VU-mode. The effective address is the address of `cs1`. The authorising capability for the operation is `cs1`. A copy of the loaded value is written to `rd`.

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer Mode Description

Performs a load as though V=1; i.e., with the address translation and protection, and endianness, that apply to memory accesses in either VS-mode or VU-mode. The effective address is the is `rs1`. The authorising capability for the operation is ddc. A copy of the loaded value is written to `rd`.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* HLV.B[U], HLV.H[U], HLV.W

Zcheripurecap, H

Prerequisites for *Integer Pointer Mode* HLV.B[U], HLV.H[U], HLV.W

Zcheripurecap, Zcherihybrid, H

Prerequisites for *Capability Pointer Mode* HLV.WU, HLV.D

RV64, Zcheripurecap, H

Prerequisites for *Integer Pointer Mode* HLV.WU, HLV.D

RV64, Zcheripurecap, Zcherihybrid, H

Capability Pointer Mode Operation

TBD

Integer Pointer Mode Operation

TBD

12.12.8. HLV.C

Synopsis

Hypervisor virtual-machine load capability

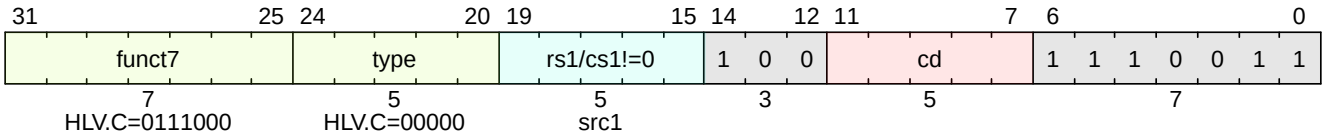
Capability Pointer Mode Mnemonic

```
hlv.c cd, cs1
```

Integer Pointer Mode Mnemonic

```
hlv.c cd, rs1
```

Encoding



Capability Pointer Mode Description

Load a CLEN+1 bit value from memory as though V=1; i.e., with the address translation and protection, and endianness, that apply to memory accesses in either VS-mode or VU-mode. The effective address is the address of `cs1`. The authorising capability for the operation is `cs1`. A copy of the loaded value is written to `cd`.

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer Mode Description

Load a CLEN+1 bit value from memory as though V=1; i.e., with the address translation and protection, and endianness, that apply to memory accesses in either VS-mode and VU-mode. The effective address is `rs1`. The authorising capability for the operation is ddc. A copy of the loaded value is written to `cd`.

Resulting value of **`cd`**

The tag value written to `cd` is 0 if the tag of the memory location loaded is 0 or the authorizing capability (ddc or `cs1`) does not grant C-permission.

If the authorizing capability does not grant LM-permission, and the tag of `cd` is 1 and `cd` is not sealed, then an implicit ACPERM clearing W-permission and LM-permission is performed to obtain the intermediate permissions on `cd`.

If the authorizing capability does not grant EL-permission, and the tag of `cd` is 1, then an implicit ACPERM clearing EL-permission and restricting the Capability Level (CL) to the level of the authorizing capability is performed to obtain the final permissions on `cd`.

If the authorizing capability does not grant EL-permission, and the tag of `cd` is 1, then an implicit ACPERM restricting the Capability Level (CL) to the level of the authorizing capability is performed. If `cd` is not sealed, this implicit ACPERM also clears EL-permission to obtain the final permissions on `cd` (see Table 30).

Missing LM-permission does not affect untagged values since this could result in surprising bit patterns when copying non-capability data. Similarly, sealed capabilities are

not modified as they are not directly dereferenceable.

Missing EL-permission also affects the level of sealed capabilities since notionally the Capability Level (CL) of a capability is not a permission but rather a data flow label attached to the loaded value. However, untagged values are not affected by EL-permission.

While the implicit ACPERM introduces a dependency on the loaded data, microarchitectures can avoid this by deferring the actual masking of permissions until the loaded capability is dereferenced or the metadata bits are inspected using GCPERM or GCHI.

This instruction can propagate tagged capabilities which have malformed bounds, have reserved bits set or have a permission field which cannot be produced by ACPERM.

Exceptions

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

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites for *Capability Pointer Mode*

Zcheripurecap, H

Prerequisites for *Integer Pointer Mode*

Zcheripurecap, Zcherihybrid, H

Capability Pointer Mode Operation

TBD

Integer Pointer Mode Operation

TBD

12.12.9. HSV.B

See HSV.W.

12.12.10. HSV.H

See HSV.W.

12.12.11. HSV.D

See HSV.W.

12.12.12. HSV.W

Synopsis

Hypervisor virtual-machine store

Capability Pointer Mode Mnemonics (RV64)

```
hsv.b rs2, cs1
hsv.h rs2, cs1
hsv.w rs2, cs1
hsv.d rs2, cs1
```

Integer Pointer Mode Mnemonics (RV64)

```
hsv.b rs2, rs1
hsv.h rs2, rs1
hsv.w rs2, rs1
hsv.d rs2, rs1
```

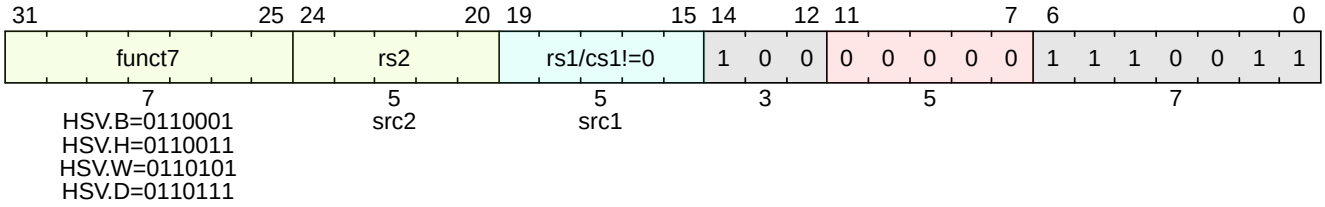
Capability Pointer Mode Mnemonics (RV32)

```
hsv.b rs2, cs1
hsv.h rs2, cs1
hsv.w rs2, cs1
```

Integer Pointer Mode Mnemonics (RV32)

```
hsv.b rs2, rs1
hsv.h rs2, rs1
hsv.w rs2, rs1
```

Encoding



Capability Pointer Mode Description

Performs a store as though V=1; i.e., with the address translation and protection, and endianness, that apply to memory accesses in either VS-mode or VU-mode. The effective address is 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.

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer Mode Description

Performs a store as though V=1; i.e., with address translation and protection, and endianness, that apply to memory accesses in either VS-mode or VU-mode. The effective address is `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 exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode* HSV.B, HSV.H, HSV.W

Zcheripurecap, H

Prerequisites for *Integer Pointer Mode* HSV.B, HSV.H, HSV.W

Zcheripurecap, Zcherihybrid H

Prerequisites for *Capability Pointer Mode* HSV.D

RV64, Zcheripurecap, H

Prerequisites for *Integer Pointer Mode* HSV.D

RV64, Zcheripurecap, Zcherihybrid H

Capability Pointer Mode Operation

TBD

Integer Pointer Mode Operation

TBD

12.12.13. HSV.C

Synopsis

Hypervisor virtual-machine store capability

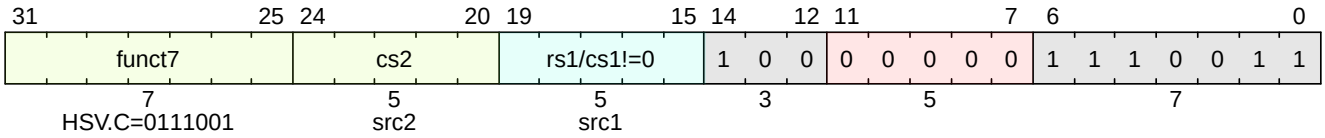
Capability Pointer Mode Mnemonic

```
hsv.c cs2, cs1
```

Integer Pointer Mode Mnemonic

```
hsv.c cs2, rs1
```

Encoding



Capability Pointer Mode Description

Store a CLEN+1 bit value in `cs2` to memory as though V=1; i.e., with the address translation and protection, and endianness, that apply to memory accesses in either VS-mode or VU-mode. The effective address is the address of `cs1`. The authorising capability for the operation is `cs1`.

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer Mode Description

Store a CLEN+1 bit value in `cs2` to memory as though V=1; i.e., with the address translation and protection, and endianness, that apply to memory accesses in either VS-mode or VU-mode. The effective address is the `rs1`. The authorising capability for the operation is ddc.

Tag of the written capability value

The capability written to memory has the tag set to 0 if the tag of `cs2` is 0 or if the authorizing capability (ddc or `cs1`) does not grant C-permission.

The stored tag is also set to zero if the authorizing capability does not have SL-permission set but the stored data has a Capability Level (CL) of 0 (*local*).

This instruction can propagate tagged capabilities which have malformed bounds, have reserved bits set or have a permission field which cannot be produced by ACPERM.

Exceptions

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

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set

CAUSE	Reason
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant W-permission, or the AP field could not have been produced by ACPERM
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

This instruction is illegal if the CHERI register access is disabled for the current privilege.

Prerequisites for *Capability Pointer Mode*

Zcheripurecap, H

Prerequisites for *Integer Pointer Mode*

Zcheripurecap, Zcherihybrid, H

Capability Pointer Mode Operation

TBD

Integer Pointer Mode Operation

TBD

12.12.14. HLVX.HU

See HLVX.WU.

12.12.15. HLVX.WU

Synopsis

Hypervisor virtual machine load from executable memory

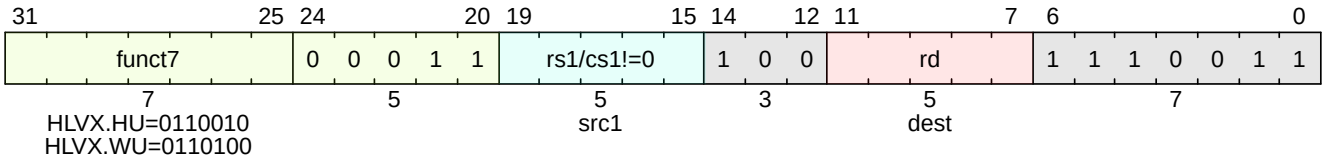
Capability Pointer Mode Mnemonics

```
hlvx.hu rd, cs1
hlvx.wu rd, cs1
```

Integer Pointer Mode Mnemonics

```
hlvx.hu rd, rs1
hlvx.wu rd, rs1
```

Encoding



Capability Pointer Mode Description

Performs a load with the **execute** permission taking the place of **read** permission during address translation and as though V=1; i.e., with the address translation and protection, and endianness, that apply to memory access in either VS-mode or VU-mode. The effective address is the address of `cs1`. The authorising capability for the operation is `cs1`. A copy of the loaded value is written to `rd`.

Any instance of this instruction with a `cs1` of `c0` would certainly trap (with a CHERI tag violation), as `c0` is defined to always hold a NULL capability. As such, the encodings with a `cs1` of `c0` are RESERVED for use by future extensions.

Integer Pointer Mode Description

Performs a load with the **execute** permission taking the place of **read** permission during address translation and as though V=1; i.e., with the address translation and protection, and endianness, that apply to memory access in either VS-mode or VU-mode. The effective address is `rs1`. The authorising capability for the operation is ddc. A copy of the loaded value is written to `rd`.

Exceptions

CHERI fault exceptions occur when the authorising capability fails one of the checks listed below; in this case, *CHERI data fault* is reported in the mtval2 or stval2 TYPE field and the corresponding code is written to CAUSE.

CAUSE	Reason
Tag violation	Authority capability tag set to 0, or has any reserved bits set
Seal violation	Authority capability is sealed
Permission violation	Authority capability does not grant R-permission or X-permission, or the AP field could not have been produced by ACPERM

CAUSE	Reason
Invalid address violation	The effective address is invalid according to Invalid address conversion
Bounds violation	At least one byte accessed is outside the authority capability bounds, or the capability has malformed bounds

Prerequisites for *Capability Pointer Mode*

Zcheripurecap, H

Prerequisites for *Integer Pointer Mode*

Zcheripurecap, Zcherihybrid, H

Capability Pointer Mode Operation

TBD

Integer Pointer Mode Operation

TBD

Appendix A: CHERI System Implications

CHERI processors need memory systems which support the capability validity tags in memory.

There are, or will soon be, a wide range of CHERI systems in existence from tiny IoT devices up to server chips.

There are two types of bus connections used in SoCs which contain CHERI CPUs:

1. Tag-aware busses, where the bus protocol is extended to carry the tag along with the data. This is typically done using user defined bits in the protocol.
   a. These busses will read tags from memory (if tags are present in the target memory) and return them to the requestor.
   b. These busses will write the tag to memory as an extension of the data write.
2. Non-tag aware busses, i.e. current non-CHERI aware busses.
   a. Reads of tagged memory will not read the tag.
   b. Writes to tagged memory will clear the tag of any CLEN-aligned CLEN-wide memory location where any byte matches the memory write.

The fundamental rule for any CHERI system is that the tag and data are always accessed atomically. For every naturally aligned CLEN-wide memory location, it must never be possible to:

1. Update any data bytes without also writing the tag
   a. This implies clearing the tag if a non-CHERI aware bus master overwrites a capability in memory
2. Read a tagged value with mismatched (stale or newer) data
3. Set the tag without also writing the data.

Clearing tags in memory does not necessarily require updating the associated data.

A.1. Small CHERI system example

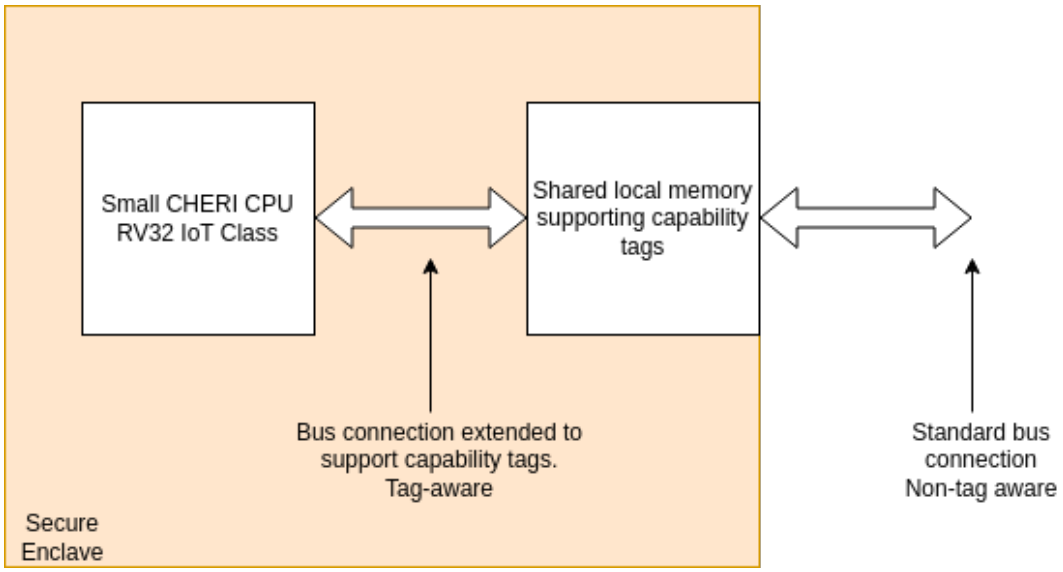


Figure 69. Example small CHERI system with local capability tag storage

This example shows a minimum sized system where only the local memory is extended to support capability tags. The tag-aware region is highlighted. All tags are created by the CHERI CPU, and only stored locally. The memory is shared with the system, probably via a secure DMA, which is not tag aware.

Therefore the connection between CPU and memory is tag-aware, and the connection to the system is not tag aware.

All writes from the system port to the memory must clear any memory tags to follow the rules from above.

A.2. Large CHERI system example

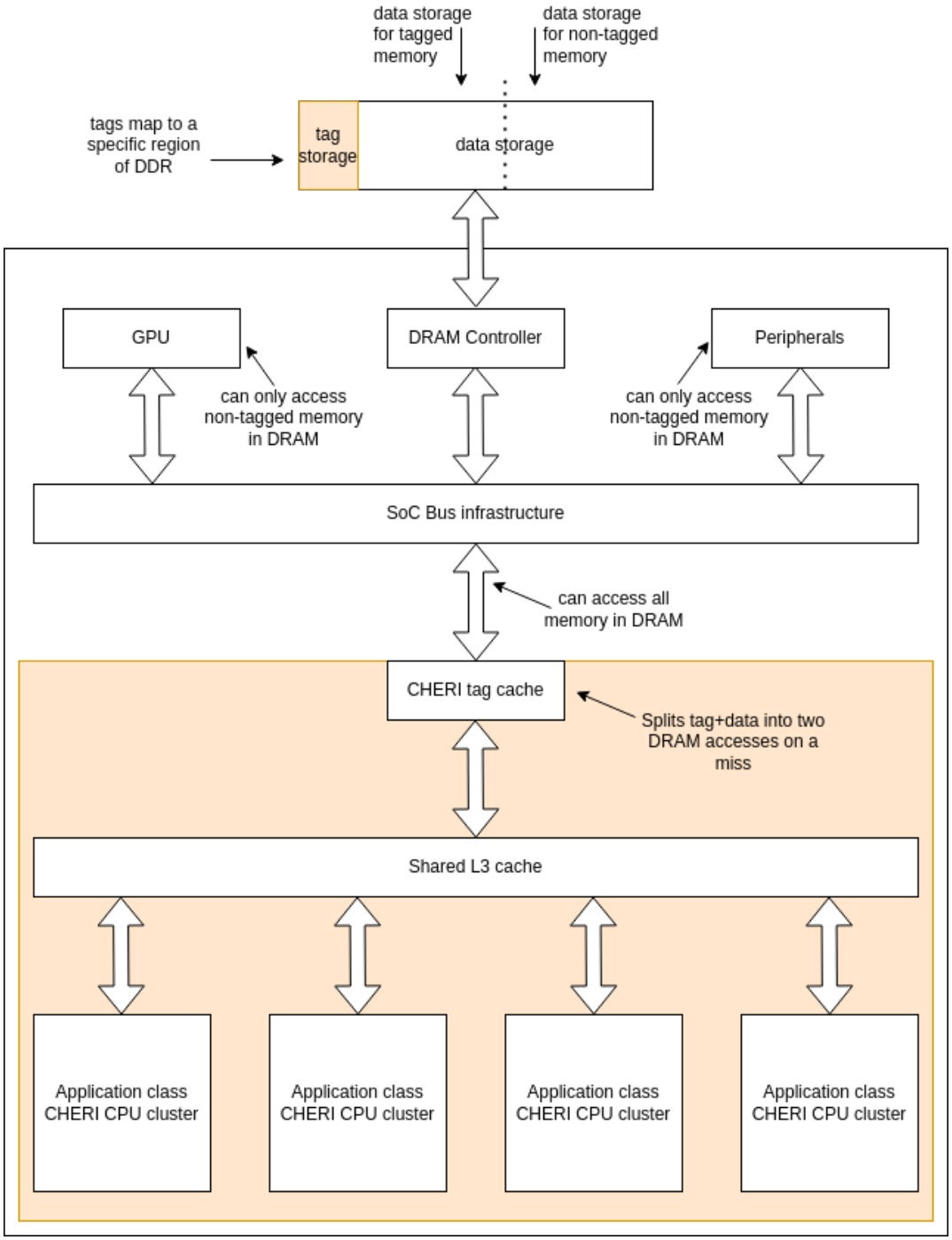


Figure 70. Example large CHERI system with tag cache

In the case of a large CHERI SoC with caches, all the cached memory visible to the CHERI CPUs must support tags. All memory is backed up by DRAM, and standard DRAM does not offer 129-bit words and so a typical system will have a tag cache IP.

A region of DRAM is reserved for CHERI tag storage.

The tag cache sits on the boundary of the tag-aware and non-tag-aware memory domains, and it provides the bridge between the two. It stores tags locally in its cache, and if there is a miss, it will create an extra bus request to access the region of DRAM reserved for tag storage. Therefore in the case of a miss a single access is split into two - one to access the data and one to access the tag.

The key property of the tag cache is to preserve the atomic access of data and tags in the memory system so that all CPUs have a consistent view of tags and data.

The region of DRAM reserved for tag storage must be only accessible by the tag cache, therefore no bus initiators should be able to write to the DRAM without the transactions passing through the tag cache.

Therefore the GPUs and peripherals cannot write to the tag storage in the DRAM, or the tagged memory data storage region. These constraints will be part of the design of the network-on-chip. It *is* possible for the GPU and peripherals to read the tagged memory data storage region of the DRAM, if required.

It would be possible to allow a DMA to access the tagged memory region of the DRAM directly to allow swap to/from DRAM and external devices such as flash. This will require the highest level of security in the SoC, as the CHERI protection model relies on the integrity of the tags, and so the root-of-trust will need to authenticate and encrypt the transfer, with anti-rollback protection.

For further information on the tag cache see (*Efficient Tagged Memory*, 2017).

A.3. Large CHERI pure-capability system example

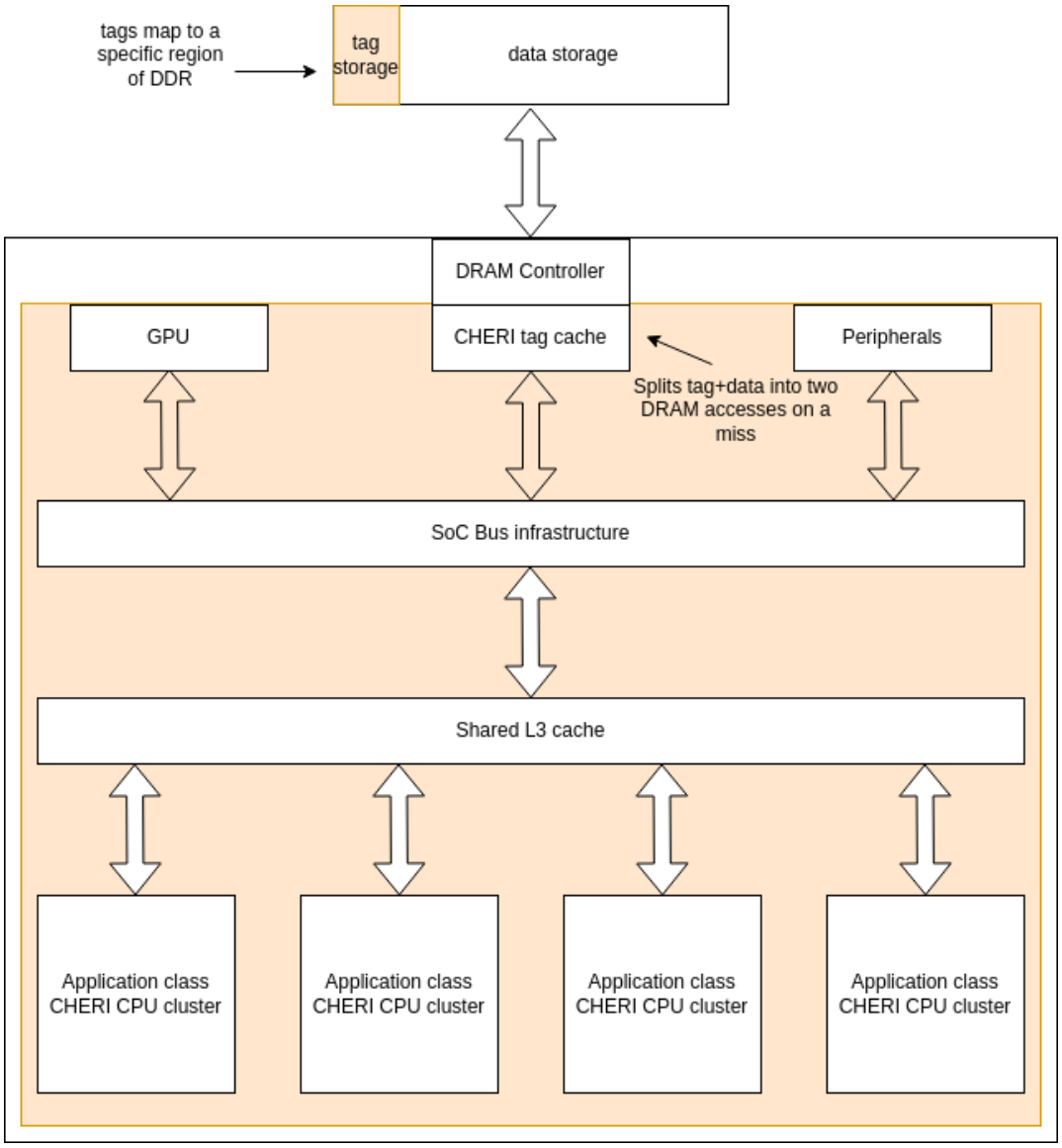


Figure 71. Example large CHERI system with only tag-aware bus masters

In this example every DRAM access passes through the tag cache, and so *all* bus masters are tag-aware and can access the tagged memory if permitted by the network-on-chip.

The system topology is simpler than in Figure 70.

There is likely to be a performance difference between the two systems. The main motivation for Figure 70 is to avoid the GPU DRAM traffic needing to look-up every tag in the tag cache, potentially adding overhead to every transaction.

Appendix B: Extension summary

B.1. Zabhlrsc

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

These instructions are not controlled by the CRE bits in mseccfg, menvcfg or senvcfg.

Table 43. Zabhlrsc instruction extension

Mnemonic	Zabhlrsc	Function
LR.H	✔	Load reserved half
LR.B	✔	Load reserved byte
SC.H	✔	Store conditional half
SC.B	✔	Store conditional byte

B.2. Zish4add

Zish4add is a separate extension independent of CHERI, but improves performance for CHERI code as the natural data width of pointers has doubled.

These instructions are not controlled by the CRE bits in mseccfg, menvcfg or senvcfg.

Table 44. Zish4add instruction extension

Mnemonic	Zish4add	Function
SH4ADD	✔	shift and add, representability check in Capability Mode
SH4ADD.UW	✔	shift and add unsigned words, representability check in Capability Mode

B.3. Zcheripurecap

Zcheripurecap defines the set of instructions supported by a core when in *Capability Pointer Mode*.

Some instructions depend on the presence of other extensions, as listed in Table 45.

Table 45. Zcheripurecap instruction extension - Pure Capability Pointer Mode instructions

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
LC	✔	✔													Load cap via capability register
SC	✔	✔													Store cap via capability register

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
C.LCSP	✔	✔				✔									Load cap capability, SP relative
C.SCSP	✔	✔				✔									Store cap capability, SP relative
C.LC	✔	✔				✔									Load cap capability
C.SC	✔	✔				✔									Store cap capability
C.LWSP	✔	✔				✔									Load word capability, SP relative
C.SWSP	✔	✔				✔									Store word capability, SP relative
C.LW	✔	✔				✔									Load word capability
C.SW	✔	✔				✔									Store word capability
C.LD		✔				✔									Load word capability
C.SD		✔				✔									Store word capability
C.LDSP		✔				✔									Load word capability
C.SDSP		✔				✔									Store word capability
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

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
C.LHU	✔	✔						✔							Load unsigned half
LWU		✔													Load unsigned word
LD		✔													Load double
SB	✔	✔													Store byte
C.SB	✔	✔						✔							Store byte
SH	✔	✔													Store half
C.SH	✔	✔						✔							Store half
SW	✔	✔													Store word
SD		✔													Store double
AUIPC	✔	✔													Add immediate to PCC address
CADD	✔	✔													Increment cap address by register, representability check
CADDI	✔	✔													Increment cap address by immediate, representability check
SCADDR	✔	✔													Replace capability address, representability check
GCTAG	✔	✔													Get tag field
GCPERM	✔	✔													Get hperm and uperm fields as 1-bit per permission, packed together
CMV	✔	✔													Move capability register
ACPERM	✔	✔													AND capability permissions (expand to 1-bit per permission before ANDing)

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
GCHI	✔	✔													Get metadata
SCHI	✔	✔													Set metadata and clear tag
SCEQ	✔	✔													Full capability bitwise compare, set result true if both are fully equal
SENTRY	✔	✔													Seal capability
SCSS	✔	✔													Set result true if cs1 and cs1 tags match and cs2 bounds and permissions are a subset of cs1
CBLD	✔	✔													Set cd to cs2 with its tag set after checking that cs2 is a subset of cs1
SCBNDS	✔	✔													Set register bounds on capability with rounding, clear tag if rounding is required
SCBNDSI	✔	✔													Set immediate bounds on capability with rounding, clear tag if rounding is required
SCBNDSR	✔	✔													Set bounds on capability with rounding up as required
CRAM	✔	✔													Representable Alignment Mask: Return mask to apply to address to get the requested bounds

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
GCBASE	✔	✔													Get capability base
GCLEN	✔	✔													Get capability length
GCTYPE	✔	✔													Get capability type
C.ADDI16SP	✔	✔				✔									ADD immediate to stack pointer, CADD in Capability Mode
C.ADDI4SPN	✔	✔				✔									ADD immediate to stack pointer, CADDI in Capability Mode
C.MV	✔	✔				✔									Register Move, cap reg move in Capability Mode
C.J	✔	✔				✔									Jump to PC+offset, bounds check minimum size target instruction
C.JAL	✔					✔									Jump to PC+offset, bounds check minimum size target instruction, link to cd
JAL	✔	✔				✔									Jump to PC+offset, bounds check minimum size target instruction, link to cd
JALR	✔	✔													Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
C.JALR	✔	✔				✔									Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap
C.JR	✔	✔				✔									Indirect cap jump, bounds check minimum size target instruction, unseal target cap
DRET	✔	✔													Return from debug mode, sets ddc from dddc and pcc from dpcc
MRET	✔	✔													Return from machine mode handler, sets pcc from mtvecc , needs ASR-permission
SRET	✔	✔													Return from supervisor mode handler, sets pcc from stvecc, needs ASR-permission
CSRRW	✔	✔													CSR write - can also read/write a full capability through an address alias
CSRRS	✔	✔													CSR set - can also read/write a full capability through an address alias
CSRRC	✔	✔													CSR clear - can also read/write a full capability through an address alias

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
CSRRWI	✔	✔													CSR write - can also read/write a full capability through an address alias
CSRRSI	✔	✔													CSR set - can also read/write a full capability through an address alias
CSRRCI	✔	✔													CSR clear - can also read/write a full capability through an address alias
CBO.INVAL	✔	✔			✔										Cache block invalidate (implemented as clean)
CBO.CLEAN	✔	✔			✔										Cache block clean
CBO.FLUSH	✔	✔			✔										Cache block flush
CBO.ZERO	✔	✔			✔										Cache block zero
PREFETCH.R	✔	✔			✔										Prefetch instruction cache line, always valid
PREFETCH.W	✔	✔			✔										Prefetch read-only data cache line
PREFETCH.I	✔	✔			✔										Prefetch writeable data cache line
LR.C	✔	✔	✔												Load reserved capability
LR.D			✔												Load reserved double
LR.W			✔												Load reserved word
LR.H	✔	✔		✔											Load reserved half
LR.B	✔	✔		✔											Load reserved byte
SC.C	✔	✔	✔												Store conditional capability

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
SC.D			✔												Store conditional double
SC.W			✔												Store conditional word
SC.H	✔	✔		✔											Store conditional half
SC.B	✔	✔		✔											Store conditional byte
AMOSWAP.C	✔	✔	✔												Atomic swap of cap
AMO<OP>.W	✔	✔	✔												Atomic op of word
AMO<OP>.D		✔	✔												Atomic op of double
C.FLD	✔												✔		Load floating point double
C.FLDSP	✔												✔		Load floating point double, sp relative
C.FSD	✔												✔		Store floating point double
C.FSDSP	✔												✔		Store floating point double, sp relative
FLH	✔	✔									✔				Load floating point half capability
FSH	✔	✔									✔				Store floating point half capability
FLW	✔	✔										✔			Load floating point word capability
FSW	✔	✔										✔			Store floating point word capability
FLD	✔	✔											✔		Load floating point double capability

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
FSD	✔	✔											✔		Store floating point double capability
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
ADD.UW		✔					✔								add unsigned words, representability check in Capability Mode
SH1ADD	✔	✔					✔								shift and add, representability check in Capability Mode
SH1ADD.UW		✔					✔								shift and add unsigned words, representability check in Capability Mode
SH2ADD	✔	✔					✔								shift and add, representability check in Capability Mode

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
SH2ADD.UW		✔					✔								shift and add unsigned words, representability check in Capability Mode
SH3ADD	✔	✔					✔								shift and add, representability check in Capability Mode
SH3ADD.UW		✔					✔								shift and add unsigned words, representability check in Capability Mode
SH4ADD		✔													shift and add, representability check in Capability Mode
SH4ADD.UW		✔													shift and add unsigned words, representability check in Capability Mode
HLV.B	✔	✔													Hypervisor virtual machine load byte
HLV.BU	✔	✔													Hypervisor virtual machine load unsigned byte
HLV.H	✔	✔													Hypervisor virtual machine load half word
HLV.HU	✔	✔													Hypervisor virtual machine load unsigned half word
HLV.W	✔	✔													Hypervisor virtual machine load word
HLV.WU		✔													Hypervisor virtual machine load unsigned word

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
HLV.D		✔													Hypervisor virtual machine load double
HLV.C	✔	✔													Hypervisor virtual machine load capability
HSV.B	✔	✔													Hypervisor virtual machine store byte
HSV.H	✔	✔													Hypervisor virtual machine store half word
HSV.W	✔	✔													Hypervisor virtual machine store word
HSV.D		✔													Hypervisor virtual machine store double
HSV.C	✔	✔													Hypervisor virtual machine store capability
HLVX.HU	✔	✔													Hypervisor virtual machine load half word from executable memory
HLVX.WU	✔	✔													Hypervisor virtual machine load wordfrom executable memory

B.4. Zcherihybrid

Zcherihybrid defines the set of instructions added by the *Integer Pointer Mode*, in addition to Zcheripurecap.

Zcherihybrid implies Zcheripurecap

Table 46. Zcherihybrid instruction extension - Integer Pointer Mode instructions

Mnemonic	RV32	RV64	A	Zabhlrsc	Zicbo[mpz]	C or Zca	Zba	Zcb	Zcmp	Zcmt	Zfh	F	D	V	Function
SCMODE	✔	✔													Set the mode bit of a capability, no permissions required
GCMODE	✔	✔													Get the mode bit of a capability, no permissions required
MODESW.CAP	✔	✔													Directly switch mode into *Capability Pointer Mode*
MODESW.INT	✔	✔													Directly switch mode into *Integer Pointer Mode*
C.FLW	✔											✔			Load floating point word capability
C.FLWSP	✔											✔			Load floating point word, sp relative
C.FSW	✔											✔			Store floating point word capability
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

Appendix C: Capability Width CSR Summary

Table 47. CSRs renamed and extended to capability width

CLEN CSR	Alias	Prerequisites
dpcc	dpc	Sdext
dscratch0c	dscratch0	Sdext
dscratch1c	dscratch1	Sdext
mtvecc	mtvec	M-mode
mscratchc	mscratch	M-mode
mepcc	mepc	M-mode
stvecc	stvec	S-mode
sscratchc	sscratch	S-mode
sepcc	sepc	S-mode
vstvecc	vstvec	H
vsscratchc	vsscratch	H
vsepcc	vsepc	H
jvtc	jvt	Zcmt
utidc	utid	Zstid
stidc	stid	Zstid
vstidc	vstid	Zstid
mtidc	mtid	Zstid

Table 48. Action taken on writing to extended CSRs[**]

CLEN CSR	Action on XLEN write	Action on CLEN write
dpcc	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change
dscratch0c	Update the CSR using SCADDR.	direct write
dscratch1c	Update the CSR using SCADDR.	direct write
mtvecc	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change, including the MODE field in the address for simplicity. Vector range check [*] if vectored mode is programmed.	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change, including the MODE field in the address for simplicity. Vector range check [*] if vectored mode is programmed.
mscratchc	Update the CSR using SCADDR.	direct write

CLEN CSR	Action on XLEN write	Action on CLEN write
mepcc	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change
stvecc	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.
sscratchc	Update the CSR using SCADDR.	direct write
sepcc	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change
vstvecc	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.
vsscratchc	Update the CSR using SCADDR.	direct write
vsepcc	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change
jvtc	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change
utidc	Update the CSR using SCADDR.	direct write
stidc	Update the CSR using SCADDR.	direct write
vstidc	Update the CSR using SCADDR.	direct write
mtidc	Update the CSR using SCADDR.	direct write

* The vector range check is to ensure that vectored entry to the handler is within bounds of the capability written to `Xtvecc`. The check on writing must include the lowest (0 offset) and highest possible offset (e.g. 64 * MXLEN bits where HICAUSE=16).

** XLEN bits of extended capability CSRs are written when executing CSRRWI, CSRRC, CSRRS, CSRRCI or CSRRSI regardless of the CHERI execution mode. When using CSRRW, CLEN bits are written when the CHERI execution mode is *Capability Pointer Mode* and XLEN bits are written when the mode is *Integer Pointer Mode*; therefore, writing XLEN bits with CSRRW is only possible when

Zcherihybrid is implemented.

Table 49. Action taken on writing to new capability CSRs[+]

CLEN CSR	Action on XLEN write	Action on CLEN write
dddc	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change
mtdc	Update the CSR using SCADDR.	direct write
stdc	Update the CSR using SCADDR.	direct write
vstdc	Update the CSR using SCADDR.	direct write
ddc	Apply Invalid address conversion. Always update the CSR with SCADDR even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change
dinfc	Ignore	Ignore

[+] XLEN bits of new capability CSRs added in Zcherihybrid are written when executing CSRRWI, CSRRC, CSRRS, CSRRCI or CSRRSI regardless of the CHERI execution mode. CLEN bits are always written when using CSRRW regardless of the CHERI execution mode.

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.*

Table 50. CLEN-wide CSRs storing executable vectors or data pointers

CLEN CSR	Executable Vector	Data Pointer	Unseal On Execution
dpcc	✔		✔
mtvecc	✔		
mepcc	✔		✔
stvecc	✔		
sepcc	✔		✔
vstvecc	✔		
vsepcc	✔		✔
jvtc	✔		
dddc		✔	
ddc		✔	

Some CSRs store executable vectors or data pointers as shown in Table 50. These CSRs do not need to store the full width address on RV64. If they store fewer address bits then writes are subject to the invalid address check in Invalid address conversion.

Table 51. CLEN-wide CSRs which store all CLEN+1 bits

CLEN CSR	Store full metadata
dscratch0c	✔
dscratch1c	✔
mscratchc	✔
sscratchc	✔
vsscratchc	✔
dinfc	✔
utidc	✔
stidc	✔
vstidc	✔
mtidc	✔

Table 51 shows which CLEN-wide CSRs store all CLEN+1 bits. No other CLEN-wide CSRs store any reserved bits. All CLEN-wide CSRs store *all* non-reserved metadata fields.

Table 52. All CLEN-wide CSRs. Zcheripurecap is a prerequisite for all CSRs in this table

CLEN CSR	Prerequisites	Address	Permissions	Reset Value	Description
dpcc	Sdext	0x7b1	DRW	tag=0, otherwise undefined	Debug Program Counter Capability
dscratch0c	Sdext	0x7b2	DRW	tag=0, otherwise undefined	Debug Scratch Capability 0
dscratch1c	Sdext	0x7b3	DRW	tag=0, otherwise undefined	Debug Scratch Capability 1
mtvecc	M-mode	0x305	MRW, ASR-permission	Infinite	Machine Trap-Vector Base-Address Capability
mscratchc	M-mode	0x340	MRW, ASR-permission	tag=0, otherwise undefined	Machine Scratch Capability
mepcc	M-mode	0x341	MRW, ASR-permission	Infinite	Machine Exception Program Counter Capability
stvecc	S-mode	0x105	SRW, ASR-permission	Infinite	Supervisor Trap-Vector Base-Address Capability
sscratchc	S-mode	0x140	SRW, ASR-permission	tag=0, otherwise undefined	Supervisor Scratch Capability
sepcc	S-mode	0x141	SRW, ASR-permission	Infinite	Supervisor Exception Program Counter Capability

CLEN CSR	Prerequisites	Address	Permissions	Reset Value	Description
vstvecc	H	0x205	HRW, ASR-permission	Infinite	Virtual Supervisor Trap-Vector Base-Address Capability
vsscratchc	H	0x240	HRW, ASR-permission	tag=0, otherwise undefined	Virtual Supervisor Scratch Capability
vsepcc	H	0x241	HRW, ASR-permission	Infinite	Virtual Supervisor Exception Program Counter Capability
jvtc	Zcmt	0x017	URW	tag=0, otherwise undefined	Jump Vector Table Capability
dddc	Zcheri hybrid, Sdext	0x7bc	DRW	tag=0, otherwise undefined	Debug Default Data Capability (saved/restored on debug mode entry/exit)
mtdc	Zcheri hybrid, M-mode	0x74c	MRW, ASR-permission	tag=0, otherwise undefined	Machine Trap Data Capability (scratch register)
stdc	Zcheri hybrid, S-mode	0x163	SRW, ASR-permission	tag=0, otherwise undefined	Supervisor Trap Data Capability (scratch register)
vstdc	Zcheri hybrid, H	0x245	HRW, ASR-permission	tag=0, otherwise undefined	Virtual Supervisor Trap Data Capability (scratch register)
ddc	Zcheri hybrid	0x416	URW	Infinite	User Default Data Capability
dinfc	Sdext	0x7bd	DRW	Infinite	Source of Infinite capability in debug mode, writes are ignored
utidc	Zstid	0x480	Read: U, Write: U, ASR-permission	tag=0, otherwise undefined	User thread ID
stidc	Zstid	0x580	Read: S, Write: S, ASR-permission	tag=0, otherwise undefined	Supervisor thread ID
vstidc	Zstid	0xA80	Read: VS, Write: VS, ASR-permission	tag=0, otherwise undefined	Virtual supervisor thread ID
mtidc	Zstid	0x780	Read: M, Write: M, ASR-permission	tag=0, otherwise undefined	Machine thread ID

Appendix D: Instructions and CHERI Execution Mode

Table 53, Table 54 and Table 55 summarise on which CHERI execution mode each instruction may be executed in.

Table 53. Instructions valid for execution in Capability Pointer Mode only

Mnemonic	Zcherihybrid	Zcheripurecap	Function
C.LCSP		✔	Load cap capability, SP relative
C.SCSP		✔	Store cap capability, SP relative
C.LC		✔	Load cap capability
C.SC		✔	Store cap capability

Table 54. Instructions valid for execution in Integer Pointer Mode only

Mnemonic	Zcherihybrid	Zcheripurecap	Function
C.FLW	✔		Load floating point word capability
C.FLWSP	✔		Load floating point word, sp relative
C.FSW	✔		Store floating point word capability
C.FSWSP	✔		Store floating point word, sp relative
C.FLD	✔		Load floating point double
C.FLDSP	✔		Load floating point double, sp relative
C.FSD	✔		Store floating point double
C.FSDSP	✔		Store floating point double, sp relative

Table 55. Instructions valid for execution in both Integer Pointer Mode and Capability Pointer Mode

Mnemonic	Zcherihybrid	Zcheripurecap	Function
LC	✔	✔	Load cap via int pointer
SC	✔	✔	Store cap via int pointer
C.LWSP	✔	✔	Load word capability, SP relative
C.SWSP	✔	✔	Store word capability, SP relative
C.LW	✔	✔	Load word capability
C.SW	✔	✔	Store word capability
C.LD	✔	✔	Load word capability
C.SD	✔	✔	Store word capability
C.LDSP	✔	✔	Load word capability
C.SDSP	✔	✔	Store word capability
LB	✔	✔	Load signed byte
LH	✔	✔	Load signed half

Mnemonic	Zcherihybrid	Zcheripurecap	Function
C.LH	✔	✔	Load signed half
LW	✔	✔	Load signed word
LBU	✔	✔	Load unsigned byte
C.LBU	✔	✔	Load unsigned byte
LHU	✔	✔	Load unsigned half
C.LHU	✔	✔	Load unsigned half
LWU	✔	✔	Load unsigned word
LD	✔	✔	Load double
SB	✔	✔	Store byte
C.SB	✔	✔	Store byte
SH	✔	✔	Store half
C.SH	✔	✔	Store half
SW	✔	✔	Store word
SD	✔	✔	Store double
AUIPC	✔	✔	Add immediate to PCC address
CADD	✔	✔	Increment cap address by register, representability check
CADDI	✔	✔	Increment cap address by immediate, representability check
SCADDR	✔	✔	Replace capability address, representability check
GCTAG	✔	✔	Get tag field
GCPERM	✔	✔	Get hperm and uperm fields as 1-bit per permission, packed together
CMV	✔	✔	Move capability register
ACPERM	✔	✔	AND capability permissions (expand to 1-bit per permission before ANDing)
GCHI	✔	✔	Get metadata
SCHI	✔	✔	Set metadata and clear tag
SCEQ	✔	✔	Full capability bitwise compare, set result true if both are fully equal
SENTRY	✔	✔	Seal capability
SCSS	✔	✔	Set result true if cs1 and cs1 tags match and cs2 bounds and permissions are a subset of cs1
CBLD	✔	✔	Set cd to cs2 with its tag set after checking that cs2 is a subset of cs1
SCBNDS	✔	✔	Set register bounds on capability with rounding, clear tag if rounding is required

Mnemonic	Zcherihybrid	Zcheripurecap	Function
SCBNDSI	✔	✔	Set immediate bounds on capability with rounding, clear tag if rounding is required
SCBNDSR	✔	✔	Set bounds on capability with rounding up as required
CRAM	✔	✔	Representable Alignment Mask: Return mask to apply to address to get the requested bounds
GCBASE	✔	✔	Get capability base
GCLEN	✔	✔	Get capability length
GCTYPE	✔	✔	Get capability type
SCMODE	✔		Set the mode bit of a capability, no permissions required
GCMODE	✔		Get the mode bit of a capability, no permissions required
MODESW.CAP	✔		Directly switch mode into *Capability Pointer Mode*
MODESW.INT	✔		Directly switch mode into *Integer Pointer Mode*
C.ADDI16SP	✔	✔	ADD immediate to stack pointer, CADD in Capability Mode
C.ADDI4SPN	✔	✔	ADD immediate to stack pointer, CADDI in Capability Mode
C.MV	✔	✔	Register Move, cap reg move in Capability Mode
C.J	✔	✔	Jump to PC+offset, bounds check minimum size target instruction
C.JAL	✔	✔	Jump to PC+offset, bounds check minimum size target instruction, link to cd
JAL	✔	✔	Jump to PC+offset, bounds check minimum size target instruction, link to cd
JALR	✔	✔	Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap
C.JALR	✔	✔	Indirect cap jump and link, bounds check minimum size target instruction, unseal target cap, seal link cap
C.JR	✔	✔	Indirect cap jump, bounds check minimum size target instruction, unseal target cap
DRET	✔	✔	Return from debug mode, sets ddc from dddc and pcc from dpcc
MRET	✔	✔	Return from machine mode handler, sets pcc from mtvecc , needs ASR-permission
SRET	✔	✔	Return from supervisor mode handler, sets pcc from stvecc, needs ASR-permission

Mnemonic	Zcherihybrid	Zcheripurecap	Function
CSRRW	✔	✔	CSR write - can also read/write a full capability through an address alias
CSRRS	✔	✔	CSR set - can also read/write a full capability through an address alias
CSRRC	✔	✔	CSR clear - can also read/write a full capability through an address alias
CSRRWI	✔	✔	CSR write - can also read/write a full capability through an address alias
CSRRSI	✔	✔	CSR set - can also read/write a full capability through an address alias
CSRRCI	✔	✔	CSR clear - can also read/write a full capability through an address alias
CBO.INVAL	✔	✔	Cache block invalidate (implemented as clean)
CBO.CLEAN	✔	✔	Cache block clean
CBO.FLUSH	✔	✔	Cache block flush
CBO.ZERO	✔	✔	Cache block zero
PREFETCH.R	✔	✔	Prefetch instruction cache line, always valid
PREFETCH.W	✔	✔	Prefetch read-only data cache line
PREFETCH.I	✔	✔	Prefetch writeable data cache line
LR.C	✔	✔	Load reserved capability
LR.D	✔	✔	Load reserved double
LR.W	✔	✔	Load reserved word
LR.H	✔	✔	Load reserved half
LR.B	✔	✔	Load reserved byte
SC.C	✔	✔	Store conditional capability
SC.D	✔	✔	Store conditional double
SC.W	✔	✔	Store conditional word
SC.H	✔	✔	Store conditional half
SC.B	✔	✔	Store conditional byte
AMOSWAP.C	✔	✔	Atomic swap of cap
AMO<OP>.W	✔	✔	Atomic op of word
AMO<OP>.D	✔	✔	Atomic op of double
C.FLD	✔	✔	Load floating point double
C.FLDSP	✔	✔	Load floating point double, sp relative
C.FSD	✔	✔	Store floating point double
C.FSDSP	✔	✔	Store floating point double, sp relative
FLH	✔	✔	Load floating point half capability

Mnemonic	Zcherihybrid	Zcheripurecap	Function
FSH	✔	✔	Store floating point half capability
FLW	✔	✔	Load floating point word capability
FSW	✔	✔	Store floating point word capability
FLD	✔	✔	Load floating point double capability
FSD	✔	✔	Store floating point double capability
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
ADD.UW	✔	✔	add unsigned words, representability check in Capability Mode
SH1ADD	✔	✔	shift and add, representability check in Capability Mode
SH1ADD.UW	✔	✔	shift and add unsigned words, representability check in Capability Mode
SH2ADD	✔	✔	shift and add, representability check in Capability Mode
SH2ADD.UW	✔	✔	shift and add unsigned words, representability check in Capability Mode
SH3ADD	✔	✔	shift and add, representability check in Capability Mode
SH3ADD.UW	✔	✔	shift and add unsigned words, representability check in Capability Mode
SH4ADD	✔	✔	shift and add, representability check in Capability Mode
SH4ADD.UW	✔	✔	shift and add unsigned words, representability check in Capability Mode
HLV.B	✔	✔	Hypervisor virtual machine load byte
HLV.BU	✔	✔	Hypervisor virtual machine load unsigned byte
HLV.H	✔	✔	Hypervisor virtual machine load half word
HLV.HU	✔	✔	Hypervisor virtual machine load unsigned half word
HLV.W	✔	✔	Hypervisor virtual machine load word
HLV.WU	✔	✔	Hypervisor virtual machine load unsigned word

Mnemonic	Zcherihybrid	Zcheripurecap	Function
HLV.D	✔	✔	Hypervisor virtual machine load double
HLV.C	✔	✔	Hypervisor virtual machine load capability
HSV.B	✔	✔	Hypervisor virtual machine store byte
HSV.H	✔	✔	Hypervisor virtual machine store half word
HSV.W	✔	✔	Hypervisor virtual machine store word
HSV.D	✔	✔	Hypervisor virtual machine store double
HSV.C	✔	✔	Hypervisor virtual machine store capability
HLVX.HU	✔	✔	Hypervisor virtual machine load half word from executable memory
HLVX.WU	✔	✔	Hypervisor virtual machine load wordfrom executable memory

Table 56. Mnemonics with the same encoding but mapped to different instructions in Integer Pointer Mode and Capability Pointer Mode

Mnemonic	*Integer Pointer Mode* mnemonic RV32	*Integer Pointer Mode* mnemonic RV64
C.LCSP	C.FLWSP	C.FLDSP
C.SCSP	C.FSWSP	C.FSDSP
C.LC	C.FLW	C.FLD
C.SC	C.FSW	C.FSD

Table 57. Instruction encodings which vary depending on the current XLEN

Mnemonic	Function
C.LCSP	Load cap capability, SP relative
C.SCSP	Store cap capability, SP relative
C.LC	Load cap capability
C.SC	Store cap capability

MODESW.CAP, *MODESW.INT* and *SCMODE* only exist in Capability Pointer Mode if Integer Pointer Mode is also present. A hart does not support the *M-bit* if it does not implement the Zcherihybrid extension.

Table 58. Conditions for detecting illegal CHERI instructions

Mnemonic	illegal insn if (1)	OR illegal insn if (2)	OR illegal insn if (3)
C.J	mode==D (optional)		
C.JAL	mode==D (optional)		
JAL	mode==D (optional)		
JALR	mode==D (optional)		
C.JALR	mode==D (optional)		
C.JR	mode==D (optional)		

Mnemonic	illegal insn if (1)	OR illegal insn if (2)	OR illegal insn if (3)
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	
C.FLW	Xstatus.fs==0		
C.FLWSP	Xstatus.fs==0		
C.FSW	Xstatus.fs==0		
C.FSWSP	Xstatus.fs==0		
C.FLD	Xstatus.fs==0		
C.FLDSP	Xstatus.fs==0		
C.FLD	Xstatus.fs==0		
C.FLDSP	Xstatus.fs==0		
C.FSD	Xstatus.fs==0		
C.FSDSP	Xstatus.fs==0		
C.FSD	Xstatus.fs==0		
C.FSDSP	Xstatus.fs==0		
FLH	Xstatus.fs==0		
FSH	Xstatus.fs==0		
FLW	Xstatus.fs==0		
FSW	Xstatus.fs==0		
FLD	Xstatus.fs==0		
FSD	Xstatus.fs==0		

Mnemonic	illegal insn if (1)	OR illegal insn if (2)	OR illegal insn if (3)
HLV.B	V=1	MODE==U AND hstatus.HU=0	
HLV.BU	V=1	MODE==U AND hstatus.HU=0	
HLV.H	V=1	MODE==U AND hstatus.HU=0	
HLV.HU	V=1	MODE==U AND hstatus.HU=0	
HLV.W	V=1	MODE==U AND hstatus.HU=0	
HLV.WU	V=1	MODE==U AND hstatus.HU=0	
HLV.D	V=1	MODE==U AND hstatus.HU=0	
HLV.C	V=1	MODE==U AND hstatus.HU=0	
HSV.B	V=1	MODE==U AND hstatus.HU=0	
HSV.H	V=1	MODE==U AND hstatus.HU=0	
HSV.W	V=1	MODE==U AND hstatus.HU=0	
HSV.D	V=1	MODE==U AND hstatus.HU=0	
HSV.C	V=1	MODE==U AND hstatus.HU=0	
HLVX.HU	V=1	MODE==U AND hstatus.HU=0	
HLVX.WU	V=1	MODE==U AND hstatus.HU=0	

Table 59 summarizes the behavior of a hart supporting both Zcheripurecap and Zcherihybrid in connection with the CRE and the CHERI execution mode while in a privilege other than debug mode.

Table 59. Hart's behavior depending on the effective CRE and CHERI execution mode

CRE	pcc.m	Authorizing capability[1]	New CHERI CSRs[2]	Extended CHERI CSRs[3]	CHERI instructions[4]	Compressed instructions remapped[5]	Note
0	X[6]	ddc or pcc	✘	XLEN	✘	No	*Fully RISC-V compatible*[7]
1	1	ddc or pcc	CLEN	XLEN	✔	No	*Integer Pointer Mode*
1	0	Instruction's capability operand	CLEN	CLEN	✔	Yes	*Capability Pointer Mode*

[1] Authorizing capability for memory access instructions.

[2] Whether accesses to new CHERI CSRs are permitted or raise illegal instruction exceptions. If permitted, then the bit width of the CSR read/write with CSRRW is indicated.

[3] The bit width of accesses to extended CHERI CSRs using CSRRW.

[4] Whether CHERI instructions are permitted or raise illegal instruction exceptions.

[5] See Table 56 for a list of remapped instructions.

[6] pcc.m is irrelevant when CRE=0.

[7] The hart is fully compatible with standard RISC-V when CRE=0 provided that pcc, mtvecc, mepcc, stvecc, sepcc, vstvecc, vsepcc and ddc hold the Infinite capability.

Bibliography

Efficient Tagged Memory. (2017). www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201711-iccd2017-efficient-tags.pdf

Joly, N., ElSherei, S., & Amar, S. (2020). *Security analysis of CHERI ISA*. github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf

RISC-V. (2021). *RISC-V "V" Vector Extension*. github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf

RISC-V. (2022). *RISC-V Debug Specification*. github.com/riscv/riscv-debug-spec/raw/c93823ef349286dc71a00928bddb7254e46bc3b5/riscv-debug-stable.pdf

RISC-V. (2023). *RISC-V Privileged Specification*. github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-056b6ff-2023-10-02/priv-isa-asciidoc.pdf

RISC-V. (2023). *RISC-V Unprivileged Specification*. github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-056b6ff-2023-10-02/unpriv-isa-asciidoc.pdf

RISC-V. (2023). *RISC-V Code-size Reduction Specification*. github.com/riscv/riscv-code-size-reduction/releases/download/v1.0.4-3/Zc-1.0.4-3.pdf

Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Barnes, G., Chisnall, D., Clarke, J., Davis, B., Eisen, L., Filardo, N. W., Fuchs, F. A., Grisenthwaite, R., Joannou, A., Laurie, B., Markettos, A. T., Moore, S. W., … Xia, H. (2023). *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)* (UCAM-CL-TR-987; Issue UCAM-CL-TR-987). University of Cambridge, Computer Laboratory. doi.org/10.48456/tr-987

Woodruff, J., Joannou, A., Xia, H., Fox, A., Norton, R. M., Chisnall, D., Davis, B., Gudka, K., Filardo, N. W., Markettos, A. T., & others. (2019). Cheri concentrate: Practical compressed capabilities. *IEEE Transactions on Computers*, *68*(10), 1455–1469. doi.org/10.1109/TC.2019.2914037