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, Robert Riglar, Florian Schmaus

Version v0.9.8, 20260327: Intermediate Release

Table of Contents

Preamble

This document is in the Stable state

Assume anything could still change, but limited change should be expected.

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>
- Alfredo Mazzinghi <alfredo.mazzinghi@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>
- Prashanth Mundkur <prashanth@riscv.org>
- Peter G. Neumann <neumann@csl.sri.com>
- Robert Norton <robert.norton@cl.cam.ac.uk>
- Alexander Richardson <alexrichardson@google.com>
- Robert Riglar <robert.riglar@codasip.com>
- Michael Roe <mr101@cam.ac.uk>
- Peter Rugg <peter.rugg@cl.cam.ac.uk>
- Florian Schmaus <florian.schmaus@codasip.com>
- 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>
- Jay Williams <jay.williams@codasip.com>
- Adrian Wise <adrian.wise@codasip.com>
- Jonathan Woodruff <jonathan.woodruff@cl.cam.ac.uk>
- Jason Zhijingcheng Yu <yu.zhi@comp.nus.edu.sg>

Chapter 1. Introduction

This chapter is only included in the standalone CHERI spec and not part of the integrated document.

1.1. CHERI Concepts and Terminology

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

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

Design goals include incremental adoption 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.

1.2. CHERI for RISC-V

This specification is based on publicly available documentation including CHERI v9 (Watson et al., 2023) and CHERI Concentrate (Woodruff et al., 2019).

RISC-V CHERI introduces several new bases that are informally collectively called RVY (e.g., RV32Y, RV64Y). Any CHERI RISC-V implementation must support one of these CHERI bases. CHERI will define profiles that provide equivalent functionality to the standard RVA, RVB, and RVM profiles. These RVY-based profiles (RVYA, RVYB, RVYM) will list the compatible extensions.

Most existing extensions are compatible with RVY with no modifications. Incompatible extensions will be listed in the profile documents.

Zicfiss is currently incompatible. An RVY version has not yet been developed.

This section lists new extensions available to RVY, and also extensions where the behavior is modified for RVY.

Almost all existing RISC-V extensions can be added to an implementation, but in most cases they will have some behavioral differences and/or new instructions operating on capabilities.

1.2.1. Stable Extensions and Specifications

Table 1. Unprivileged stable RVY base ISAs and extensions

Extension or Specification	Description
RV32Y, RV64Y	Base ISAs
Zysentry	Ambient capability sealing instruction
Zybld	Extension for building capabilities

Extension or Specification	Description
Zytopr	Extension for extracting the top bound
Zybndsrdw	Extension for setting bounds round down to representable length
Zyhybrid	Hybrid extension for RVI compatibility
C (RVY added instructions)	New 16-bit encodings added to Zca
Zba (RVY added instructions)	New 32-bit encodings added to Zba
Zalrsc (RVY added instructions)	New 32-bit encodings added to Zalrsc
Zaamo (RVY added instructions)	New 32-bit encodings added to Zaamo

Table 2. Unprivileged stable extensions where RVY modifies the behavior

Extension or Specification	Description
RVI (RVY modified behavior)	RVI instructions modified by RVY
C (RVY modified behavior)	C instructions modified by RVY
V (RVY modified behavior)	V instructions modified by RVY
Zicbom (RVY modified behavior)	Zicbom instructions modified by RVY
Zicbop (RVY modified behavior)	Zicbop instructions modified by RVY
Zicboz (RVY modified behavior)	Zicboz instructions modified by RVY
Zicsr (RVY modified behavior)	Zicsr instructions modified by RVY

Table 3. Unprivileged stable extension used by CHERI software

Extension or Specification	Description
Zabhlrsc	Byte and halfword LR/SC functionality

CHERI defines new state and behavior for the privileged modes:

- Machine-Level ISA (RVY)
- Supervisor-Level ISA (RVY)

Table 4. Privileged stable extensions and specifications

Extension or Specification	Description
Zyhybrid for Privileged Architectures	Hybrid extension for RVI compatibility
Supervisor-Level ISA for Virtual Memory (RV64Y)	Virtual Memory
Sdtrig (RVY)	Debug triggers
Svyrg	MMU-based capability data flow control and measurement

Table 5. Debug stable extensions and specifications

Extension or Specification	Description
Sdext (RVY)	External debug support

1.2.2. Experimental Extensions and Specifications

The extensions in this section have not been fully prototyped and so are not considered ratification ready.

Table 6. Unprivileged experimental extensions and specifications

Extension	Description
Zcmt (RV32Y)	Table Jump for RV32Y
Zcmp (RV32Y)	Push/pop and double move for RV32Y
Zyseal	Capability-mediated capability-(un)sealing instructions
Zycheriot	CHERIoT unprivileged extension

Table 7. Privileged experimental extensions and specifications

Extension	Description
Hypervisor "H" (RVY)	Hypervisor
Ssnpm, Smnpm, Smmpm, Sspm, Supm (RVY)	Pointer Masking
Smycheriot	CHERIoT privileged extension

Chapters for the unprivileged specification

Chapter 2. RV32Y and RV64Y Base Capability Instruction Sets, Version 1.0

This chapter will appear in the unpriv spec after the RV32I chapter.

This chapter introduces several new bases that are informally collectively called RVY (e.g., RV32Y, RV64Y), that extend the RV32 and RV64 instruction sets with CHERI. The CHERI bases can be composed with other standard options to bases such as E (16-registers), Zfinx and endianness.

2.1. CHERI Overview

CHERI enhances the base ISA by adding 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 are replaced by capabilities to provide memory access control.

2.2. CHERI protection model

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[1] in-memory capabilities cannot be dereferenced (*integrity*), and that rights associated with capabilities shall only ever be equal to or less permissive (*monotonicity*). Tampering or modifying capabilities in an attempt to elevate their rights will yield an invalid capability. Attempting to dereference via an invalid capability will result in a hardware exception.

[1] Not all possible corrupted states are detected, see Section 2.5, “Integrity of Capabilities”.

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 system initialization, initial capabilities are made available to software by the execution environment via general purpose 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.

2.3. Capability Registers and Format

RVY extends all registers that have to be able to hold addresses to `2*XLEN` bits (hereafter referred to as YLEN), adding metadata to protect its integrity, limit how it is manipulated, and control its use. In addition to widening to YLEN, each register also gains a one-bit capability tag which is defined below.

RVY specifies the minimum required fields that the capability format must support, and their semantics. All current RV64Y bases are little endian and therefore the memory representation of capabilities always holds the address in the first `XLEN` bytes of memory followed by the metadata. There are multiple legal (and generally software-compatible) encodings for the in-memory representation of the capability metadata, and they are described later in this chapter.

RVY is designed to be highly extensible, and therefore must allow multiple metadata encoding formats to exist. Specific encoding formats may support different ISA extensions, and add new architectural permissions. For more details on the currently available formats see Section 2.4.

Depending on the target domain different capability requirements exist, but for XLEN=32 there are insufficient bits available in the capability metadata to define one common format for all use cases. For example, a small embedded controller with limited address space of a few megabytes may need to make different tradeoffs than an auxiliary core within a larger SoC with many gigabytes of address space, and an accelerator using the RVY ISA may need yet another set of features. Therefore, multiple capability encodings with observable (but generally uninteresting) behavior differences need to be defined. While the in-memory bit pattern of capabilities across these systems as well as the precision of bounds may differ, software that does not depend on the raw bits in memory is source (and often also binary) compatible with these different capability encodings.

It is possible to generate code for RVY that is compatible with all valid in-memory encodings. However, to allow compilers and/or software library authors to make assumptions about encoding-dependent properties such as bounds precisions, these are encoded via an `LY<params>` suffix to the base ISA name such as:

- `RV32LYmw10rc1pc_IMAF_Zicsr`, which refers to the encoding format defined in RV32LYmw10rc1pc for RV32Y, or
- `RV64LYmw14rc1ps_IMAF_Zicsr`, which refers to the encoding format defined in RV64LYmw14rc1ps for RV64Y

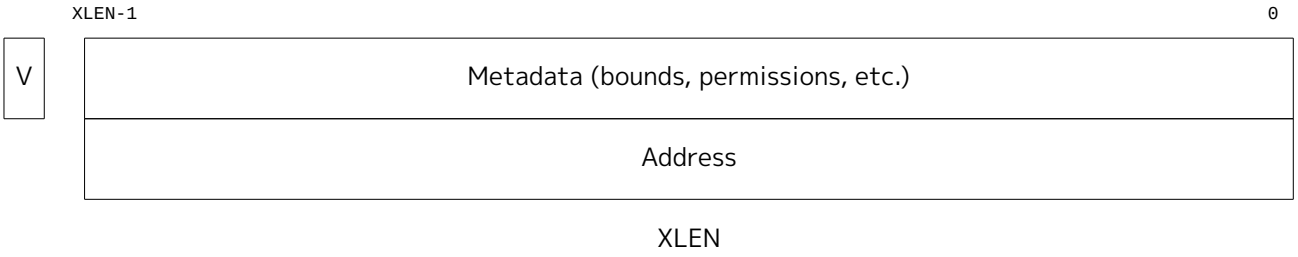


Figure 1. CHERI Capability structure

2.3.1. Address

The lower XLEN bits of a capability encode the address of where the capability points. This is also referred to as the integer part of the capability. For registers that are extended but currently hold non-capability data, all other fields are typically zero.

Future extensions may add `2*XLEN`-bit operations to make use of the wide registers for efficient handling of `2*XLEN`-bit non-capability data.

2.3.2. Capability Tag

The capability tag is an additional bit added to addressable memory and all YLEN-bit registers. It is stored

separately and may be referred to as *out of band* or *hidden*, and is *hardware managed*. It indicates whether a YLEN-bit register or YLEN-aligned memory location contains a valid capability. If the capability tag is set, the capability is valid and can be dereferenced (contingent on checks such as permissions or bounds).

All registers or memory locations able to hold a capability are YLEN bits wide with an additional hidden capability tag bit. These are referred to as being *YLEN-bit* in this specification.

The capability tag cannot be directly set by software; it is *not* a conventionally accessible bit of state. If the capability tag is set then it shows that the capability has been derived correctly according to the principles listed above (*provenance*, *integrity*, *monotonicity*). If the rules are followed then the capability tag will propagate through the instructions that modify, load or store the capability.

Therefore, whenever an instruction writes a capability with the capability tag set to a register:

- At least one capability tag must have been set in the input operands.
  - This is the *provenance* check.
- The requested operation must have been legal and does not increase bounds or permissions.
  - This is the *monotonicity* check.
- The instruction may have checked that none of the input capability were corrupted.
  - This is the *integrity* check. Only some instructions must perform this check, see Section 2.5.

Capability load/store require the *provenance* check:

- Any store that wrote the capability to memory was correctly authorized.
- Any load that read the capability from memory was correctly authorized.

When an operation fails a check, either due to software error or malicious intent, then the operation raises an exception or sets the resulting capability tag to zero.

For a capability to be valid, the capability tag must be set, otherwise a capability is invalid.

Using an invalid capability to dereference memory or authorize any operation raises an exception. All capabilities derived from invalid capabilities are themselves invalid, i.e., their capability tags are zero.

Writing non-capability data into a register or memory location causes the capability tag to be cleared. When the capability tag associated with the memory location is zero, the location contains non-capability data.

2.3.3. Capability tags in registers

Every YLEN-bit register has a one-bit capability tag, indicating whether the capability in the register is valid to be dereferenced. This capability tag is cleared whenever an invalid capability operation is performed. Examples of such invalid operations include writing only the integer portion (the address field) of the register or attempting to increase bounds or permissions.

2.3.4. Capability tags in memory

Capability tags are tracked through the memory subsystem: every aligned YLEN-bit wide region has a non-addressable one-bit capability tag, which the hardware manages atomically with the data. The capability tag is set to zero if any byte in the YLEN/8 aligned memory region is ever written using an operation other than a store of a capability operand. For example, the SY instruction may set the capability tag to one if all the instruction's prerequisites are met, but SW will always set the capability tag to zero.

All system memory and caches that store capabilities must preserve this abstraction, handling the capability tags atomically with the data.

2.3.5. Capability Bounds

Capabilities encode memory bounds, i.e., the lowest and highest byte in memory that it is permitted to access when dereferenced for data memory access, or for instruction execution.

Checking is on a byte-by-byte basis, so that it is possible for a memory access to be fully in-bounds, partially out-of-bounds or fully out-of-bounds.

It is not permitted to make any partially or fully out-of-bounds memory accesses.

Every capability has two memory address bounds: *base* representing the lowest accessible byte, and *top* representing one byte above the highest accessible byte.

- The *base* is XLEN bits and is *inclusive*.
- The *top* is (XLEN+1)-bits and is *exclusive*.

Inclusive top, with XLEN bits, was considered but rejected in favor of the exclusive top (with an additional bit to allow covering the entire address space). Using exclusive top allows representing a zero-length capability starting at address zero, which is not possible with inclusive top unless you introduce a special value for `-1`.

- The *length* is (XLEN+1)-bits and is defined to be *top - base*.

Therefore a memory location `A` in the range `base ≤ A < top` is within bounds, and so valid to access.

Checking every byte of every executed instruction and every byte of every data memory access is fundamental to the memory safety which CHERI provides. In a typical load/store unit, the expansion of the bounds from `rs1` *and bounds checking is in parallel with the address calculation, the memory translation and/or the PMA/PMP checking.*

A compressed format is used to encode the bounds with a scheme similar to floating-point using an exponent and a mantissa. Therefore small exponents can allow byte granularity on the bounds, but larger exponents give coarser granularity. One bounds encoding format based upon (Woodruff et al., 2019) is defined in Section 2.10.1.6.

Future CHERI bases may use encoding formats with different bounds encoding schemes (see Section 2.4).

Software can query the bounds of a capability held in a general-purpose register using the following instructions:

- The *base* is returned by the YBASER instruction.
- The *length* is returned by the YLENR instruction.
  - YLENR saturates the *length* to XLEN bits
- The *top* is returned by the YTOPR instruction (if Zytopr is implemented).
  - Otherwise the *top* can be calculated using a saturating addition of the YBASER and YLENR results.

2.3.6. Deriving New Bounds

On system initialization, one or more Root capabilities are available; typically, these have bounds which cover all of memory and the maximum permissions set. All smaller capabilities are derived from these.

The ISA does not allow the bounds (and permissions) of a capability with its capability tag set to be increased (*monotonicity*).

The YBNDSW, YBNDSWI and YBNDSRW instructions generate new capabilities from a source capability and a length operand (`rs2` or `imm`). The resulting capability has its address set as the *base* bound and its length as requested, subject to encoding granularity constraints. These instructions write the newly generated capability to the destination register. The granularity constraints mean that not all requested combinations of *top* and *base* bounds can be encoded exactly.

- YBNDSW sets the *base* to `rs1.address`, and the *length* to `rs2`. Set the capability tag to zero if the bounds cannot be encoded exactly.
- YBNDSWI sets the *base* to `rs1.address`, and the *length* to the immediate value. Set the capability tag to zero if the bounds cannot be encoded exactly.
- YBNDSRW sets the *base* to `rs1.address`, and the *length* to `rs2`. The bounds may be rounded up if they cannot be encoded exactly.
  - If YBNDSRW rounds up the requested bounds, they must still be no larger than the initial bounds.
- YAMASK can be used to calculate the nearest precisely encodable *length* and *base* values for a given size.

The bounds are encoded relative to the address field, sharing some upper bits of the address. The number of shared bits depends on the exponent, see Section 2.10.1.6.

2.3.7. Representability and Updating the Address

Software sometimes uses out-of-bounds pointers, for example, to represent a one-past-the end pointer to an array in C (`arr + size`). To support this pattern, CHERI must allow these out-of-bounds pointers to be held in capabilities while still protecting the bounds and integrity. Because the CHERI Concentrate (Woodruff et al., 2019) encoding scheme for memory bounds shares the upper bits of the address with the bounds, only a limited range of out-of-bounds pointers can be represented for any given set of bounds. This range is known as the **representable region**. The relationship between the bounds and the representable region is illustrated in Figure 2.

While the C standard only requires one-past-the-end pointers to be valid, real-world software has been observed to (temporarily) create pointers that point multiple bytes past the end or even before the start. To maximize software compatibility, the bounds representation was designed to allow for out-of-bounds pointers.

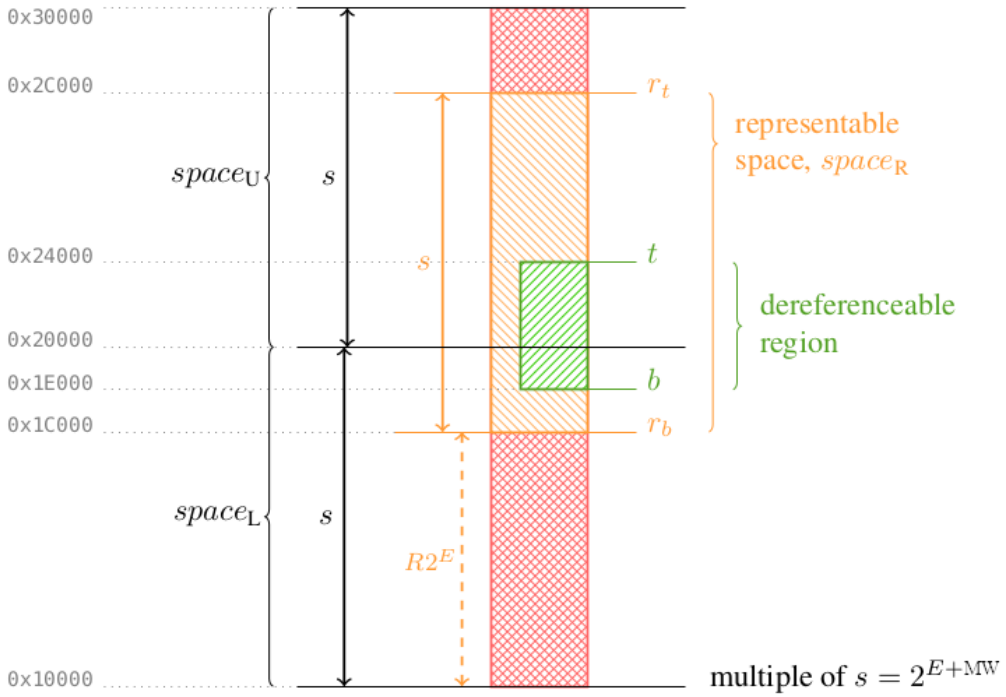


Figure 2. Memory address bounds and representable region encoded within a capability

E, MW and R in the figure are all introduced in Section 2.10.1.6.2 along with the bounds decoding.

The maximum range of address values that the pointer can take without changing the interpretation of bounds is defined by the representable region.

The historic, uncompressed CHERI-MIPS 256-bit capability encoding had separate 64-bit values for the base and top memory bounds, and another for metadata fields. This scheme could represent all out-of-bounds pointers with a very high hardware cost.

Since deriving a new capability with a different address could change the meaning of the bounds, all such derived capabilities (e.g., deriving the next pc capability from the existing pc via control-flow instructions or sequential execution) and instructions that return valid derived capabilities, must check that the new address is within the representable region defined by the source capability. If the interpretation of bounds *has* changed, then the capability tag of the derived capability is set to zero, so that it is invalid for use.

Software can derive a capability with a new address using instructions such as YADDRW, YADD and YADDI.

YADDRW writes back a derived capability with a new address field and, if the capability tag was previously set, sets the capability tag of the derived capability to one if the resulting capability still has the same bounds interpretation.

Existing software sometimes temporarily moves pointers outside of arrays, and then only comes back into the valid range on dereference, so the encoding was designed to allow valid capabilities to be out-of-bounds.

RVY implementations that use a different encoding scheme to RV32LYmw10rc1pc /RV64LYmw14rc1ps (e.g., for accelerators or specific micro-controllers) may specify an alternative to the representable region check, and may never allow the address to be out of bounds. Therefore, the rest of the specification uses the phrase **represented exactly** *for this check.*

2.3.8. Memory space

A hart supporting RVY 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 capability tag bit for each naturally aligned YLEN bits (e.g., 16 bytes in RV64), so that capabilities with their capability tag set can only be stored in naturally aligned addresses. Capability 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 2.10.1 is also circular, so address 0 is within the Representable Range of a capability where address 2^{XLEN} - 1 is within the bounds. However, the decoded top address of a capability is XLEN + 1 bits wide and does **not** wrap, so a capability with base 2^{XLEN} - 1 and top 2^{XLEN} + 1 is not a subset of the infinite capability and does not authorize access to the byte at address 0. Like malformed bounds (see Section 2.10.1.6.4), it is impossible for a CHERI core to generate a valid capability with top > 2^{XLEN}. 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.

2.3.9. Capability Type (CT)

This metadata value indicates the type of the capability and determines which operations the capability authorizes. Which capability types a given CHERI platform supports is a function of the extensions and capability encoding format in use. The capability encoding specifies a mapping between some bits within the capability format (usually described as "the CT field") and the capability types. The mapping must be able to encode type `0` but has few other requirements. It need not, for example, be interpreted as an (un)signed binary rendering of CT values. Standard capability type mapping is shown in Table 8.

Table 8. Capability type mapping in the RV32LYmw10rc1pc/RV64LYmw14rc1ps bases

Type Name	Integer Value	Description
Unsealed	0	Unsealed capability granting access to memory
Sentry	1	Immutable sealed entry point

The integer value in the table above defines the architectural capability type (i.e., the result of the YTYPER instruction) but does not need to be encoded that way in memory. Other base ISAs may provide different type mappings, but the value `0` must always represent an unsealed capability.

Unsealed capabilities

When `CT=0`, the capability authorizes access to a region of memory as defined by the permissions and bounds. All CHERI systems must support unsealed capabilities.

Sealed capabilities

Capabilities with `CT≠0` are sealed against modification and cannot be dereferenced to access memory. Instructions that operate on capabilities will produce a result with a cleared capability tag if the source capability is sealed and the operation would alter its address, bounds, or permissions. Extensions that augment capability metadata must describe their interaction(s) with sealed capabilities.

Given a capability with `CT=0`, deriving a capability with `CT≠0` is termed "sealing" (or "sealing with type `x`" when a particular output `CT=x` is meant). In the other direction, deriving a `CT=0` capability from a `CT≠0` capability is termed "unsealing" (or "unsealing from type `x`" when a particular input `CT=x` is meant). In general, each of these actions may require **authority** to operate at the non-zero type; extensions will specify how software expresses this authority for types not defined above.

Capability encodings may also make the set of CT-field values that may be used to seal a particular capability depend on the permissions granted by that capability. For example, it can be a useful space optimization to differentiate the CT-field values for capabilities granting X-permission from those not granting X-permission; the X-permission in the capability encoding effectively adds an additional bit to the CT-field field. (RV32LYmw1Orc1pc/RV64LYmw14rc1ps does not avail itself of this option, but, for example, the RV32LYenccheriot1 capability encoding does.)

Ambient sealing type

Some capability types are said to be "ambiently available" (or just "ambient") if they do not require specific authority to seal a capability (with that type). For example, if Zysentry is available on a given platform, the type with which it seals capabilities is considered ambiently available. (With the capabilities of RV32LYmw1Orc1pc/RV64LYmw14rc1ps, that would be type `1`.)

Sentry capability type

It is useful to have **immutable** function pointers within a CHERI software system. Sealed capabilities are a natural foundation, providing immutability. The JALR (RVY) instruction may unseal capabilities of particular, encoding-specified types before installing them into the pc. Capabilities sealed with such a type are dubbed "sentries" (a portmanteau of "sealed entries"). The JALR (RVY) instruction may also *seal* the return pointers it generates with encoding-specified types.

TODO: ARC COMMENT: "sentries" is a bad choice for a shortened form of "sealed entries" as the standard english or even CS connotation of sentry doesn't match with a sealed entry.

Sentry capabilities can establish a form of control-flow integrity between mutually distrusting code.

In addition to using sealed capabilities as sentries for secure entry points, sealed capabilities can also be useful to software as secure software tokens. YSUNSEAL can be used to convert such a token back to an unsealed capability. A future extension may add an unseal instruction for performance.

The set of RVY base ISAs do not make any use of non-zero `CT` values. The capability encoding format only needs to encode `CT` information if any extensions (such as Zysentry) are present which support non-zero `CT` values.

2.3.10. Architectural Permissions (AP)

This metadata field encodes architecturally defined permissions of the capability. Permissions grant access subject to the capability tag being set, the capability being unsealed, and bounds checks passing. Any 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 defined in RVY are listed in Table 9. All capability formats must support at least this set of permissions. Extensions building on top of RVY may define additional permissions: an example of such an extension is Zylevels1. The in-memory format of these permissions depends on the capability encoding.

Permissions can be cleared when deriving a new capability value (using YPERMC) but they can never be added.

Table 9. AP-field summary

Permission	Type	Comment
R-permission	Data memory permission	Authorize data memory read access

Permission	Type	Comment
W-permission	Data memory permission	Authorize data memory write access
X-permission	Instruction memory permission	Authorize instruction memory execute access
C-permission	Data memory permission	Authorize loading/storing of capability tags
LM-permission	Data memory permission	Used to restrict the permissions of loaded capabilities.
ASR-permission	Privileged state permission	Authorize privileged instructions and CSR accesses.

Read Permission (R)

Allow reading data from memory.

Write Permission (W)

Allow writing data to memory.

Execute Permission (X)

Allow instruction execution.

Capability Permission (C)

Allow reading capability tags from memory if R-permission is also granted.

Allow writing capability tags to memory if W-permission is also granted.

If C-permission is missing then the capability tags for capability loads and stores are read and written as zero.

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.

The permission stripping behavior *only* applies to loaded capabilities that have their capability tag set and are not sealed. This ensures that capability loads of non-capability data do not modify the loaded value, and that sealed capabilities are not modified.

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.

Access System Registers Permission (ASR, primarily used to authorize CSR accesses)

Allow read and write access to all privileged CSRs, some unprivileged CSRs and some privileged instructions. ASR-permission permission checks always use the permission in the pc.

In the unprivileged RVY ISA, the following are affected by ASR-permission:

- **Thread ID CSRs:** The utidc CSR requires ASR-permission for writing but not for reading.
- **Instructions:** CBO.INVAL (RVY) requires ASR-permission in the pc.

In the privileged RVY ISA, the following are affected by ASR-permission:

- **Instructions:** MRET (RVY), SRET (RVY), and DRET (RVY) require ASR-permission in the pc .
- **CSRs:** The privileged `stidc`, `vstidc`, and `mtidc` CSRs require ASR-permission for writing, but not for reading (if access is permitted in the current privilege mode). All other privileged CSRs require ASR-permission in the pc for any access. Access to debug CSRs requires ASR-permission.

This permission is important in privileged execution environments. Removing this permission allows constraining privileged software to a sandbox that cannot be subverted by changing privileged state.

Extensions may add additional non-privileged CSRs that require ASR-permission.

2.3.10.1. Permission Transitions

Not all capability permissions are *orthogonal* (that is, some permissions inherently *depend* on others). As such, using YPERMC to clear some permissions may have the effect of clearing others as well, such that a permission bit being set in the result of YPERMR implies that all bits for permissions upon which it depends will also be set.

For the base set of permissions just defined, the following rules apply. Extensions that define new permission bits may also introduce new dependency constraints. Currently defined examples are:

1. Zyhybrid, see Section 6.1.3
2. Zylevels1, see Section 9.4.2
3. Zyseal, see Table 46

Capability *encodings* may impose additional constraints to reduce the number of bits necessary to represent permissions.

Table 10. YPERMC base rules

YPERMC Rule	Permission	Valid only if
base-1	C-permission	R-permission or W-permission
base-2	LM-permission	C-permission and R-permission
base-3	ASR-permission	X-permission

When using RV64LYmw14rc1ps and RV32LYmw10rc1pc the complete set of rules with and without Zyhybrid and Zylevels1 are shown in Table 35 and Table 37.

2.3.11. Software-Defined Permissions (SDP)

The metadata also contains an encoding-dependent number of software-defined permission (SDP) bits. They can be inspected by the kernel or application programs to enforce restrictions on API calls (e.g., permit/deny system calls, memory allocation, etc.). They can be cleared by YPERMC but are not interpreted by the CPU otherwise.

While these bits are not used by the hardware as architectural permissions, modification follows the same rules: SDP bits can only be cleared and never set on valid capabilities.

This property is required to ensure restricted programs cannot forge capabilities that would pass the software-enforced checks.

Software is completely free to define the usage of these bits.

2.3.12. Special Capabilities

2.3.12.1. Root Capabilities

Root (sometimes also *primordial*, *initial*) capabilities are those provided by the system at reset. In some systems (and capability encodings), there is a single "Infinite" capability value, which grants all permissions and has bounds covering the whole 2^{XLEN} address space; in such systems, root capabilities are often Infinite. More generally, the *set* of root capabilities often collectively grant all permissions to all addresses. By way of example, an encoding may prohibit one capability from authorizing both write and execute; a system using such an encoding would typically make available maximally permissive read-write and read-execute capabilities as part of their root set.

How unprivileged software receives its root capabilities is largely an ABI question; the privileged specification will detail requirements of capability registers' reset state (and so on the root capabilities held therein).

Because particular sets of requirements recur throughout the specification, we define some useful shorthand terminology.

Root Executable Capability

An unsealed capability that has bounds covering all addresses and grants at least all of X-permission, R-permission, C-permission, LM-permission, and ASR-permission. Extensions introducing new permissions may require these to be provided by root executable capabilities.

Root Data Capability

An unsealed capability that has bounds covering all addresses and grants at least all of R-permission, W-permission, C-permission, and LM-permission. Extensions introducing new permissions may require these to be provided by root data capabilities.

2.3.12.2. NULL Capability

A capability with all-zero metadata, a zero capability tag, and an address of zero is referred to as the *NULL* capability. This capability grants no permissions and any dereference results in raising an exception.

2.4. CHERI encoding formats

CHERI implementations make trade-offs in their "encoding formats", such as the *precision* of bounds and the *expressible set* of combinations of permissions, that impact certain behaviors of the ISA. The CHERI base ISA name includes a compact description of these decisions, and the definition of all CHERI base ISAs includes sufficient details of a capability encoding format to precisely define the execution behavior of all base ISA instructions.

The names RV32Y and RV64Y refer generically to CHERI base ISAs, which share common semantics but have different metadata encodings. The RV32Y and RV64Y names on their own do not describe the complete base ISAs, but define all the *semantics* of a RVY machine, where certain behavioral parameters must be filled in by the capability encoding. For example, the YBNDSRW instruction yields a capability with new bounds that are rounded differently depending on the number of bits allocated to the bounds in the capability format. Systems with XLEN=64 can allocate more bits towards the mantissa of the bounds than XLEN=32 system and therefore have a larger Representable Range.

These observable differences in behaviour are similar to floating point numbers where the supported operations and their semantics are the same for all formats, but the exact result depends on the representation (e.g., 16/32/64-bit IEEE-754 or other floating point formats).

The encoding format parameters are specified in the architecture string using a `LY<params>` syntax, which can be used by software and/or the compiler to optimize for that specific in-memory format.

The base `L<encoding_parameters>` encoding format scheme is designed to separate the aspects of capability encoding that influence the behavior of RVY instructions from the other aspects defined by the encoding format. The intent is to simplify the future development of more (possibly domain-specific) encoding formats while also more directly conveying compatibility of architectural bases.

The following parameters are currently defined to distinguish existing formats:

Parameters for bounds precision and representability

These parameters affect the behavior of YBNDSW/YBNDSRW as well as the representability check that affects the tag-clearing behavior of how far you can go out of bounds with YADDRW/YADD.

- **Mantissa width (mw<N>)**: The mantissa width for the bounds encoding. For example, a format with 14 bits of mantissa, would use `mw14`. The mantissa width influences the precision of bounds (much like the mantissa width of floating point numbers defines arithmetic precision).
- **Maximum exponent (ea or e<N>)**: If *all* meaningful values of the exponent can be encoded directly in the exponent field this uses `ea`. Otherwise `e<N>` describes the largest encodable exponent for a valid capability except for the one used for the whole address space. For example, the RV32LYenccheriot1 format uses `e14` since it uses a four-bit exponent field that can encode only the values 0 through 14 and 24 (which, with its 9-bit mantissas, is sufficient to provide an encoding of bounds that cover the whole address space). The default value for this parameter is `ea` and may be omitted in that case.
- **Representable region (rc1 or r0)**: The number of mantissa bits used to guarantee representability of out-of-bounds capability values. So far two options have been used in commercial implementations: `rc1`, where the encoding uses one additional bit and uses a *centered* out-of-bounds region, and `r0` where zero bits are used for out-of-bounds capabilities and there is no guaranteed out-of-bounds range.

Parameter for permissions

This parameter defines which permission modifications using YPERMC also clear additional (non-encodable) permissions.

- **Permission encoding (ps or pc)**: Whether permissions are encoded using a *simple* representation of one bit per architectural permission (`ps`) or a *compressed* encoding that leverages dependencies between permissions and does not allow encoding all possible combinations (`pc`).

Parameter for in-memory format

- **In-memory encoding (enc<suffix>)**: This parameter defines layout of capability metadata fields in memory. The default value for this parameter is `enc1` and may be omitted in that case. Custom formats may define alternate layouts where all other parameters are otherwise identical, so this parameter can be used to differentiate such formats. Additionally, there is one other value, `enc0`, which means the compiler and/or software must avoid making any assumptions about the in-memory representation.

In general, only very specialized software such as offline crash-dump debuggers need to know the exact memory representation, and all other software can and should use the inspection instructions such as YBASER, etc.

Other parameters

These other parameters are not used by the formats currently set forward for ratification but are being used in custom extensions on top of RVY.

- AUIPC (RVY) shift (as<N>): The shift amount used in the RVY variant of AUIPC (RVY). This is 12 (as in RVI) unless specified otherwise.

As with other extensions, a version number suffix may be added to the `LY` string to identify a new version with the same parameters but different in-memory representation. Any newer version must be forwards compatible with any existing software using the same parameters. Custom capability encoding formats may be implemented (their name suffix should start with `LYX`).

Any future capability encoding formats must provide all the capability guarantees listed in this chapter and must undergo formal verification of these properties before being proposed for ratification.

The behavior of all RVY instructions is fully defined by the parameters in the `LY` string. Therefore, any implementation with matching parameters is binary compatible (assuming the same extensions are implemented).

The currently known RVY base capability encoding formats are listed below:

- For RV64Y, the capability encoding format based on University of Cambridge CHERI v9 (Watson et al., 2023): RV64LYmw14rc1ps
- For RV32Y, the capability encoding format based on University of Cambridge CHERI v9 (Watson et al., 2023): RV32LYmw10rc1pc

It is expected that the actual RVY instantiation is chosen using a profile rather than a full architecture string and therefore the unwieldy suffix should rarely be user-visible.

For RV64, the number of spare bits in the capability encoding ensure sufficient future extensibility and therefore no custom formats exist.

However, RV32Y has insufficient bits available to enable features for all target domains, so in addition to the RV32LYmw10rc1pc further RV32Y-extending capability formats exist:

- The CHERIoT encodings (with `LYmw9e14r0as11pc` parameters) optimized for smaller devices: RV32LYenccheriot1, RV32LYenccheriot2, and RV32LYenccheriot3.

RV32YE (16-register) implementations are recommended to use one of these CHERIoT encodings.

2.5. Integrity of Capabilities

CHERI enforces the following rules (integrity checks) for all valid capabilities :

1. The bounds are not malformed.
2. The capability metadata does not have any bits, or fields set to a reserved value.
    a. An example of a reserved field value is permissions which cannot be legally produced by YPERMC.

If any of the above rules do not hold for a capability with the capability tag set, it indicates either:

- State corruption due to memory or logic faults, or
- The presence of incompatible or faulty CHERI IP within the system.

These checks are much less rigorous than parity or ECC protection, and are only used to detect simple problems with the capability metadata.

Performing these checks is optional except for instructions that are defined to accept arbitrary input bit patterns and can write an output with the capability tag set. All instructions that require integrity checks have this noted in the instruction description.

Currently YBLD is the only defined instruction that sets the `capability tag` *on an arbitrary bit pattern and therefore must perform an integrity check on the input capability in* `rs2`*. For all other instructions, performing these checks is optional unless the hart implements additional reliability features (since malformed capabilities with the capability tag set cannot exist under normal operating conditions).*

Even though valid capabilities which fail the integrity check could not have been legally generated by the local hart, defining the handling in the architecture allows the behavior to be precisely specified for all $2^{(YLEN+1)}$ *input values.*

2.6. Extended State

As stated above, all state which can hold addresses are extended from XLEN to YLEN bits.

2.6.1. General Purpose Registers

The XLEN-wide integer registers (e.g., `sp`, `a0`) are all extended to YLEN bits and associated capability tags, as shown in Figure 3.

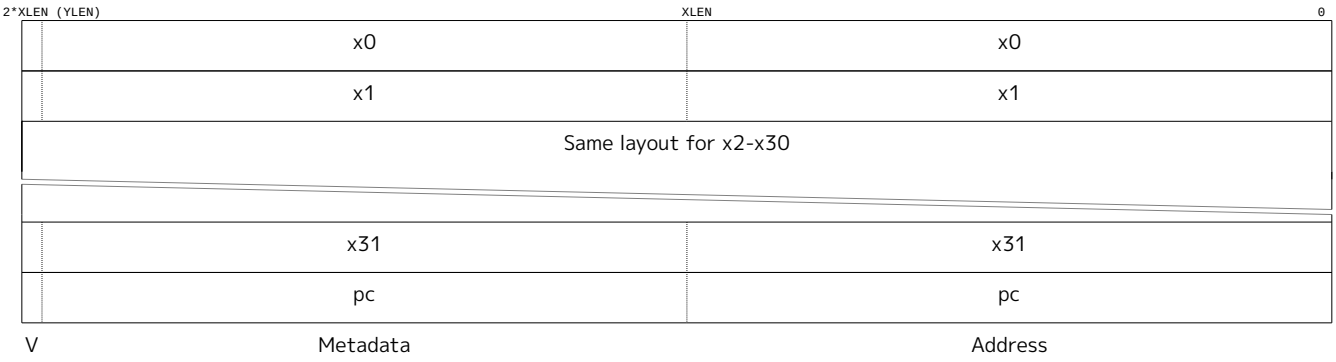


Figure 3. Extended registers in RVY

The zero register is extended with zero metadata and a zero capability tag: this is called the NULL capability.

2.6.2. The Program Counter Capability (`pc`)

The `pc` is extended to be a capability. Extending the `pc` allows the range of branches, jumps and linear execution for currently executing code to be restricted. The pc address field is the `pc` in the base RISC-V ISA so that the hardware automatically updates it as instructions are executed.

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

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

- All integrity checks must have passed.

On system initialization the `pc` bounds and permissions must be set such that the program can run successfully (e.g., by setting it to a Root Executable capability to ensure *all* instructions are in bounds).

Future ISA extensions should respect these rules so that the checked bits do not need to be stored in all copies of the pc in the implementation.

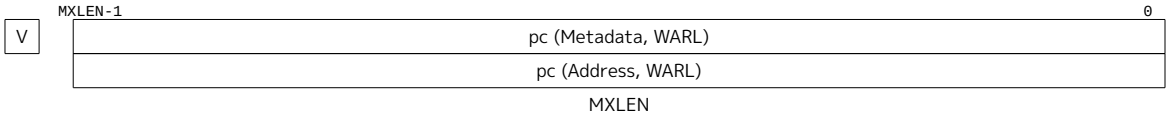


Figure 4. Program Counter Capability

2.6.3. Added CSRs

RVY adds the YLEN-bit CSR shown in Table 11.

Table 11. Capability CSRs added in RVY

YLEN CSR	Permissions	Description
utidc	RW, ASR-permission required for writes, not reads	User Thread ID Capability

2.6.3.1. User Thread Identifier Capability (utidc)

The utidc register is an unprivileged CSR used to identify the current software thread. Any operation that modifies utidc raises an exception unless the ASR-permission is set in the current pc.

While the RISC-V ABI includes a thread pointer (tp) register, it is not usable for the purpose of reliably identifying the current software thread because the tp register is a general purpose register and can be changed arbitrarily by untrusted code. Therefore, this specification offers an additional CSR that facilitates a trusted source for identifying software threads.

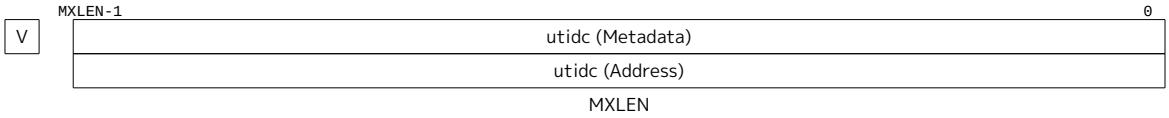


Figure 5. User thread identifier capability register

TODO: ARC COMMENT: This description of compartments is confusing and seems to mix kernel mode thread switching and user mode compartment switching? Not clear why utidc is needed.

The following should probably move to a programmers guide

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. The utidc register supports a model where untrusted code is separated by trusted code and each call from one piece of untrusted code to another piece of untrusted code goes through trusted code. Often, the trusted code is referred to as a **trampoline**. *Sentries can be called from different software threads and thus there needs to be a way of identifying the current software thread. While identifying the current software 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 utidc register is designed to hold a capability, which can only be used for memory accesses by trusted code. In a commonly used model on CHERI systems, the trusted code's responsibility is only to switch between compartments, but not to switch threads. This responsibility is usually taken over by more privileged code, e.g., an operating system kernel running on a different privilege level. The privileged code switches software threads and writes the utidc register.*
>
> *Every piece of code in the user space (and more privileged levels) can read the contents of the utidc register. However, the memory authorized by the capability in utidc must not be accessible to untrusted code, but only to trusted code. In order to protect this capability, it can be sealed. The trusted code will be given means to unseal this capability (say, via YSUNSEAL). For the untrusted code, the memory pointed to by utidc is inaccessible. The sealed capability itself is no secret, but the memory to which it points is a secret and must not be accessed by any untrusted code.*
>
> *Trusted code can use utidc to implement secure compartment switches. Often, the capability therein is used to implement a trusted stack. Whenever a compartment switch happens, the trusted code can pass arguments between the caller and callee compartment avoid capability leaks between the two compartments. The trusted code can store capabilities on the trusted stack when calling out of a compartment and can install them when returning to the same compartment.*

2.6.4. Extended CSRs

All CSRs that can hold addresses are extended to YLEN bits.

RVY has three classes of CSRs

- XLEN-bit CSRs, which do not contain addresses
  - e.g., *fcsr* from the "F" extension
- XLEN-bit CSRs extended to YLEN bits, which are able to contain addresses (referred to as *extended CSRs*)
  - e.g., *jvt* from the "Zcmt" extension
- YLEN-bit CSRs, which are added by RVY and contain addresses
  - e.g., utidc

When accessing CSRs these rules are followed:

1. Accesses to XLEN-bit CSRs are as specified by Zicsr
2. Accesses to YLEN-bit CSRs and extended CSRs, using CSRRW will:
   a. Read YLEN bits
   b. Write YLEN bits, and will write the capability tag to zero if any integrity check fails
3. Accesses to YLEN-bit CSRs and extended CSRs, using instructions other than CSRRW will:
   a. Read YLEN bits
   b. Write an XLEN-bit value to the address field, and use the semantics of the YADDRW instruction to determine the final written value

> *Any YLEN-bit or extended CSR may have additional rules defined to determine the final written value of the metadata and/or to write zero to the capability tag.*

The assembler pseudoinstruction to read a capability CSR `csrr rd, csr`, is encoded as `csrrs rd, csr, x0`.

Table 12. YLEN-bit CSR and Extended CSR access summary for RVY

Instruction	Read Width	Write Width
CSRRW rd==x0		YLEN
CSRRW rd!=x0	YLEN	YLEN
CSRR[C|S] rs1==x0	YLEN	
CSRR[C|S] rs1!=x0	YLEN	XLEN
CSRRWI rd==x0		XLEN
CSRRWI rd!=x0	YLEN	XLEN
CSRR[C|S]I uimm==x0	YLEN	
CSRR[C|S]I uimm!=x0	YLEN	XLEN

In Table 12, when there is no read or write width shown, the CSR access is *not* made and there are no side-effects following standard Zicsr rules.

2.7. Capability checks

With RVY, every memory access performed by a CHERI core must be authorized by a capability.

Instruction fetches and data memory accesses may result in a fatal exception if the access is out of bounds, or if the authorizing capability is missing the required permissions. I.e.,:

- all load instructions requires R-permission
- all store instructions require W-permission
- all indirect jumps require X-permission on the target capability

Instruction fetch is also authorized by a capability: the program counter capability (pc) which extends the PC.

This allows code fetch to be bounded, preventing a wide range of attacks that subvert control flow with non-capability data.

The authorizing capability is either named explicitly (the base register of a load/store operation) or implicitly (when executing a branch, pc is used for authorization).

E.g., `lw t0, 16(sp)` loads a word from memory, getting the address, bounds, and permissions from the `sp` (capability stack pointer) register.

No other exception paths are added by RVY: in particular, capability manipulations do not raise an exception, but may set capability tag of the resulting capability to zero if the operation is not permitted.

2.8. Added Instructions

RVY adds new instructions to operate on capabilities.

2.8.1. Instructions to Update The Capability Pointer

Creating a new capability with a different address (i.e., updating the pointer) requires specific instructions instead of integer ADD/ADDI. These instructions all include a check that the resulting address can be represented exactly within the new capability.

Table 13. Instructions which update the address field summary in RVY

Mnemonic	Description
YADDI	Capability pointer increment by immediate
YADD	Capability pointer increment
YADDRW	Write capability address

2.8.1.1. YADDI

See YADD.

2.8.1.2. YADD

Synopsis

Capability pointer increment

Mnemonic

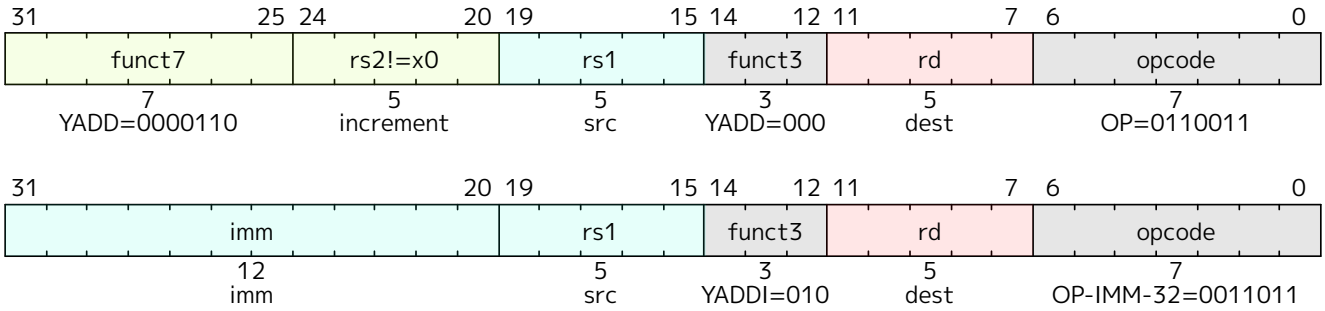
```
yadd rd, rs1, rs2
yaddi rd, rs1, imm
```

Suggested assembly syntax

```
add rd, rs1, rs2
addi rd, rs1, imm
```

The suggested assembly syntax distinguishes from integer `add`/`addi` *by operand type.*

Encoding



ARC note - the proposed YADDI encoding has since been allocated by the P extension proposal and will require a new final allocation.

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

Description

Copy the capability in register `rs1` to register `rd`.

For YADD, increment `rd.address` by the value in `rs2[XLEN-1:0]` .
For YADDI, increment `rd.address` by the immediate value `imm`.

Set `rd.tag=0` if `rs1` is sealed.

Set `rd.tag=0` if the resulting capability cannot be represented exactly.

Set `rd.tag=0` if `rs1` fails any integrity checks.

Included in

RVY

Operation for YADD

```
let cs1_val = C(cs1);
```

```
let rs2_val = X(rs2);

let newCap = incCapAddrChecked(cs1_val, rs2_val);

C(cd) = newCap;
RETIRE_SUCCESS
```

Operation for YADDI

```
let cs1_val = C(cs1);
let immBits : xlenbits = sign_extend(imm);

let newCap = incCapAddrChecked(cs1_val, immBits);

C(cd) = newCap;
RETIRE_SUCCESS
```

2.8.1.3. YADDRW

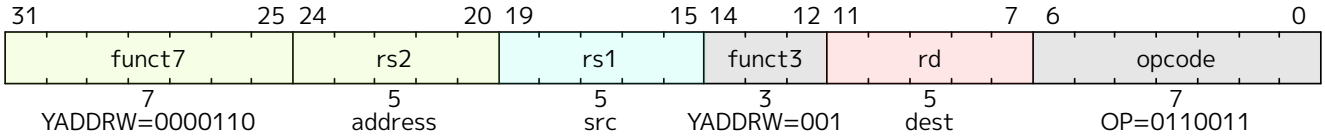
Synopsis

Write capability address

Mnemonic

```
yaddrw rd, rs1, rs2
```

Encoding



Description

Copy the capability `rs1` to `rd`.

Set `rd.address` to `rs2[XLEN-1:0]`.

Set `rd.tag=0` if `rs1` is sealed.

Set `rd.tag=0` if the resulting capability cannot be represented exactly.

Set `rd.tag=0` if `rs1` fails any integrity checks.

Included in

RVY

Operation

```
C(cd) = setCapAddrChecked(C(cs1), X(rs2));
RETIRE_SUCCESS
```

2.8.2. Instructions to Manipulate Capabilities

For security, capabilities can only be modified in restricted ways. Special instructions are provided to copy capabilities or derive a new capability using manipulations such as *shrinking* the bounds (YBNDSW), *reducing* the permissions (YPERMC) or *authorizing* a capability with another one which has a superset (or identical) bounds and permissions (YBLD).

Table 14. Summary of RVY instructions that create a modified capability

Mnemonic	Description
YPERMC	Clear capability permissions
YMV	Capability register copy
YHIW[1]	Write capability metadata and clear capability tag (pseudo)
YBNDSWI	Write capability bounds by immediate
YBNDSW	Write capability bounds
YBNDSRW	Write capability bounds, rounding up if required
YSUNSEAL	Unseal by superset reconstruction

[1] YHIW is a pseudoinstruction for PACKY

2.8.2.1. YPERMC

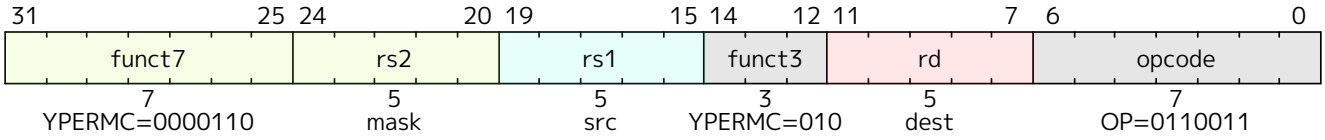
Synopsis

Clear capability permissions

Mnemonics

```
ypermc rd, rs1, rs2
```

Encoding



Description

YPERMC performs the following operations:

1. Convert the AP-field, SDP-field, and any other extension-defined permission-like fields of capability `rs1` into a bit field with the format shown in Figure 6.
2. The initial value in register `rs2[XLEN-1:0]` is treated as a bit mask that specifies bit positions to be cleared in the bit field. Any bit that is high in `rs2` will cause the corresponding bit to be cleared in the bit field.

> *Future extensions may include hardwired permission bits, in which case they are not cleared by set bits in* `rs2`.

3. Encode the resulting architectural permissions as specified by the encoding in use. This may involve iteratively applying the rules in Section 2.3.10.1, as well as any rules added by extensions or the capability encoding, until a fixed point is reached, arriving at a set of permissions not further reduced by any rule. Any permission-dependent capability metadata is updated to reflect any permissions removed by this procedure. Note that encodings may specify procedures for computing the new permission set; while these must be idempotent and respect the rules of Section 2.3.10.1, they need not be of the form "iterate to fixed point".

> *Depending on the base ISA and supported extensions, some combinations of permissions cannot be encoded or are not useful. In these cases, YPERMC will return a minimal sets of permissions, which may be no permissions. Therefore, it is possible that requesting to clear a permission also clears others, but YPERMC will never add new permissions.*

4. Copy `rs1` to `rd`, and update the AP-field, SDP-field, and any others therein with the newly calculated versions.
5. Set `rd.tag=0` if `rs1` is sealed and any bits in the AP-field or SDP-field were affected by YPERMC; extensions must describe whether such changes to their bits also necessitate capability tag clearing.
6. Set `rd.tag=0` if any integrity checks fail.

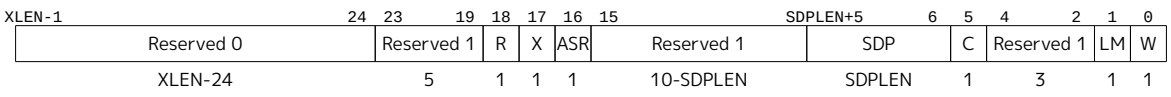


Figure 6. Capability permissions bit field

> *If a future extension adds a new permission that overlaps with an existing permission (e.g., finer-grained ASR-permission), then clearing the original must also clear the new permission.*

This ensures software forward-compatibility: for example, a kernel that does not know about finer-grained ASR-permission subsets must still be able to prevent all access to privileged instructions and CSRs by clearing ASR-permission.

Any future extension that defines new permissions that are a refinement of existing permissions (e.g., finer-grained ASR-permission or those of Zylevels1) must be allocated to the bits that are currently reported as 1 to ensure forward-compatibility. Completely new permissions (e.g., sealing) should use the bits that are reported as zero in the current specification.

Extensions like Zylevels1 introduce bits that are, conceptually, labels on a capability rather than a permission granted by the capability. These bits are, nevertheless, still adjusted using the YPERMC instruction. This avoids the need for a dedicated instruction and allows simultaneous changes of these labels and permissions.

Included in

RVY

Operation

TODO

2.8.2.2. YMV

Synopsis

Capability register copy

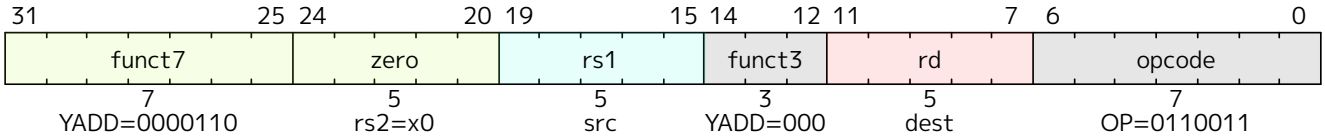
Mnemonic

```
ymv rd, rs1
```

Suggested assembly syntax

```
ymv rd, rs1
```

Encoding



YMV is encoded as YADD with `rs2=x0`.

Description

The contents of capability register `rs1` are written to capability register `rd`. YMV unconditionally does a bit-wise copy from `rs1` to `rd` .

This instruction can propagate valid capabilities which fail integrity checks.

Included in

RVY

Operation

```
C(cd) = C(cs1);
RETIRE_SUCCESS
```

2.8.2.3. PACKY

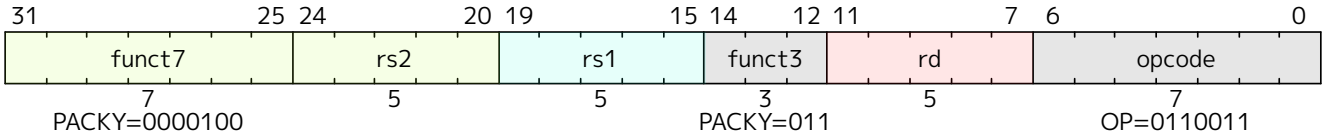
Synopsis

Pack Y register

Mnemonic

```
packy rd, rs1, rs2
```

Encoding



Description

The PACKY instruction packs the least-significant XLEN-bits of `rs1` and `rs2` into `rd`, and sets `rd.tag=0`.

Included in

RVY

2.8.2.4. YHIW

Synopsis

Capability set metadata

Mnemonic

```
yhiw rd, rs1, rs2
```

Encoding

YHIW is a pseudoinstruction for PACKY

Description

Copy `rs1` to `rd`.

Replace the capability metadata of `rs1` (i.e., bits [YLEN-1:XLEN]) with `rs2` and set `rd.tag` to 0.

The value of `rs1.tag` *does not affect the result.*

Included in

RVY

Operation

```
let capVal = C(cs1);
let intVal = X(rs2);
let newCap = bitsToCap(false, intVal @ capVal.address);
C(cd) = newCap;
RETIRE_SUCCESS
```

2.8.2.5. YBNDSWI

See YBNDSW.

2.8.2.6. YBNDSW

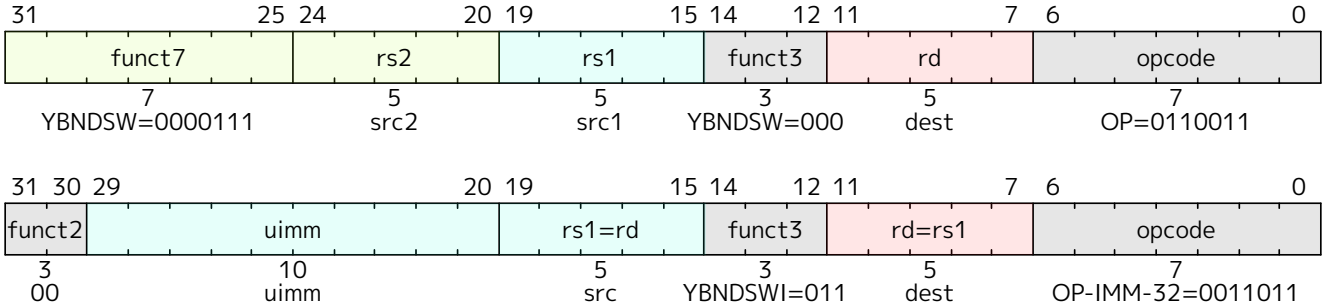
Synopsis

Write capability bounds

Mnemonics

```
ybndsw rd, rs1, rs2
ybndswi rd, rs1, imm
```

Encoding



99.5% of uses of unrestricted allocation of rs1 and rd for YBNDSWI have rs1=rd, and so the cases where they do not match are reserved.

ARC note - This is a placeholder encoding pending final allocation. We want to ensure that RVY minimized usage of opcode space, so a 10-bit immediate is sufficient and rs1!=rd does not need to be supported, and therefore could use the same number of bits as a R-type instruction.

Description

Copy the capability from register `rs1` to register `rd`. Set the base address of its bounds to the value of `rs1.address` and set the length of its bounds to `rs2[XLEN-1:0]` for YBNDSW, or `imm` for YBNDSWI.

Set `rd.tag=0` if `rs1.tag=0`, `rs1` is sealed or if `rd` 's bounds exceed `rs1` 's bounds.

Set `rd.tag=0` if the requested bounds cannot be encoded exactly.

Set `rd.tag=0` if `rs1` fails any integrity checks.

YBNDSWI uses the following formula to determine the requested length:

- `((imm[7:0] + 257) << imm[9:8]) - 256`

This formula does not actually require two additions in decode as it expands to the following:

`imm[9:8]`	*Resulting immediate*	*Immediate range*
0	`(imm[7:0]<<0) + 1`	*1, 2, ..., 255, 256*
1	`(imm[7:0]<<1) + 258`	*258, 260, ..., 766, 768*
2	`(imm[7:0]<<2) + 772`	*772, 776, ..., 1788, 1792*
3	`(imm[7:0]<<3) + 1800`	*1800, 1808, ..., 3832, 3840*

Included in

RVY

Operation for YBNDSW

```
let cs1_val = C(cs1);
let length = X(rs2);
let newBase = cs1_val.address;
let newTop : CapLenBits = zero_extend(newBase) + zero_extend(length);
// inCapBoundsNoWrap returns false if the input bounds are malformed.
let inBounds = inCapBoundsNoWrap(cs1_val, newBase, unsigned(length));
let (exact, newCap) : (bool, Capability) = setCapBounds(cs1_val, newBase,
newTop);
let cond = not(inBounds & exact) |
           boundsMalformed(newCap) |
           not(capReservedValid(newCap)) |
           capIsSealed(newCap);
C(cd) = clearTagIf(newCap, cond);
RETIRE_SUCCESS
```

Operation for YBNDSWI

TODO

2.8.2.7. YBNDSRW

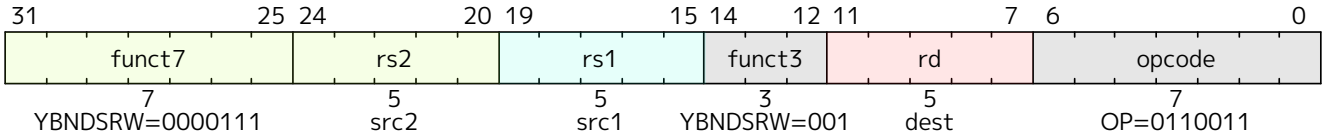
Synopsis

Write capability bounds, rounding up if required

Mnemonic

```
ybndsrw rd, rs1, rs2
```

Encoding



Description

Copy the capability from register `rs1` to register `rd`. Set the base address of its bounds to the value of `rs1.address` and set the length of its bounds to `rs2[XLEN-1:0]`.

The base is rounded down and the top is rounded up by the smallest amounts needed to form a capability covering the requested base and top.

Set `rd.tag=0` if `rs1.tag=0`, `rs1` is sealed or if `rd` 's bounds exceed `rs1` 's bounds.

Set `rd.tag=0` if `rs1` fails any integrity checks.

Included in

RVY

Operation

```
let cs1_val = C(cs1);
let length = X(rs2);
let newBase = cs1_val.address;
let newTop : CapLenBits = zero_extend(newBase) + zero_extend(length);
// inCapBoundsNoWrap returns false if the input bounds are malformed.
let inBounds = inCapBoundsNoWrap(cs1_val, newBase, unsigned(length));
let (_, newCap) : (bool, Capability) = setCapBounds(cs1_val, newBase,
newTop);
let cond = not(inBounds) |
           boundsMalformed(newCap) |
           not(capReservedValid(newCap)) |
           capIsSealed(newCap);
C(cd) = clearTagIf(newCap, cond);
RETIRE_SUCCESS
```

2.8.2.8. YSUNSEAL

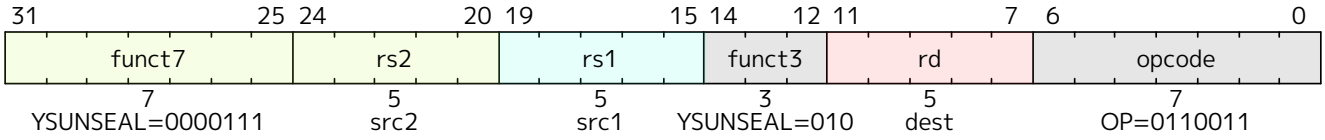
Synopsis

Unseal by superset reconstruction

Mnemonic

```
ysunseal rd, rs1, rs2
```

Encoding



Description

Copy `rs2` to `rd`.

Set `rd.ct=0`. (That is, unseal `rd`.)

Set `rd.tag=1` if:

1. `rs1.tag=1`, and
2. `rs1` passes all integrity checks, and
3. `rs1` is not sealed (that is, `rs1` has zero CT-field), and
4. `rs2.tag` is set, and
5. `rs2` passes all integrity checks, and
6. `rs2` is sealed (that is, `rs2` has non-zero CT-field), and
7. `rs2` 's permissions and bounds are equal to or a subset of `rs1` 's, and
8. any extension-specific constraints on YSUNSEAL hold.

Otherwise, set `rd.tag=0`

When `rs1` is `x0` YSUNSEAL will copy `rs2` to `rd` and clear `rd.tag` and `rd.ct`. However future extensions may add additional behavior to update currently reserved fields, and so software should not assume `rs1==0` to be a pseudo-instruction for capability tag and type clearing.

YSUNSEAL is intended to enable "superset unsealing" of opaque handles to software objects. Specifically, a software component can:

1. *allocate the memory for these objects from a region of address space,*
2. *render capabilities to these objects opaque by sealing (with, for example, the YSENTRY instruction, if present),*
3. *distribute these handles to other software components, and*
4. *later use its (unsealed) capability to the backing region as the authority (in `rs1`) of a YSUNSEAL instruction to recover an unsealed capability to the object backing a handle (in `rs2`) received from other components.*

The result of YSUNSEAL will be untagged if the received capability is not a handle to an object in the recipient's address space. This makes it easy for recipient software to ensure that received capabilities actually are handles to the recipient's objects.

While YSUNSEAL requires that the capability in its `rs2` is sealed, it imposes no requirements on which non-zero CT-field value has been used to seal `rs2`. If the capability encoding defines multiple non-zero CT-field values and software wishes to distinguish between them, it must use YTYPER on the sealed capability.

Included in

RVY

Operation

2.8.3. Instructions to Decode Capability Bounds

The *bounds* describing the range of addresses the capability gives access to are stored in a compressed format. These instructions query the bounds and related information.

Table 15. Instructions which decode capability bounds summary in RVY

Mnemonic	Description
YBASER	Read capability base address
YLENR	Read capability length

2.8.3.1. YBASER

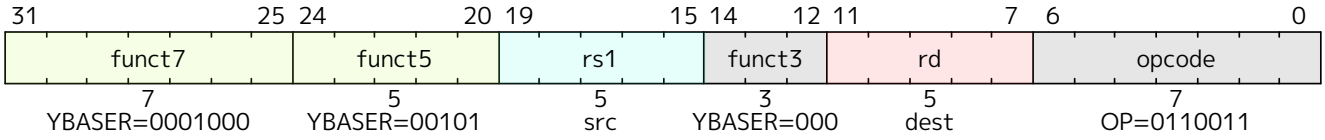
Synopsis

Read capability base address

Mnemonic

```
ybaser rd, rs1
```

Encoding



Description

Decode the base integer address from `rs1` 's bounds and write the result to `rd`.

If `rs1` 's bounds can't be decoded, or `rs1` fails any integrity checks, then return zero.

> *The value of* `rs1.tag` *does not affect the result.*

Included in

RVY

Operation

```
let capVal = C(cs1);
X(rd) = match getCapBoundsBits(capVal) {
  None() => zeros(),
  Some(base, _) => base
};
RETIRE_SUCCESS
```

2.8.3.2. YLENR

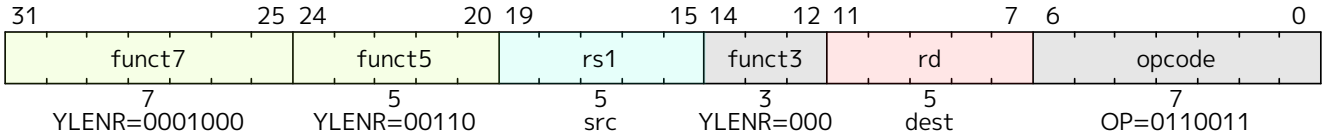
Synopsis

Read capability length

Mnemonic

```
ylenr rd, rs1
```

Encoding



Description

Calculate the length of `rs1` '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`.

Return the maximum length, 2^{MXLEN}-1, if the length of `rs1` is 2^{MXLEN}.

If `rs1` 's bounds can't be decoded, or `rs1` fails any integrity checks, then return zero.

> *The value of* `rs1.tag` *does not affect the result.*

Included in

RVY

Operation

```
let capVal = C(cs1);
// getCapLength returns 0 if the bounds are malformed
let len = getCapLength(capVal);
X(rd) = to_bits(xlen, if len > cap_max_addr then cap_max_addr else len);
RETIRE_SUCCESS
```

2.8.4. Instructions to Extract Capability Fields

These instructions either directly read bit fields from the metadata or capability tag, or only apply simple transformations on the metadata.

Table 16. Instructions which extract capability fields summary in RVY

Mnemonic	Description
YTAGR	Read capability tag
YPERMR	Read capability permissions
YTYPER	Read capability type
YHIR[1]	Read capability metadata (pseudo)

[1] YHIR is a pseudoinstruction for SRLIY

2.8.4.1. YTAGR

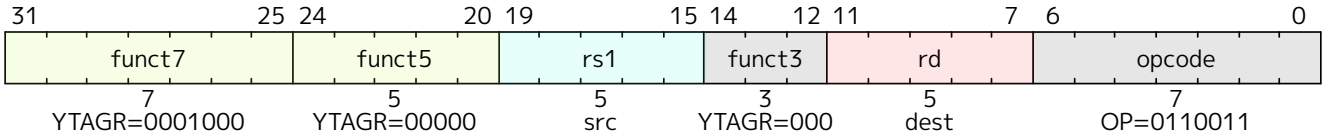
Synopsis

Read capability tag

Mnemonic

```
ytagr rd, rs1
```

Encoding



Description

Zero extend the value of `rs1.tag` and write the result to `rd`.

Included in

RVY

Operation

```
let capVal = C(cs1);
X(rd) = zero_extend(bool_to_bits(capVal.tag));
RETIRE_SUCCESS
```

2.8.4.2. YPERMR

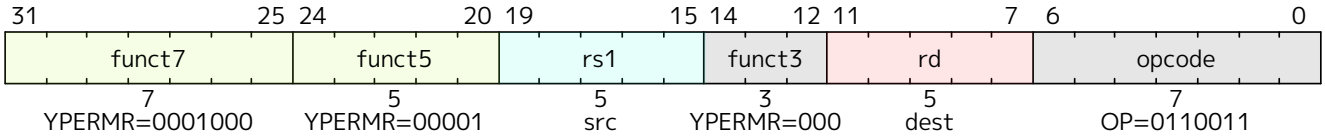
Synopsis

Read capability permissions

Mnemonic

```
ypermr rd, rs1
```

Encoding



Description

Convert the unpacked AP-field, SDP-field, and any other extension-defined permission-like fields of capability `rs1` into a bit field, with the same format as used by YPERMC (see Figure 6), and write the result to `rd`.

All bits in the `[23:0]` range that are reserved or assigned to extensions that are not implemented by the current hart always report 1.

All architectural permission bits in `rd` are set to 0 if any integrity checks failed.

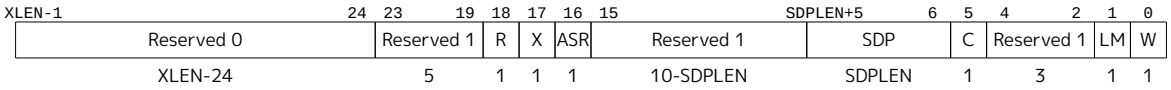


Figure 7. Capability permissions bit field

The value of `rs1.tag` *does not affect the result.*

Included in

RVY

Operation

```
let capVal = C(cs1);
X(rd) = packPerms(getArchPermsLegalized(capVal), capVal.sd_perms).bits;
RETIRE_SUCCESS
```

2.8.4.3. YTYPER

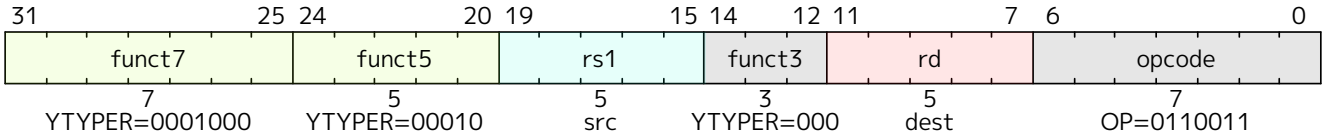
Synopsis

Read capability type

Mnemonic

```
ytyper rd, rs1
```

Encoding



Description

Decode the architectural capability type (CT-field) from `rs1` and write the result to `rd`.

> *The value of* `rs1.tag` *does not affect the result.*

Included in

RVY

Operation

```
let capVal = C(cs1);
X(rd) = zero_extend(bool_to_bits(capVal.sealed));
RETIRE_SUCCESS
```

2.8.4.4. SRLIY

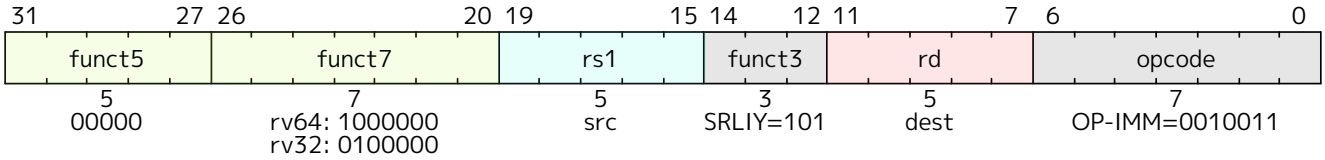
Synopsis

Logical right shift of Y register

Mnemonic

```
srliy rd, rs1, shamt
```

Encoding



Description

Logical right shift of Y register `rs1` to `rd`, zero-filling the upper bits of the result.

Currently the only valid shift distance is XLEN places, a future extension may add an arbitrary shift distance.

For RV32Y, `srliy rd, rs1, 32` *has an identical encoding and operation to the RV64 instruction* `srli rd, rs1, 32`.

Included in

RVY

Operation

TODO

2.8.4.5. YHIR

Synopsis

Read capability metadata (pseudo)

Mnemonic

`yhir rd, rs1`

Encoding

`yhir rd, rs1` is a pseudoinstruction for `srliy rd, rs1, XLEN`

Description

Copy the metadata (bits [YLEN-1:XLEN]) of capability `rs1` into `rd`.

> *The value of* `rs1.tag` *does not affect the result.*

Included in

RVY

Operation

```
let capVal = C(cs1);
X(rd) = capToMetadataBits(capVal).bits;
RETIRE_SUCCESS
```

2.8.5. Miscellaneous Instructions to Handle Capability Data

Table 17. Miscellaneous capability instruction summary in RVY

Mnemonic	Description
SYEQ	Capability equality comparison including capability tag
YLT	Capability less than comparison including capability tag
YAMASK	Capability alignment mask

2.8.5.1. SYEQ

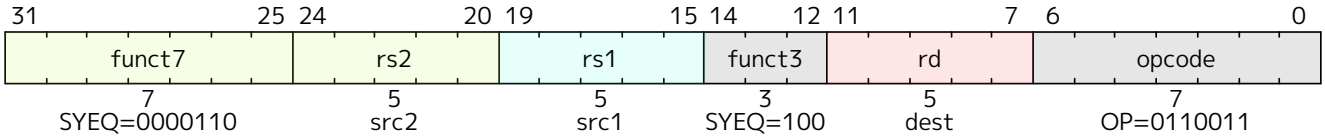
Synopsis

Capability equality comparison including capability tag

Mnemonic

```
syeq rd, rs1, rs2
```

Encoding



Description

Set `rd` to 1 if all bits (i.e., YLEN bits and the capability tag) of capabilities `rs1` and `rs2` are equal, otherwise set `rd` to 0.

Included in

RVY

Operation

```
let cs1_val = C(cs1);
let cs2_val = C(cs2);
X(rd) = zero_extend(bool_to_bits(cs1_val == cs2_val));
RETIRE_SUCCESS
```

2.8.5.2. YLT

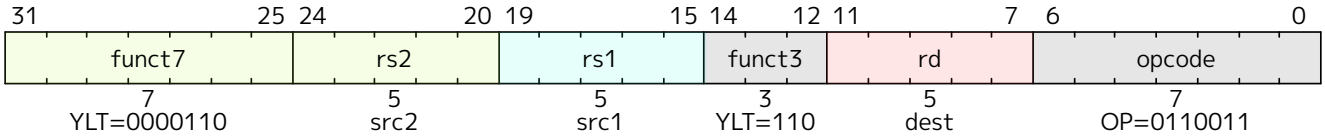
Synopsis

Set Capability Subset

Mnemonic

```
ylt rd, rs1, rs2
```

Encoding



Description

`rd` is set to 1 if:

1. the capability tag of capabilities `rs1` and `rs2` are equal, and
2. the bounds and permissions of `rs2` are a subset of those of `rs1`, and
3. neither `rs1` nor `rs2` fail any integrity checks
4. any extension-specific constraints capability subset relationships hold.

Otherwise set `rd` to 0. Extensions may further impose constraints on when `rd` is set to 1.

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

Included in

RVY

Operation

```
  let cs1_val = C(cs1);
  let cs2_val = C(cs2);

  X(rd) = zero_extend(bool_bits(
    (cs1_val.tag == cs2_val.tag) &
    capIsSubset(cs2_val, cs1_val) /* capIsSubset returns false if either
input
                                     has malformed bounds, perms, or non-zero
                                     reserved bits */
  ));
  RETIRE_SUCCESS
```

2.8.5.3. YAMASK

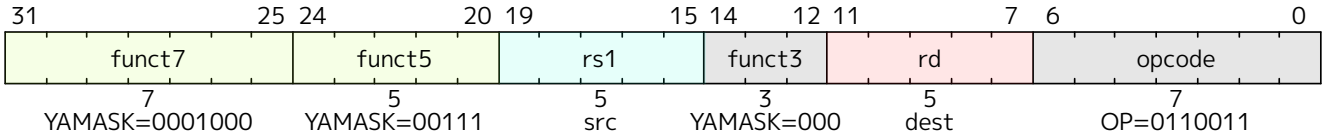
Synopsis

Capability alignment mask

Mnemonic

```
yamask rd, rs1
```

Encoding



Description

`rd[XLEN-1:0]` 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[XLEN-1:0]`. The upper bits of `rd` are zero extended. See Section 2.10.1.6 for the algorithm used to compute the next representable length.

Included in

RVY

Operation

```
let len = X(rs1);
X(rd) = getRepresentableAlignmentMask(len);
RETIRE_SUCCESS
```

2.8.6. Instructions to Load and Store Capability Data

New loads and stores are introduced to handle capability data, LY and SY. They atomically access YLEN bits of data and the associated capability tag.

All capability memory accesses check for C-permission in the authorizing capability in `rs1`.

If C-permission is granted then:

- LY reads YLEN bits of data from memory, and returns the associated capability tag.
- SY writes YLEN bits of data to memory, and writes the associated capability tag.

If C-permission is not granted then:

- LY reads YLEN bits from memory, but does not return the associated capability tag, instead zero is returned.
- SY writes YLEN bits to memory, and writes zero to the associated capability tag.

All capability data memory access instructions require YLEN-aligned addresses, and will take an access fault exception if this requirement is not met. They cannot be emulated.

An access fault is raised instead of a misaligned exception since these instructions cannot be emulated since there is one hidden capability tag per YLEN-aligned memory region.

All memory accesses, of any type, require permission from the authorizing capability in `rs1`.

- All loads require R-permission, otherwise they raise an exception.
- All stores require W-permission, otherwise they raise an exception.

Under some circumstances LY will *modify* the data loaded from memory before writing it back to the destination register. See LY for details.

Table 18. Capability load/store instruction summary in RVY

Mnemonic	Description
LY	Load capability
SY	Store capability

2.8.6.1. LY

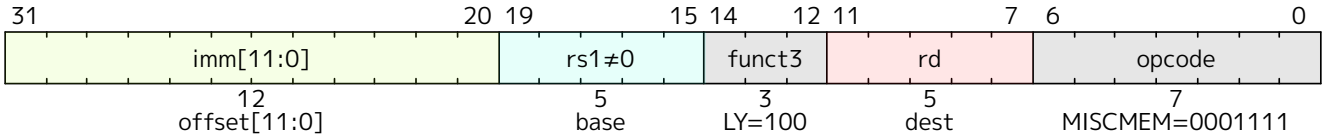
Synopsis

Load capability

Mnemonic

```
ly rd, offset(rs1)
```

Encoding



Any instance of this instruction with `rs1`=`x0` will raise an exception, as `x0` is defined to always hold a NULL capability. As such, the encodings with `rs1`=`x0` are RESERVED for use by future extensions.

Description

Calculate the effective address of the memory access by adding `rs1.address` to the sign-extended 12-bit offset.

Authorize the memory access with the capability in `rs1`.

Load a naturally aligned YLEN-bit data value from memory.

If the PMA is *CHERI Capability Tag* then load the associated capability tag, otherwise set the capability tag to zero.

The capability tag may also be cleared under platform specified conditions.

Extensions may also specify conditions which clear the capability tag.

Use the YLEN-bit data and the capability tag to determine the value of `rd` as specified below.

Platforms may strip loaded capability tags for a number of reasons. For example:

- *The RVY privileged specification permits Physical Memory Attributes (PMAs) that cannot store capability tags, as might be the case with MMIO. The capability tags transported by SY instructions to these regions are cleared and so will not read back.*
- *The MMU of the RVY privileged specification introduces page mappings that always clear loaded capability tags even if the loaded location has a set capability tag. Privileged software can use such mappings to limit capability propagation between virtual address spaces while still allowing for data exchange.*
- *CHERIoT platforms use so-called "capability load filters" to allow software (usually shared heap allocators) to ensure that capabilities pointing to deallocated memory cannot be loaded into the register file from memory.*
  - *A future extension is likely to specify the behavior of the "capability load filter" for embedded CHERI systems.*

This instruction can propagate valid capabilities which fail integrity checks.

Determining the final value of `rd`

If the capability tag is zero, or the authorizing capability (`rs1`) does not grant C-permission then set `rd.tag=0`. In this case the steps below do not apply.

If the capability tag is zero, all YLEN bits are transferred to the register without mutation, such that, for example, a SY instruction writing the destination register back to memory produces an exact copy. That is, the transferred value is not subjected to capability integrity checks and may, if viewed as a capbility, not be derivable from the system's capability roots. Especially, if the capability encoding has reserved bits or reserved values within its fields, these must be faithfully transported through registers. This property is essential for efficient, "capability-oblivious" implementation of C's `memcpy()`.

If `rd.tag=1`, `rd` is not sealed and `rs1` does not grant LM-permission, then an implicit YPERMC is performed to clear W-permission and LM-permission from `rd`.

Extensions may define further circumstances under which implict YPERMC-s or other mutation of loaded capabilities may take place.

While the implicit YPERMC introduces a dependency on the loaded data, implementations can avoid this by deferring the actual masking of permissions until the loaded capability is dereferenced or the metadata bits are inspected using YPERMR or YHIR. Additionally, metadata modifications are on naturally aligned data, and so on the read path from a data cache, the modification typically happens in parallel with data alignment multiplexers.

When sending load data to a trace interface, implementations trace the final value written to `rd` which may not match the value in memory.

Exceptions

Load access fault exception when the effective address is not aligned to YLEN/8.

Exceptions occur when the authorizing capability fails one of the checks listed below:

Kind	Reason
CHERI Load Access Fault	Authorizing capability tag is set to 0.
CHERI Load Access Fault	Authorizing capability is sealed.
CHERI Load Access Fault	Authorizing capability does not grant the necessary permissions.
CHERI Load Access Fault	At least one byte accessed is outside the authorizing capability bounds, or the bounds could not be decoded.
CHERI Load Access Fault	Authorizing capability failed any integrity check.

Included in

RVY

Operation

```
let offset : xlenbits = sign_extend(imm);
let (auth_val, vaddr) = get_cheri_mode_cap_addr(rs1_cs1, offset);
let aq : bool = false;
let rl : bool = false;

match check_and_handle_load_vaddr_for_triggers(vaddr, get_arch_pc()) {
```

```
    Some (ret) => return ret,
    None () => ()
  };
  if not(capTaggedAndReservedValid(auth_val)) then {
    handle_cheri_exception(CapCheckType_Data, CapEx_TagViolation);
    RETIRE_FAIL
  } else if capIsSealed(auth_val) then {
    handle_cheri_exception(CapCheckType_Data, CapEx_SealViolation);
    RETIRE_FAIL
  } else if not(canR(auth_val)) then {
    handle_cheri_exception(CapCheckType_Data, CapEx_PermissionViolation);
    RETIRE_FAIL
  } else if not(validAddrRange(vaddr, cap_size) |
capBoundsInfinite(auth_val)) then {
    handle_cheri_exception(CapCheckType_Data, CapEx_InvalidAddressViolation);
    RETIRE_FAIL
  } else if not(inCapBounds(auth_val, vaddr, cap_size)) then {
    handle_cheri_exception(CapCheckType_Data, CapEx_LengthViolation);
    RETIRE_FAIL
  } else if not(is_aligned_addr(vaddr, cap_size)) then {
    handle_mem_exception(vaddr, E_Load_Addr_Align());
    RETIRE_FAIL
  } else match translateAddr(vaddr, Read(Cap)) {
    TR_Failure(E_Extension(_)) => { internal_error(__FILE__, __LINE__,
"unexpected cheri exception for cap load") },
    TR_Failure(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
    TR_Address(addr, pbmt, ptw_info) => {
      let c = mem_read_cap(addr, pbmt, aq, aq & rl, false);
      match c {
        MemValue(v) => {
          let cr = clearTagIf(v, ptw_info.ptw_lc == PTW_LC_CLEAR |
not(canC(auth_val)));
          C(cd) = legalizeLM(cr, auth_val);
          RETIRE_SUCCESS
        },
        MemException(e) => {handle_mem_exception(vaddr, e); RETIRE_FAIL }
      }
    }
  }
```

2.8.6.2. SY

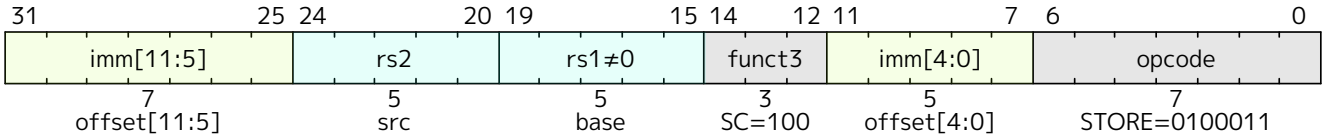
Synopsis

Store capability

Mnemonic

```
sy rs2, offset(rs1)
```

Encoding



Any instance of this instruction with `rs1`=`x0` *will raise an exception, as* `x0` *is defined to always hold a NULL capability. As such, the encodings with* `rs1`=`x0` *are RESERVED for use by future extensions.*

Description

Calculate the effective address of the memory access by adding `rs1.address` to the sign-extended 12-bit offset.

Authorize the memory access with the capability in `rs1`.

Store a naturally aligned YLEN-bit data value in `rs2` to memory and the associated capability tag in `rs2`.

This instruction can propagate valid capabilities which fail integrity checks.

Stored Capability Tag Value

The stored capability tag is set to zero if:

1. `rs2.tag=0`, or
2. `rs1` does not grant C-permission, or
3. The PMA is *CHERI Capability Tag Strip*

Extensions may define further circumstances under which stored capabilities may have their capability tags cleared.

Exceptions

Store/AMO access fault exception when the effective address is not aligned to YLEN/8.

Store/AMO access fault if the stored capability tag is set to one and the PMA is *CHERI Capability Tag Fault*.

Exceptions occur when the authorizing capability fails one of the checks listed below:

Kind	Reason
CHERI Store/AMO Access Fault	Authorizing capability tag is set to 0.
CHERI Store/AMO Access Fault	Authorizing capability is sealed.
CHERI Store/AMO Access Fault	Authorizing capability does not grant the necessary permissions.

Kind	Reason
CHERI Store/AMO Access Fault	At least one byte accessed is outside the authorizing capability bounds, or the bounds could not be decoded.
CHERI Store/AMO Access Fault	Authorizing capability failed any integrity check.

Included in

RVY

Operation

```
  let offset : xlenbits = sign_extend(imm);
  let (auth_val, vaddr) = get_cheri_mode_cap_addr(rs1_cs1, offset);
  let cs2_val = C(cs2);
  let aq : bool = false;
  let rl : bool = false;
  let cs2_val = clearTagIf(cs2_val, not(canC(auth_val)));

  match check_and_handle_store_vaddr_for_triggers(vaddr, get_arch_pc()) {
    Some (ret) => return ret,
    None () => ()
  };
  if not(capTaggedAndReservedValid(auth_val)) then {
    handle_cheri_exception(CapCheckType_Data, CapEx_TagViolation);
    RETIRE_FAIL
  } else if capIsSealed(auth_val) then {
    handle_cheri_exception(CapCheckType_Data, CapEx_SealViolation);
    RETIRE_FAIL
  } else if not(canW(auth_val)) then {
    handle_cheri_exception(CapCheckType_Data, CapEx_PermissionViolation);
    RETIRE_FAIL
  } else if not(validAddrRange(vaddr, cap_size) |
capBoundsInfinite(auth_val)) then {
    handle_cheri_exception(CapCheckType_Data, CapEx_InvalidAddressViolation);
    RETIRE_FAIL
  } else if not(inCapBounds(auth_val, vaddr, cap_size)) then {
    handle_cheri_exception(CapCheckType_Data, CapEx_LengthViolation);
    RETIRE_FAIL
  } else if not(is_aligned_addr(vaddr, cap_size)) then {
    handle_mem_exception(vaddr, E_SAMO_Addr_Align());
    RETIRE_FAIL
  } else match translateAddr(vaddr, Write(if cs2_val.tag then Cap else Data))
{
    TR_Failure(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
    TR_Address(addr, pbmt, _) => {
      let eares : MemoryOpResult(unit) = mem_write_ea_cap(addr, aq & rl, rl,
false);
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let res : MemoryOpResult(bool) = mem_write_cap(addr, pbmt, cs2_val,
```

```
aq & rl, rl, false);
          match (res) {
            MemValue(true)  => RETIRE_SUCCESS,
            MemValue(false) => internal_error(__FILE__, __LINE__, "store got
false from mem_write_value"),
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL
}
          }
        }
      }
    }
  }
```

2.9. Changes to Existing RISC-V Base ISA Instructions

RVY extend existing instructions that are used for handling addresses so that they manipulate a whole capability.

- Whenever an input operand is used as an address (e.g., the load/store base address), all capability bits are fed into the instruction instead of just XLEN bits.
- Any instruction that writes back an address (e.g., AUIPC (RVY) or CSRRW (RVY)) to the destination register, writes a full capability register instead of just XLEN bits. For all other results the high bits of the register and the capability tag are zeroed.
- Whenever a capability with a new address is returned, the result is *always* created using the semantics of the YADDRW instruction.

`ADD` and `ADDI` are not affected by the rule above. Even though they *are* used for handling addresses, they also have other uses. New encodings are used for capability addition: YADD and YADDI. They must be used for all address incrementing.

Integer add (`ADD`) and capability add (YADD) have separate encodings. Using a single encoding for both is undesirable:

1. *Integer ADD is most commonly used for purposes other than address calculations.*
2. *For high performance implementations which can issue multiple ADDs, it means that the integer ADD units don't need the upper halves of the operands, and don't need the capability check logic on the result.*
3. *The compiler and/or programmer would have to execute another metadata clearing instruction after each ADD to ensure that compartments don't leak capabilities.*

The rules above apply to the base ISA instructions listed in the following subsections, but also apply to instructions added by other extensions. Any change to instruction semantics (or remapping of opcodes) for RVY is called out in the chapter defining the extension.

2.9.1. Changes to load/stores

All load and store instructions behave as described in Load and Store Instructions with one fundamental difference:

- Any memory instruction that has `rs1` as a base address register reads the full capability register instead. The base address is unchanged, i.e., using the value from `rs1`. The metadata and capability tag are used to authorize the access.
- For a load instruction, the lower XLEN bits of the result written to the destination register are the same as in the RV32I/RV64I specification.

All load and store instructions authorized by `rs1` raise exceptions if any of these checks fail:

- `rs1` must not be `x0`[1]
- The capability tag (`rs1.tag`) must be set
- `rs1` must be unsealed
- For loads, R-permission must be set in `rs1`
- For stores, W-permission must be set in `rs1`
- All integrity checks on `rs1` must pass

[1] All load/store encodings are *reserved* if `rs1=x0` (since dereferencing NULL always faults).

All load instructions, except for the RVY LY, always zero the capability tag and metadata of the result register.

All store instructions, except for the RVY SY, always write zero to the capability tag or capability tags associated with the memory locations that are written to. Therefore, misaligned stores may clear up to two associated capability tag bits.

The changed interpretation of the base register also applies to all loads, stores and all other memory operations defined in later chapters of this specification with a base operand of `rs1` unless stated otherwise. **Under RVY *all* loads and stores are authorized by `rs1`.**

These rules affect the following base ISA instructions listed in Table 20, and also apply to instructions added by other extensions, e.g.,:

- Floating-point loads and stores.
- Vector load and stores.
- Atomic memory accesses, see "Zaamo" for RVY and "Zalrsc" for RVY.

Table 19. Changed RISC-V base ISA load/store instructions summary in RVY

Mnemonic	Description
LD, LW[U], LH[U], LB[U]	Integer loads (authorized by the capability in `rs1`)
SD, SW, SH, SB	Integer stores (authorized by the capability in `rs1`)

2.9.2. Changes to PC

- Whenever the address field of the pc is modified, it is *always* updated using the semantics of the YADDRW instruction. This includes adding an offset to the pc from direct jumps and branches for both the target address and the link register. In this case, e.g., `new_pc = YADDRW(old_pc, offset)`
- JALR (RVY) copies `rs1` into the pc, and increments the address field with the offset. In this case, e.g., `new_pc = YADDRW(rs1, offset)`

These rules affect the following base ISA instructions listed in Table 20, and also apply to instructions added by other extensions, e.g.:

Table 20. Changed RISC-V base ISA PC relative instructions summary in RVY

Mnemonic	Description
AUIPC (RVY)	Add upper immediate to pc
JAL (RVY)	Immediate offset jump, and link and seal to capability register
JALR (RVY)	Jump to capability register, and link and seal to capability register

2.9.3. AUIPC (RVY)

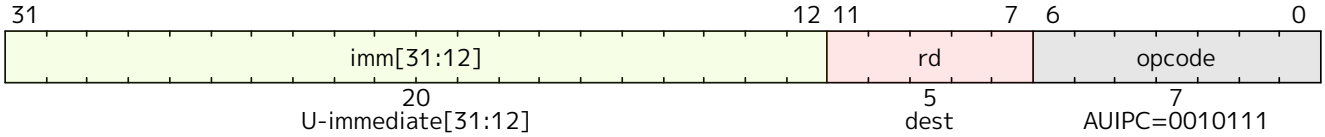
Synopsis

Add upper immediate to pc

Mnemonic

```
auipc rd, imm
```

Encoding



This instruction is extended from the version in the base ISA.

Description

Form a 32-bit offset from the 20-bit immediate filling the lowest bits with zero; the number of places to shift is determined by the capability encoding's choice of the AUIPC shift value (12, unless otherwise specified by the capability encoding format). Take the value of the AUIPC instruction's pc, increment its address by the 32-bit offset using the semantics of the YADDRW instruction and write the result to `rd`.

Set `rd.tag=0` if the resulting capability cannot be represented exactly.

Included in

RVI (RVY modified behavior)

Operation

```
let off : xlenbits = sign_extend(imm @ 0x000);
let (representable, newCap) = setCapAddr(PCC, PC + off);
C(cd) = clearTagIf(newCap, not(representable));
RETIRE_SUCCESS
```

2.9.4. The AUIPC Shift

The RISC-V base integer ISA frequently splits signed 32-bit constants accross instructions as the addition of a signed 12-bit constant and a 20-bit constant shifted left by 12 bits. For example, the I-type instruction `ADDI`, with its 12-bit immediate, is to be combined with the U-type `LUI` instruction and its 20-bit immediate when values beyond the reach of a signed 12-bit value are needed. To reach a given value in this way involve "overshooting" the desired value: for example, to materialize `0xf01` (`3841`) into a register, one uses `LUI` to materialize `0x1000` (`4096`) and `ADDI` to *subtract* `0xff` (`255`). Similarly, the U-type `AUIPC` instruction, with its 20-bit immediate, is designed to compose well with the signed 12-bit immediate operands of load (I-type) and store (S-type) instructions.

When manipulating addresses within capabilities, there is a risk that such two-step sequences could take the address out of bounds before attempting to bring it back within bounds. Many capability encodings, including those of RV32LYmw10rc1pc, have a **representable range** sufficient to ensure that any capability whose length is larger than 2 KiB (that is, those for which a signed 12-bit displacement might be insufficient) are able to represent at least 2 KiB on either side of their bounds. However, this is not an **essential** property of capability encodings, and so this specification allows the capability **encoding** to specify the shift used within address-manipulating instructions with shifted immediates. For AUIPC (RVY)

specifically, we refer to this value as **the AUIPC shift**, and take it to be 12 unless the capability encoding sets it to another value. (Taking the shift to be 11 instead of 12 decreases the reach of `AUIPC` from ±2 GiB to ±1 GiB, but ensures that all values within that range can be obtained with the same sign bit in the `AUIPC` immediate and subsequent 12-bit immediate(s), thereby ensuring that in-bounds addresses can be reached without risk of the intermediate computation exceeding capability bounds.)

Future extensions that add instructions with similar semantics should make use of this same encoding-specified shift value or otherwise allow the capability encoding to set the shift amount.

It is possible for the compiler to generate code that is compatible with any AUIPC shift by emitting a AUIPC with a zero offset followed by a sequence of LUI and YADD/YADDI.

2.9.5. JAL (RVY)

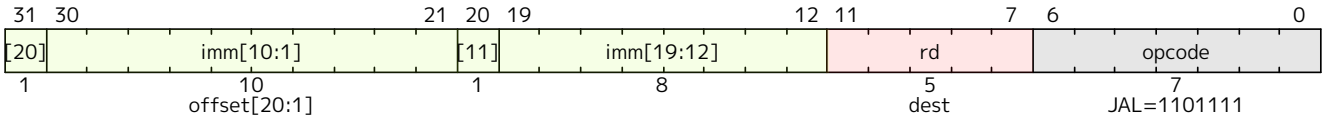
Synopsis

Immediate offset jump, and link and seal to capability register

Mnemonic

```
jal rd, offset
```

Encoding



Description

Jump to the target pc.

Increment pc.address by the sign-extended offset to form the target pc. The pc of the next instruction is sealed and written to `rd`.

Both address increments use the semantics of the YADDRW instruction to determine the result.

A future extension may raise an exception on the branch instruction itself if fetching a minimum sized instruction at the target pc will raise a CHERI Instruction Access Fault. Performing the pc bounds check at the branch source instead of on instruction fetch is helpful for debugging and can simplify the implementation of CPUs with very short pipelines.

Included in

RVI (RVY modified behavior)

Operation

TODO

2.9.6. JALR (RVY)

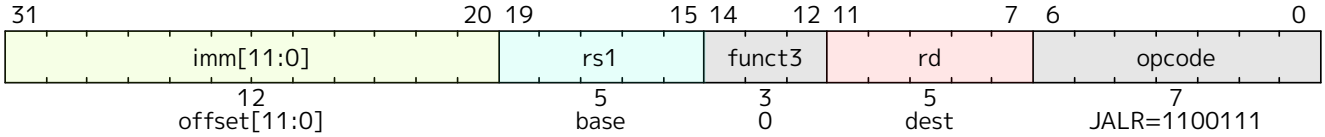
Synopsis

Jump to capability register, and link and seal to capability register

Mnemonic

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

Encoding



This instruction is extended from the version in the base ISA.

Description

Indirect jump to the target capability in `rs1` with an address offset.

The description below contains three hooks for extending JALR (RVY) behavior, used by extensions to give a large degree of extensibility. Unless an extension, such as Zysentry, is implemented which explicitly reference any of the hooks, then take no action for any of them.

Operationally, copy `rs1` to the target pc and then...

1. Compute the target pc address in two steps, each using the semantics of the YADDRW instruction:
   a. Increment the address of the target pc by the sign-extended 12-bit `offset`, and
   b. set bit zero of the target pc address to zero.
2. HOOK 1: Optionally clear the capability tag of the target pc depending on the CT-field value and the numerical values of `rd` and `rs1`
3. Unseal the target pc if it is a sentry capability, or clear the capability tag if it is any other sealed type.
4. If `rd≠x0`, compute the return capability and install it to `rd`:
   a. add the width of this instruction to the current pc using the semantics of the YADDRW instructions, and
   b. HOOK 2: Optionally seal the return capability as a sentry capability with a CT-field defined by the implemented extensions.
5. HOOK 3: Optionally make other architectural state updates.
6. Jump to the target pc.

When a sealed capability is passed as the input to JALR (RVY), its address must have bit zero clear and the instruction must have a zero offset, or the target pc will have its capability tag set to zero, since updates to its address are interpreted with YADDRW semantics.

A future extension may raise an exception on the JALR (RVY) instruction itself if the target pc will raise a CHERI Instruction Access Fault at the target. NOTE: A sentry defines a secure function entry point, and as such the offset in the JALR instruction must be zero. This is enforced by the use of YADDRW to add the offset which clears the tag of all sealed capabilities if the offset is non-zero.

Included in

RVI (RVY modified behavior)

Operation

TODO

2.9.7. Changes to BEQ, BNE

For `beq` and `bne` only, if `rs1≤rs2` then the encoding is RESERVED.

These encodings are redundant and may be used by future extensions.

Future behavior for these reserved branch encodings may include branching on capability tag values only, or YLEN-bit compares.

If the target of a taken branch lies outside the bounds of pc, the next instruction fetch will raise an exception.

A future extension may raise an exception on the branch instruction itself if fetching a minimum sized instruction at the target pc will raise a CHERI Instruction Access Fault.

2.10. The RV64LYmw14rc1ps Capability Base for RV64

ARC-NOTE: We originally had these two sections as separate chapters since they have a forward reference to the levels extension to define how those bits are allocated. Is this forward reference acceptable or do we need to push the definition into the Zylevels chapter? We originally had the latter, but we got feedback from implementers that it makes the spec more difficult to digest and confirm correct behavior.

This section describes the in-memory format and properties of the capability encoding intended for RV64.

The format is closely modeled upon features from CHERI v9 (Watson et al., 2023), and the bounds encoding scheme is based upon CHERI Concentrate (Woodruff et al., 2019).

2.10.1. Capability Encoding

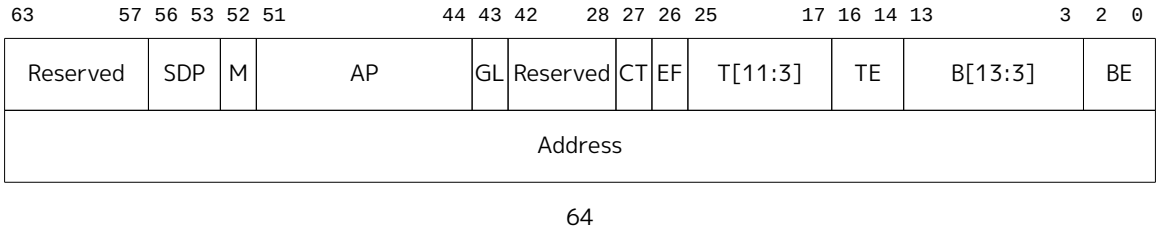


Figure 8. Capability encoding for RV64LYmw14rc1ps

Reserved bits must be 0 in valid capabilities and are available for future extensions to RVY.

The encoding diagram above of the capability format includes some fields which depend upon the presence of extensions:

Zyhybrid

When Zyhybrid is supported, capabilities include an M-bit (bit 52). If the capability does not grant X-

permission or Zyhybrid is not supported, the M-bit is always *reserved* and so must be zero in valid capabilities.

Zylevels1

If Zylevels1 is available, bits 6 and 7 of the `AP` field are allocated, otherwise they must always be set to one for any valid capability. Additionally, the GL flag is only available if Zylevels1 is implemented, otherwise it is reserved and must be zero.

This capability encoding has the following properties that affect the observable behavior of RVY instructions such as YBNDSW and YPERMC:

- **Mantissa width (mw14)**: The mantissa width for the bounds encoding uses 14 bits
- **Maximum exponent (e52)**: The maximum value for the exponent in a valid capability is 52.
- **Representable region (rc1)**: The encoding uses one additional bit to ensure a *centered* region of at least 1/4 of the capability size remains representable when creating out-of-bounds derived capabilities.
- **Permission encoding (ps)**: The permissions are encoded using a simple representation of one bit per architectural permission.

2.10.1.1. Capability Encoding Summary

Table 21. RV64LYmw14rc1ps parameter summary

Parameter	Value	Comment
MW	14	Mantissa width
EW	6	Exponent width
CAP_MAX_E	52	Maximum exponent value
enableL8	0	Whether the encoding format includes the L_8 bit
AP_MBit	0	Whether the M-bit is encoded in the AP-field
AP_MAX	ones	Value of the AP field giving maximum permissions

Table 22. RV64LYmw14rc1ps extension summary

Extension	Comment
Zyhybrid	Compatible
Zylevels1	Compatible
Zysentry	Compatible
Zyseal	Will be compatible once new permissions are encoded
All RV64Y versions of other standard extensions	Compatible

Table 23. RV64LYmw14rc1ps Feature summary

Feature	Comment
Representable region	At least 1/4 of the capability size
Permission encodings	All combinations can be represented, some combinations are reserved

2.10.1.2. Architectural Permissions (AP) Encoding

The permissions field is 8 bits wide and is encoded using one bit per architectural permission as shown in Table 24. A permission is granted if its corresponding bit, and those of any dependent permissions, are set; otherwise, the capability does not grant that permission. 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 legal to encode a subset of all combinations, but this capability encoding does not make use of this redundancy as there are sufficient reserved bits.

In the future, if there are no further remaining reserved bits, but extensions need to allocate further capability metadata bits, it will be possible to use the redundancy to create a future version of this encoding that is fully compatible for all software but with observable differences in the in-memory representation of capabilities.

Table 24. Encoding of architectural permissions for RV64LYmw14rc1ps

Bit	Encoded permission
0	C-permission
1	W-permission
2	R-permission
3	X-permission
4	ASR-permission
5	LM-permission
6	LG-permission if Zylevels1 is implemented; reserved `1` otherwise.
7	SL-permission if Zylevels1 is implemented; reserved `1` otherwise.

Future extensions may define new permissions and, if so, must augment the above table.

2.10.1.3. Capability Mode (M) Encoding

The M-bit is only assigned meaning when the implementation supports Zyhybrid *and* X-permission is set. In valid capabilities, the bit assigned to the M-bit must be zero if X-permission isn't set.

2.10.1.4. Software-Defined Permissions (SDP) Encoding

The SDP-field is 4 bits wide. The value of the SDP-field bits of the YPERMR result maps 1:1 to the SDP-field in the capability.

2.10.1.5. Capability Type (CT) Encoding

The capabilities of this chapter define a 1-bit field for CT-field values; this field directly encodes the values `0` and `1`. The value `1` is...

- considered ambiently available for YBLD,
- used as the type for capabilities sealed by YSENTRY instructions, regardless of the input capability's permission, if Zysentry is present in the platform.

Additionally, JALR (RVY) both

- unseals input capabilities of type 1 and
- seals its return capabilities with type 1.

JALR (RVY) places no constraints on the triple of input CT-field value, `rd` selector, and `rs1` selector. That is, JALR (RVY) will, as directed, attempt to jump to any unsealed or sealed capability in any register regardless of which register comes to hold the sealed return pointer.

The permission encodings of RV64LYmw14rc1ps do not provide mappings for the Zyseal extension's SE-permission or US-permission. Thus, without further revision, the encodings of this chapter are incompatible with the Zyseal extension.

2.10.1.6. Bounds (EF, T, TE, B, BE) Encoding

2.10.1.6.1. Concept

This bounds encoding scheme is based upon (Woodruff et al., 2019).

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 a compressed format, so it is not possible to encode any arbitrary combination of base and top addresses. An invalid capability with capability tag cleared is produced when attempting to construct a capability that is not *representable* because its bounds cannot be correctly encoded. The bounds are decoded as described in Section 2.10.1.

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 capability encoding as shown in Table 21.

The exponent E indicates the position of T and B within the capability's address as described in Section 2.10.1. The bit width of the exponent (EW) is set depending on the encoding. The maximum value of the exponent is calculated as follows:

```
CAP_MAX_E = XLEN - MW + 2
```

The values of EW and CAP_MAX_E are shown in Table 21.

The address and bounds must be representable in valid capabilities i.e., when the capability tag is set (see Section 2.10.1.6.4).

2.10.1.6.2. Decoding

The metadata is encoded in a compressed format termed CHERI Concentrate (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.

The pseudocode below does not have a formal notation. It is a place-holder until the sail implementation has been integrated in the specification. In this notation, `/` means "integer division", `[]` are the bit-select operators, and arithmetic is signed.

```
EW        = (XLEN == 32) ? 5 : 6
CAP_MAX_E = XLEN - 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            = (XLEN == 32) ? L~8~ : 0
else:
    E               = CAP_MAX_E - ( (XLEN == 32) ? { L~8~, 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
```

The bounds are decoded as shown in Figure 9 and Figure 10.

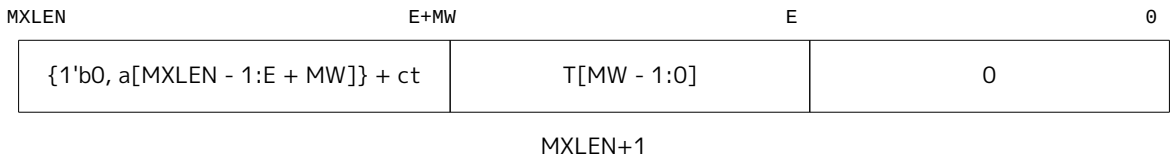


Figure 9. Decoding of the XLEN+1 wide top (t) bound

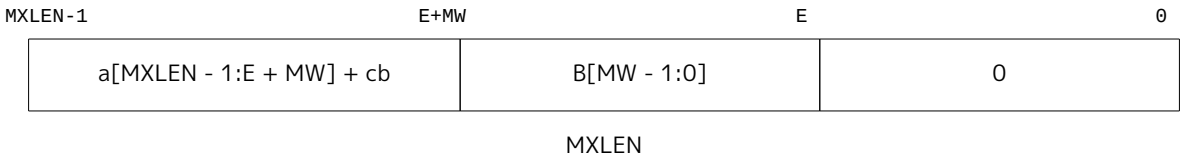


Figure 10. Decoding of the XLEN wide base (b) bound

Figure 9 and Figure 10 include ranges which may not be present when the bounds are decoded:

- If E = 0 the lower section does not exist.
- If E+MW=XLEN then the top section is only the least significant bit of c_t for the top bound, and top section doesn't exist for the bottom bound.
- If E+MW>XLEN then neither top section exists, and so the bounds are calculated with no dependency on the address field *a*.

The corrections c_t and c_b are calculated as shown below using the definitions in Table 25 and Table 26.

```
A[MW-1:0] = a[E + MW - 1:E]
R[MW-1:0] = B - 2^(MW-2)
```

The comparisons in Table 25 and Table 26 are unsigned.

Table 25. Calculation of top address correction

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

Table 26. 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. When `enableL8=1`, L_8 encodes the MSB of the length, which can be used to derive T[MW-1:MW-2], forming a full MW-wide T field.
2. EF = 0: The exponent is *internal* with E stored in the lower bits of T and B, with L_8 used for the MSB of E when `enableL8=1`. E is chosen so that the most significant non-zero bit of the length of the region aligns with T[MW - 2] such that this bit is implied by E.

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 (as well as L_8 when `enableL8=1`). 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 labeled $space_U$ in Figure 2 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.

2.10.1.6.3. Top bound MSB correction

A capability has *infinite* bounds if its bounds cover the entire address space such that the base address *b*=0

and the top address $t \geq 2^{XLEN}$, i.e., t is an XLEN + 1 bit value. However, b is an XLEN-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[XLEN: XLEN - 1] - b[XLEN - 1] > 1) )
    t[XLEN] = !t[XLEN]
```

The comparison is unsigned.

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 2.10.1.6.4); this check is equivalent to b=0 and $t \geq 2^{XLEN}$.

2.10.1.6.4. Malformed Capability Bounds

A capability is *malformed* if its bounds cannot be correctly decoded. The following check indicates whether a capability is malformed. If `enableL8` is true, the L_8 bit is available in the capability encoding format 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)
```

Capabilities with malformed bounds:

1. Return both base and top bounds as zero, which affects instructions like YBASER.
2. Cause certain manipulation instructions like YADDI to always set the capability tag of the result to zero.

2.10.2. Representable Range Check

The concept of the *representability check* was introduced in Section 2.3.7.

The definition of the check is:

- A source capability with address a, metadata m that decodes to give the bounds b and t.
- A derived capability with arbitrary address a' with the same metadata m that decodes to give the bounds b' and t'.

The address a' is within the source capability's *representable range* if `b == b' && t == t'`.

If the address a' is outside the *representable range*, then the derived capability has the capability tag set to zero.

2.10.2.1. Practical Information

An artifact of the bounds encoding is that if the new address causes `t != t'`, then it is also the case that `b`

`!= b'`.

The inverse is also true, if `b != b'` then `t != t'`.

Therefore, for representable range checking, it is acceptable to either check `t == t'` or `b == b'`.

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 < XLEN)
- 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 27. Composition of the decoded top address bound

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

The *representable range* defines the range of addresses which do not corrupt the bounds encoding. The encoding was first introduced in Section 2.10.1, and is repeated in a different form in Table 27 to aid this description.

For the address to be valid for the current bounds encoding, the value in the *Upper Section* of Table 27 *must not change* as this will change the meaning of the bounds. This is because `T`, `B` and `E` will be unchanged for the source and destination capabilities. Therefore, the Middle and Lower sections of the bounds calculation are also unchanged for source and destination capabilities.

When E > CAP_MAX_E - 2, the calculation of the top bound is entirely derived from `T` and `E` which will be identical for both the source and destination capabilities, thus guaranteeing that `t == t'`. Likewise, with such values of E, the base bound is entirely derived from `B` and `E` and therefore `b == b'`.

The calculation of the MSB of the top bound may be inverted as specified in Section 2.10.1.6.3. Assuming `(E < (CAP_MAX_E - 1))`, the truth-table for this inversion is as follows:

Table 28. Top bound MSB inversion truth table

input_t[XLEN:XLEN-1]	b[XLEN-1]	output_t[XLEN:XLEN-1]
00	0	00
01	0	01
10	0	00
11	0	01
00	**1**	**10**
01	1	01
10	**1**	**10**
11	1	01

Inspection of Table 28 shows that `output_t[XLEN]` does not depend on `input_t[XLEN]` as:

- `output_t[XLEN] = {input_t[XLEN-1], b[XLEN-1]} == 2'b01`.

This leads to the conclusions:

- If `t[XLEN-1] == t'[XLEN-1]` and `b[XLEN-1] == b'[XLEN-1]`, then it is guaranteed `t[XLEN] == t'[XLEN]`.
- If `t[XLEN-1] != t'[XLEN-1]` or `b[XLEN-1] != b'[XLEN-1]`, then the representable check will fail regardless of checking `t[XLEN] == t'[XLEN]`.

Therefore, for the purpose of representable range checking, it is not required to check that `t[XLEN]==t'[XLEN]`.

Given that `t[XLEN]` is not part of the representable range check:

- when `E == CAP_MAX_E - 2`, `t[XLEN-1:E] == T[MW-1:0]` and `b[XLEN-1:E] == B[MW-1:0]`.

Therefore, `T` and `B` are both derived from the capabilities metadata and are therefore constant. Which means that in this case too, the representable range check always passes.

As a result:

- If E > CAP_MAX_E - 3, then the representability check always passes, *even though the bounds are only infinite if E = CAP_MAX_E*

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

The gap between the object bounds and the bound of the representable range is always guaranteed to be at least 1/4 of `s`. This is represented by `R = B - 2^(MW-2)` in Section 2.10.1. This gives useful guarantees, such that if an executed instruction is in pc bounds, then it is also guaranteed that the next linear instruction is *representable*.

2.10.3. Encoding of Special Capabilities

2.10.3.1. NULL Capability Encoding

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), so its bounds cover the entire address space such that the expanded base is 0 and top is 2^{XLEN}.

Table 29. Field values of the NULL capability

Field	Value	Comment
Capability Tag	zero	Capability is not valid
SDP	zeros	Grants no permissions
AP	zeros	Grants no permissions
M^1	zero	No meaning since non-executable (Zyhybrid only)
CT	zero	Unsealed
EF	zero	Internal exponent format
$L_8{}^2$	zero	Top address reconstruction bit
T	zeros	Top address bits
T_E	zeros	Exponent bits
B	zeros	Base address bits

Field	Value	Comment
B_E	zeros	Exponent bits
Address	zeros	Capability address
Reserved	zeros	All reserved fields

[1] Only present if `AP_MBit=1` and Zyhybrid is implemented.

[2] Only present if `enableL8=1`.

Permissions added by extensions (such as those of Zylevels1) are presumed absent in NULL capabilities.

2.10.3.2. Infinite Capability Encoding

This encoding is for an *Infinite* capability value, which grants all permissions while its bounds also cover the whole address space. It includes X-permission and so includes the M-bit if Zyhybrid is supported. This infinite capability is both a Root Executable and a Root Data capability.

Table 30. Field values of the Infinite capability

Field	Value	Comment
Capability Tag	one	Capability is valid
SDP	ones	Grants all permissions
AP	AP_MAX	Grants all permissions
M[1]	one	CHERI execution mode
CT	zero	Unsealed
EF	zero	Internal exponent format
L_8[2]	zero	Top address reconstruction bit
T	zeros	Top address bits
T_E	zeros	Exponent bits
B	zeros	Base address bits
B_E	zeros	Exponent bits
Address	any[3]	Capability address
Reserved	zeros	All reserved fields

[1] Only present if `AP_MBit=1` and Zyhybrid is implemented.

[2] Only present if `enableL8` is set.

[3]If an infinite capability is used as a constant in either hardware or software, then the address field will typically be set to zero. If the address field is non-zero then it is still referred to as an infinite capability, and it still has the authority to authorize all memory accesses.

Permissions added by extensions (such as those of Zylevels1) are presumed present in Infinite capabilities.

2.11. The RV32LYmw10rc1pc Capability Base for RV32

This section describes an in-memory format and properties of a capability encoding intended for RV32. This format is heavily based upon the RV64LYmw14rc1ps format with the following changes:

- The width, or presence, of fields in the encoding are changed as defined in Table 31.
- The architectural permissions (AP) field is compressed to save encoding space, and so rules are defined for removing permissions.

2.11.1. Capability Encoding

The encoding format of the RV32LYmw10rc1pc capability is shown in Figure 8.

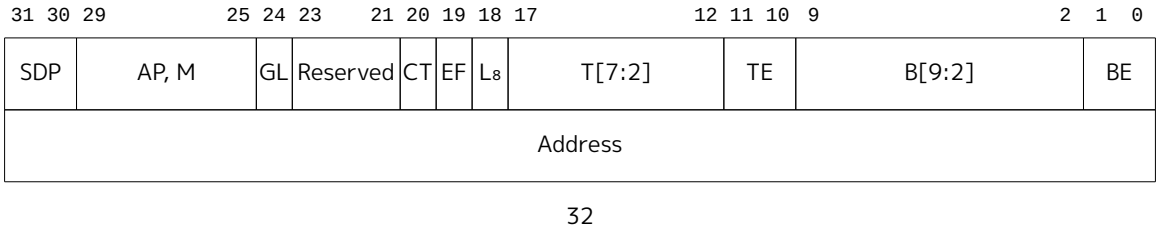


Figure 11. Capability encoding for RV32LYmw10rc1pc

Reserved bits must be 0 in valid capabilities and are available for future extensions to RVY.

Certain bits of the capability encoding are only used if certain extensions are implemented and are reserved otherwise:

Zyhybrid

When Zyhybrid is supported, capabilities include an M-bit (which is encoded as part of the M-bit encoding in the AP field). If not supported the M-bit is reserved and reads as zero.

Zylevels1

If Zylevels1 is available, additional values of the `AP,M` field are allocated, otherwise they are reserved for any valid capability. Additionally, the GL flag is only available if Zylevels1 is implemented, otherwise it is reserved and must be zero.

This capability encoding has the following properties that affect the observable behavior of RVY instructions such as YBNDSW and YPERMC:

- **Mantissa width (mw10)**: The mantissa width for the bounds encoding uses 10 bits
- **Maximum exponent (e24)**: The maximum value for the exponent in a valid capability is 24.
- **Representable region (rc1)**: The encoding uses one additional bit to ensure a *centered* region of at least 1/4 of the capability size remains representable when creating out-of-bounds derived capabilities.
- **Permission encoding (pc)**: The permissions are encoded using a compressed format that cannot represent every combination of permissions.

2.11.1.1. Capability Encoding Summary

Table 31. RV32LYmw10rc1pc parameter summary

Parameter	Value	Comment
MW	10	Mantissa width
EW	5	Exponent width
CAP_MAX_E	24	Maximum exponent value
enableL8	1	Whether the encoding format includes the L_8 bit

Parameter	Value	Comment
AP_MBit	1	Whether the M-bit is encoded in the AP-field
AP_MAX	0x8/0x9[1]	Value of the AP field giving maximum permissions

[1]If Zyhybrid is supported, then the infinite capability must represent *(Non-CHERI) Address Mode* for compatibility with standard RISC-V code. See Table 36.

Table 32. RV32LYmw1Orc1pc extension summary

Extension	Comment
Zyhybrid	Compatible
Zylevels1	Compatible
Zysentry	Compatible
Zyseal	Will be compatible once new permissions are encoded
All RV32Y versions of other standard extensions	Compatible

Table 33. RV32LYmw1Orc1pc Feature summary

Feature	Comment
Representable region	At least 1/4 of the capability size
Permission encodings	Not all combinations can be represented

2.11.1.2. Architectural Permissions and Mode (AP,M) Encoding

The permissions field is 5 bits wide and is encoded using a compressed representation as shown below. 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 legal to encode a subset of all combinations, and this redundancy is used to reduce the size of the permissions field compared to RV64LYmw14rc1ps.

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 36 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.

The M-bit is encoded as bit zero of the M-bit encoding in the AP field for the executable quadrant and only assigned meaning when the implementation supports Zyhybrid (*and* X-permission is set).

2.11.1.3. AP encoding and rules without Zylevels1 for RV32LYmw10rc1pc

Table 34. Encoding of architectural permissions for RV32LYmw10rc1pc without Zylevels1

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								
Field[2:0]	R	W	C	LM	X	ASR	Mode[1]	Notes
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-*(CHERI) Capability Mode*, 1-*(Non-CHERI) Address Mode*)								
Field[2:0]	R	W	C	LM	X	ASR	Mode[1]	Notes
0-1	✔	✔	✔	✔	✔	✔	Mode[1]	Execute + Data & Cap RW + ASR
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								
Field[2:0]	R	W	C	LM	X	ASR	Mode[1]	Notes
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								
Field[2:0]	R	W	C	LM	X	ASR	Mode[1]	Notes
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 valid capability when Zyhybrid is supported. Despite being encoded here it is* ***not*** *an architectural permission.*

When RV32LYmw10rc1pc there are many reserved permission encodings (see Table 36). It is not possible for a valid capability to have one of these values since YPERMC will never create it. It is possible for invalid capabilities to have reserved values. YPERMR will interpret reserved

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

Mode is encoded with permissions for RV32LYmw10rc1pc, but is not a permission. It is orthogonal to permissions as it can vary arbitrarily using YMODEW.

This encoding's compressed permission format specifies a *particular procedure* for encoding architectural permissions, which is used *instead of* YPERMC's default fixed-pointing procedure. If Zylevels1 is absent, the following rules are run once *in order*:

Table 35. RV32LYmw10rc1pc YPERMC rules if Zylevels1 is absent.

YPERMC Rule	Permission	Valid only if
RV32-base-1	C-permission	R-permission (supersedes base-1)
RV32-base-2	X-permission	R-permission
RV32-base-3	W-permission	not(C-permission) or LM-permission
RV32-base-4	X-permission	W-permission or C-permission
RV32-base-5	LM-permission	C-permission (supersedes base-2)
RV32-base-6	X-permission	(C-permission and LM-permission) or not (C-permission or LM-permission)
RV32-base-7	ASR-permission	W-permission and C-permission and X-permission (supersedes base-3)
RV32-base-8	M-bit	X-permission and Zyhybrid is implemented

2.11.1.4. AP encoding and rules with Zylevels1 for RV32LYmw10rc1pc

Table 36. Encoding of architectural permissions for RV32LYmw10rc1pc with Zylevels1

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										
Field[2:0]	R	W	C	LM	LG	SL	X	ASR	Mode[1]	Notes
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-*(CHERI) Capability Mode*, 1-*(Non-CHERI) Address Mode*)										
Field[2:0]	R	W	C	LM	LG	SL	X	ASR	Mode[1]	Notes
0-1	✔	✔	✔	✔	✔	✔	✔	✔	Mode[1]	Execute + Data & Cap RW + ASR
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										

Quadrant 0: Non-capability data read/write										
bit[2] = write, bit[1] reserved, bit[0] = !SL. R and C implicitly granted, LM dependent on W permission.										
Field[2:0]	R	W	C	LM	LG	SL	X	ASR	Mode[1]	Notes
0-2	reserved									
3	✔		✔						N/A	Data & Cap RO (without LM-permission)
4-5	reserved									
6	✔	✔	✔	✔		✔			N/A	Data & Cap RW (with SL-permission, no LG-permission)
7	✔	✔	✔	✔					N/A	Data & Cap RW (no SL-permission, no LG-permission)
Quadrant 3: Capability data read/write										
bit[2] = write, bit[1] reserved, bit[0] = !SL. R and C implicitly granted.										
Reserved bits for future extensions must be 1 so they are implicitly granted										
Field[2:0]	R	W	C	LM	LG	SL	X	ASR	Mode[1]	Notes
0-2	reserved									
3	✔		✔	✔	✔				N/A	Data & Cap RO
4-6	reserved									
6	✔	✔	✔	✔	✔	✔			N/A	Data & Cap RW (with SL-permission)
7	✔	✔	✔	✔	✔				N/A	Data & Cap RW (no SL-permission)

[1] *Mode (M-bit) can only be set on a valid capability when Zyhybrid is supported, otherwise such encodings are reserved. Despite being encoded here it is* ***not*** *an architectural permission.*

The following rules are run once *in order*:

Table 37. RV32LYmw1Orc1pc YPERMC rules if Zylevels1 is present.

YPERMC Rule	Permission	Valid only if
RV32-l1-1	C-permission	R-permission (supersedes base-1)
RV32-l1-2	X-permission	R-permission
RV32-l1-3	W-permission	not(C-permission) or LM-permission
RV32-l1-4	X-permission	W-permission or C-permission
RV32-l1-5	LM-permission	C-permission (supersedes base-2)
RV32-l1-6	LM-permission	W-permission or LG-permission
RV32-l1-7	LG-permission	LM-permission (supersedes Zylevels1-1)
RV32-l1-8	SL-permission	LM-permission and W-permission (supersedes Zylevels1-2)

YPERMC Rule	Permission	Valid only if
RV32-l1-9	X-permission	(C-permission and LM-permission and LG-permission and SL-permission) or (C-permission and LM-permission and LG-permission and not W-permission) or not (C-permission or LM-permission or LG-permission or SL-permission)
RV32-l1-10	ASR-permission	W-permission and C-permission and X-permission (supersedes base-3)
RV32-l1-11	M-bit	X-permission and Zyhybrid is implemented

For RV32, the encodings which have the M-bit set to 1 for *(Non-CHERI) Address Mode* are only valid if Zyhybrid is implemented. Otherwise those encodings represent invalid permissions.

2.11.1.5. Software-Defined Permissions (SDP) Encoding

The SDP-field is 2 bits wide. The value of the SDP-field bits of the YPERMR result maps 1:1 to the SDP-field in the capability.

2.11.1.6. Capability Type (CT) Encoding

Capabilities in this encoding have a 1-bit field for CT-field values which behaves in the same way as the RV64LYmw14rc1ps Section 2.10.1.5.

2.11.1.7. Bounds (EF, T, TE, B, BE, L_8) Encoding

The bounds are encoded in the same way as in RV64LYmw14rc1ps, with the appropriate values for mantissa width and maximum exponent substituted. Compared to RV64LYmw14rc1ps, this encoding uses an additional L_8 bit as described in Section 2.10.1.6.

2.11.2. Encoding of Special Capabilities

2.11.2.1. NULL Capability Encoding

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 (24), so its bounds cover the entire address space such that the expanded base is 0 and top is 2^{XLEN}.

Table 38. Field values of the NULL capability

Field	Value	Comment
Capability Tag	zero	Capability is not valid
SDP	zeros	Grants no permissions
AP	zeros	Grants no permissions
CT	zero	Unsealed
EF	zero	Internal exponent format
L_8	zero	Top address reconstruction bit
T	zeros	Top address bits
T_E	zeros	Exponent bits

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

2.11.2.2. Infinite Capability Encoding

This encoding is for an *Infinite* capability value, which grants all permissions while its bounds also cover the whole address space. It includes X-permission and so includes the M-bit if Zyhybrid is supported. This infinite capability is both a Root Executable and a Root Data capability.

Table 39. Field values of the Infinite capability

Field	Value	Comment
Capability Tag	one	Capability is valid
SDP	ones	Grants all permissions
AP	0x8/0x9[1]	Grants all permissions
CT	zero	Unsealed
EF	zero	Internal exponent format
L_8	zero	Top address reconstruction bit
T	zeros	Top address bits
T_E	zeros	Exponent bits
B	zeros	Base address bits
B_E	zeros	Exponent bits
Address	any[2]	Capability address
Reserved	zeros	All reserved fields

[1]If Zyhybrid is supported, then the infinite capability must represent *(Non-CHERI) Address Mode* for compatibility with standard RISC-V code. Therefore, the M-bit is set to 1 in the M-bit encoding in the AP field, giving the value 0x9.

[2]If an infinite capability is used as a constant in either hardware or software, then the address field will typically be set to zero. If the address field is non-zero then it is still referred to as an infinite capability, and it still has the authority to authorize all memory accesses.

Permissions added by extensions (such as those of Zylevels1) are presumed present in Infinite capabilities.

2.11.3. Representable Range Check

The representable range check behaves in exactly the same way as in RV64LYmw14rc1ps, just with the 10-bit mantissa width of this encoding.

Chapter 3. "Zysentry" Extension for Creation of Sentry Capabilities

The Zysentry extension:

1. Defines one sentry capability type, the unrestricted sentry type with a CT-field of `1`.
    a. These *unrestricted sentry* capabilities can be used as immutable code pointers for both forward and backward control flow edges. NOTE: A future extension may define more restrictive *forward-only* and *backward-only* sentry capabilities that can only be used by calls and returns respectively.
2. Adds the YSENTRY instruction to allow sealing capabilities as sentries with CT-field of `1`.

3.1. Interaction with JALR (RVY)

Zysentry adds sealing and unsealing behavior to JALR (RVY):

1. For HOOK 2: Seal the return capability as a sentry.

3.2. Added instructions

3.2.1. YSENTRY

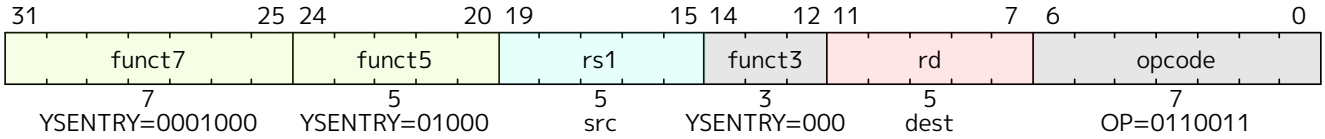
Synopsis

Seal capability as a sentry

Mnemonic

```
ysentry rd, rs1
```

Encoding



Description

Copy `rs1` to `rd`.

Set the capability type (CT-field) of `rd` to the ambient value specified by the capability encoding given the permissions granted by the capability in `rs1`.

Set `rd.tag=0` if `rs1` is sealed.

Included in

Zysentry

Operation

```
let cs1_val = C(cs1);
let inCap = clearTagIf(cs1_val, capIsSealed(cs1_val));
C(cd) = sealCap(inCap);
```

```
RETIRE_SUCCESS
```

Chapter 4. "Zybld" Extension for Building Capabilities

The Zybld extension adds the YBLD instruction to capabilities with a superset authority to validate (i.e. set the capability tag) of a capability with lesser authority.

This instruction can be used to speed up operations such as paging in memory after swap.

CHERIoT implementations do not use YBLD, so this instruction is part of an optional extension instead of the RVY base ISA.

4.1. Added instructions

4.1.1. YBLD

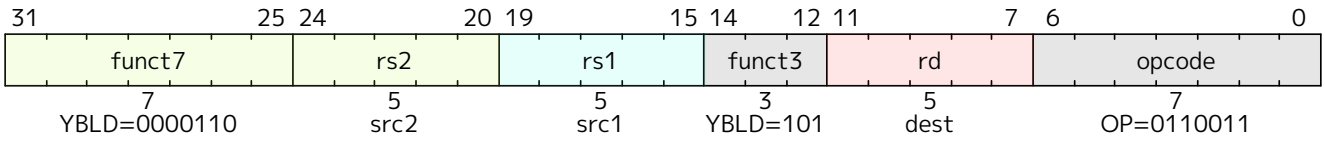
Synopsis

Build capability

Mnemonic

```
ybld rd, rs1, rs2
```

Encoding



Description

Copy `rs2` to `rd`.

If `rd.ct` (that is, its CT-field) is neither 0 nor an ambient type, then set `rd.ct` to 0.

Set `rd.tag=1` if:

1. `rs1.tag=1`, and
2. `rs1` passes all integrity checks, and
3. `rs1` is not sealed, and
4. `rs2` 's permissions and bounds are equal to or a subset of `rs1` 's, and
5. `rs2` passes all integrity checks, and
6. any extension-specific constraints on YBLD hold.

Otherwise, set `rd.tag=0`

The integrity check on `rs2` is required to prevent authorising a capability with a lack of integrity. The integrity check on `rs1` is optional.

YBLD will construct a sealed capability only if its type is ambiently available.

YBLD is typically used alongside YHIW to build capabilities from integer values.

When `rs1` is `x0` YBLD will copy `rs2` to `rd` and clear `rd.tag`. However future extensions may add additional behavior to update currently reserved fields, and so software should not assume

`rs1==0` *to be a pseudo-instruction for capability tag clearing.*

Included in

Zybld

Operation

```
  let cs1_val = C(cs1);
  let cs2_val = C(cs2);

  let tag = cs1_val.tag &
            not(capIsSealed(cs1_val)) &
            capIsSubset(cs2_val, cs1_val); /* Subset checks for malformed
bounds,
                                              perms, and reserved bits */

  C(cd) = { cs2_val with tag = tag };
  RETIRE_SUCCESS
```

Chapter 5. "Zytopr" Extension for Extracting the Top Bound for Memory Allocators

The Zytopr extension adds the YTOPR instruction to return the top bound of a capability. This offers a performance improvement over the software sequence of a saturating add of the results of YBASER and YLENR, and is useful for memory allocators.

5.1. Added instructions

5.1.1. YTOPR

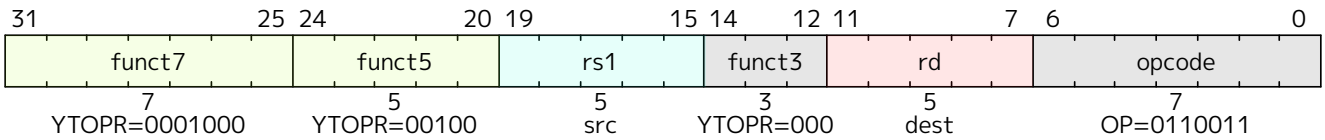
Synopsis

Read capability top address

Mnemonic

```
ytopr rd, rs1
```

Encoding



Description

Decode the base integer address from `rs1` 's bounds and write the result to `rd`.

If `rs1` 's bounds can't be decoded, or `rs1` fails any integrity checks, then return zero.

> *The value of* `rs1.tag` *does not affect the result.*

Included in

Zytopr

Operation

TODO

Chapter 6. "Zyhybrid" Extension for CHERI Execution Modes

This chapter will appear in the unpriv spec somewhere after the Zicsr chapter (since it depends on Zicsr).

Zyhybrid is an optional extension to RVY which adds the ability to dynamically change the base architecture of the hart between CHERI (RVY) and standard RISC-V (RVI/RVE).

The ability to choose between these two behaviors is referred to as switching between *CHERI Execution Modes*. The mode is controlled by a new bit (the M-bit) allocated in the pc.

Zyhybrid adds the instructions shown in Zyhybrid which add the ability to query and update the current mode.

Zyhybrid also adds a new unprivileged CSR: the default data capability, ddc. ddc is used to authorize all data memory accesses when executing RVI/RVE code.

Together with pc, ddc allows confining code runs to a compartment (also called a sandbox), where all data memory and instruction memory accesses are bounded to fixed memory regions. These compartments have full binary compatibility with all existing ratified RISC-V base architectures and extensions, i.e., non-CHERI-aware programs that execute unmodified. Provided that the privileged execution environment has set up ddc and pc appropriately, non-CHERI-aware programs will execute unmodified (as long as they don't attempt to access memory out of the defined bounds).

RVY implementations which support Zyhybrid are typically referred to as CHERI Hybrid, whereas implementations which do not support Zyhybrid are typically referred to as CHERI purecap.

6.1. CHERI Execution Modes

The two execution modes are:

(Non-CHERI) Address Mode
: Executing with the RVI (or RVE) base ISA.

If RVC encodings are supported, load/store encodings will revert to their non-CHERI encodings, such as C.LYSP reverting to C.FLWSP for RV32F. This behavior is summarized in Table 64, Table 65, Table 66 and Table 67.

Instructions which are modified on an RVY architecture (see Table 57) revert to their standard behavior.

All RVY instructions, and associated CSRs, are available in addition to RVI/RVE and all supported non-CHERI extensions.

The authorizing capability for memory access is ddc (as opposed to `rs1`). That is, all memory accesses, including PREFETCH.W (RVY) and PREFETCH.R (RVY), are implicitly authorized by ddc and only the memory address is sourced from `rs1`. PREFETCH.I (RVY) is the exception to this rule, as pc authorizes it.

ddc is also used to authorize RVY specific memory instructions such as LY and SY.

All CSR accesses to YLEN CSRs only access the lower XLEN bits, and, if writing, update the CSR using

the semantics of the YADDRW instruction (see Section 6.4).

(CHERI) Capability Mode

Executing with the RVY base ISA.

The *CHERI Execution Mode* is key in providing backwards compatibility with the base RV32I/RV64I ISA. RISC-V software is able to execute unchanged in implementations supporting Zyhybrid provided that the privileged environment sets up ddc and pc appropriately.

The CHERI execution mode is always (CHERI) Capability Mode on implementations that support RVY, but not Zyhybrid.

Software is referred to 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 RVY. Software is referred to as hybrid if it uses integer addresses **or** *CHERI capabilities for memory accesses. Hybrid software requires the CHERI RISC-V hart to support RVY and Zyhybrid.*

6.1.1. CHERI Execution Mode Encoding

The *CHERI Execution Mode* is determined by a bit in the metadata of the pc called the M-bit. Zyhybrid adds a new *CHERI Execution Mode* field (M) to the capability format. This field needs to be present only in capabilities granting X-permission, as it is only ever architecturally interpreted on the capability resident in pc.

While the M-bit can also be read/written explicitly using YMODER/YMODEW, it only affects machine state once installed into pc.

Capabilities not granting X-permission may or may not have a defined M field, and attempting to update this field may be a no-op. Zyhybrid is compatible only with capability encodings that specify transport of the M-bit (see RV32LYmw1Orc1pc and RV64LYmw14rc1ps, for examples).

- Mode (M)=0 indicates *(CHERI) Capability Mode*.
- Mode (M)=1 indicates *(Non-CHERI) Address Mode*.

Since indirect jumps copy the full target capability into pc, it allows indirect jumps to change between modes (see Section 6.1.2).

When executing YPERMC, if X-permission is removed when the M-bit is set, and the capability encoding still permits representing the M-bit, then the M-bit must be set to zero.

Only RV64LYmw14rc1ps encodes the M-bit when X-permission is not granted.

6.1.2. Changing CHERI Execution Mode

The M-bit of pc can be updated by the instructions listed in Table 40:

Table 40. Zyhybrid instructions that can perform mode changes

Mnemonic	From mode	Description
JALR (RVY)	*(CHERI) Capability Mode*	Jump to capability register, and link and seal to capability register
YMODESWI	*(CHERI) Capability Mode*	Switch execution to *(Non-CHERI) Address Mode*

Mnemonic	From mode	Description
YMODESWY	*(Non-CHERI) Address Mode*	Switch execution to *(CHERI) Capability Mode*

When JALR (RVY) copies `rs1` *into pc it includes copying the M-bit and so setting the CHERI Execution Mode of the target instruction.*

The M-bit of a X-permission-granting capability can be read and written by the instructions listed in Table 41:

Table 41. Zyhybrid instructions to observe and update the mode in a capability

Mnemonic	Description
YMODEW	Set capability execution mode
YMODER	Read capability mode

In addition to the mode switching instructions, the current mode can also be updated by setting the M-bit of a target capability using YMODEW followed by a JALR (RVY).

6.1.3. Representation of the M-bit in the capability encoding

For capabilities that do not grant X-permission, M-bit must always be interpreted and reported as 0 representing *(CHERI) Capability Mode*.

While this is not phrased as an additional rule for YPERMC to follow, beyond those of Section 2.3.10.1, capability encodings may nevertheless take advantage of this implication in their representation of architectural CHERI capabilities.

6.1.4. Observing the CHERI Execution Mode

The effective CHERI execution mode cannot be determined just by reading the M-bit from pc since it also depends on the execution environment. The following code sequence demonstrates how a program can observe the current, effective CHERI execution mode. It will write, to `x1`, the value `1` for *(CHERI) Capability Mode* (wherein `pc` has a set capability tag) or `0` for *(Non-CHERI) Address Mode* (wherein `pc` is just an address and has a clear capability tag):

```
auipc x1, 0
ytagr x1, x1
```

Implementations that support Zyhybrid will typically boot into (Non-CHERI) Address Mode so that non-CHERI-aware software can run unmodified. CHERI-aware software can observe and switch the mode as required.

6.2. Added instructions

6.2.1. YMODEW

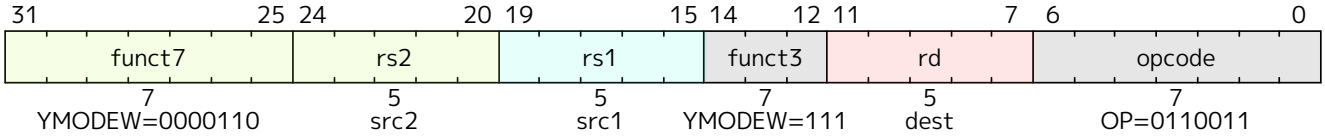
Synopsis

Set capability execution mode

Mnemonic

```
ymodew rd, rs1, rs2
```

Encoding



Description

Copy `rs1` to `rd`.

If `rs1` is sealed or if `rs1` fails any integrity check, then set `rd.tag=0`.

Otherwise, if `rs1` grants X-permission then update the M-bit of `rd` to:

1. *(CHERI) Capability Mode* if the least significant bit of `rs2` is 0, or,
2. *(Non-CHERI) Address Mode* if the least significant bit of `rs2` is 1.

> *The value of* `rs1.tag` *does not affect the result.*

Included in

Zyhybrid

Operation

```
let cap = C(cs1);
let mode = execution_mode_encdec(X(rs2)[0 .. 0]);

let cap = clearTagIf(cap, capIsSealed(cap));
let hasMode = not(permsMalformed(cap)) & canX(cap);
let newCap = if hasMode then setCapMode(cap, mode) else cap;

C(cd) = newCap;
RETIRE_SUCCESS
```

6.2.2. YMODER

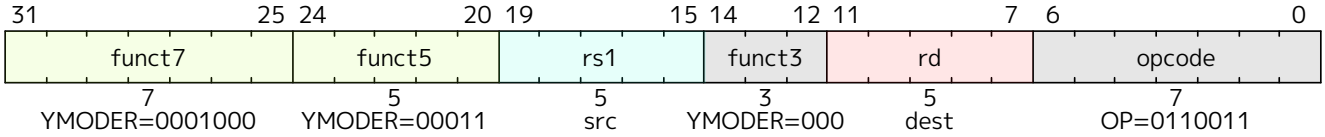
Synopsis

Read capability mode

Mnemonic

```
ymoder rd, rs1
```

Encoding



Description

Decode the CHERI execution mode from the capability in `rs1` and write the result to `rd`.

Set `rd` to 0 if `rs1` does not grant X-permission

Set `rd` to 0 if any integrity checks failed.

Otherwise set `rd` according to `rs1` 's CHERI execution mode (M-bit):

1. Set `rd` to 0 for *(CHERI) Capability Mode*, or,
2. Set `rd` to 1 for *(Non-CHERI) Address Mode*.

> *The value of* `rs1.tag` *does not affect the result.*

Included in

Zyhybrid

Operation

```
let capVal = C(cs1);
X(rd) = zero_extend(execution_mode_encdec(getCapMode(capVal)));
RETIRE_SUCCESS
```

6.2.3. YMODESWI

See YMODESWY.

6.2.4. YMODESWY

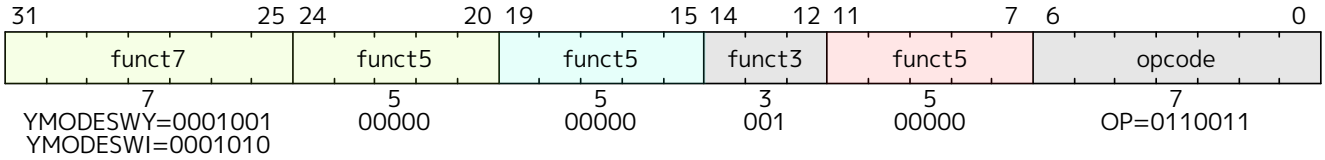
Synopsis

Switch execution mode to *(CHERI) Capability Mode* (YMODESWY), or *(Non-CHERI) Address Mode* (YMODESWI), 32-bit encodings

Mnemonic

```
ymodeswy
ymodeswi
```

Encoding



Description

Set the current CHERI execution mode in pc.

- YMODESWY: If the current mode in pc is *(Non-CHERI) Address Mode* (1), then the M-bit in pc is set to *(CHERI) Capability Mode* (0). Otherwise no effect.
- YMODESWI: If the current mode in pc is *(CHERI) Capability Mode* (0), then the M-bit in pc is set to *(Non-CHERI) Address Mode* (1). Otherwise no effect.

Included in

Zyhybrid

Operation

```
let mode : ExecutionMode = match effective_cheri_mode() {
  IntPtrMode => CapPtrMode,
  CapPtrMode => IntPtrMode,
};
if debug_mode_active then dinfc = setCapMode(infinite_cap, mode);
set_next_pcc(setCapMode(PCC, mode));
RETIRE_SUCCESS
```

6.3. Added State

Zyhybrid adds the YLEN-wide CSR shown in Table 42.

Table 42. Unprivileged YLEN-wide CSRs added in Zyhybrid

YLEN CSR	Permissions	Description
ddc	URW	User Default Data Capability

6.3.1. Default Data Capability CSR (ddc)

ddc is a read-write, user mode accessible capability CSR. It does not require ASR-permission in pc for writes or reads. Similarly to pc authorizing all control flow and instruction fetches, this capability register is implicitly checked to authorize all data memory accesses when the current CHERI mode is *(Non-CHERI) Address Mode*. On startup ddc bounds and permissions must be set such that the program can run successfully (e.g., by setting it to have sufficiently broad bounds and permissions, possibly a Root Data capability).

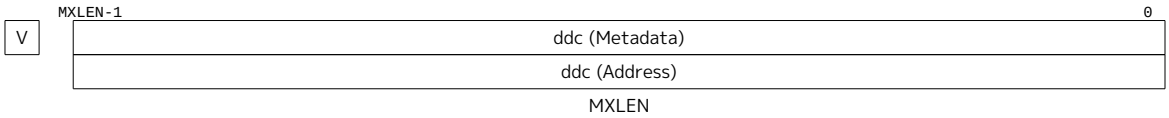


Figure 12. Unprivileged default data capability register

6.4. Changes to Zicsr Instructions

When in *(Non-CHERI) Address Mode*, there is a special rule for updating extended CSRs (e.g., jvt (RVY)). These are CSRs that are XLEN-wide for RVI/RVE but YLEN-wide for RVY.

- Writing an extended CSR writes the address field (XLEN bits) only, and the full CSR is updated using the semantics of the YADDRW instruction.
- Reading an extended CSR reads the address field (XLEN bits) only.

Accesses to extended CSRs in *(Non-CHERI) Address Mode* must access only XLEN bits for compatibility, and so use the semantics of the YADDRW instruction to determine the final written value.

YLEN-wide CSRs are accessed identically in either CHERI Execution Mode.

Table 43. YLEN-bit CSR and Extended CSR access summary for Zyhybrid

	YLEN-bit CSR[1]		Extended CSR[2]	
Instruction	Read Width	Write Width	Read Width	Write Width
CSRRW rd==x0		YLEN		XLEN
CSRRW rd!=x0	YLEN	YLEN	XLEN	XLEN
CSRR[C|S] rs1==x0	YLEN		XLEN	
CSRR[C|S] rs1!=x0	YLEN	XLEN	XLEN	XLEN
CSRRWI rd==x0		XLEN		XLEN
CSRRWI rd!=x0	YLEN	XLEN	XLEN	XLEN
CSRR[C|S]I uimm==x0	YLEN		XLEN	
CSRR[C|S]I uimm!=x0	YLEN	XLEN	XLEN	XLEN

[1] e.g., utidc

[2] e.g., jvt (RVY)

6.4.1. CSRRWI (RVY)

See CSRRCI (RVY).

6.4.2. CSRRS (RVY)

See CSRRCI (RVY).

6.4.3. CSRRSI (RVY)

See CSRRCI (RVY).

6.4.4. CSRRC (RVY)

See CSRRCI (RVY).

6.4.5. CSRRCI (RVY)

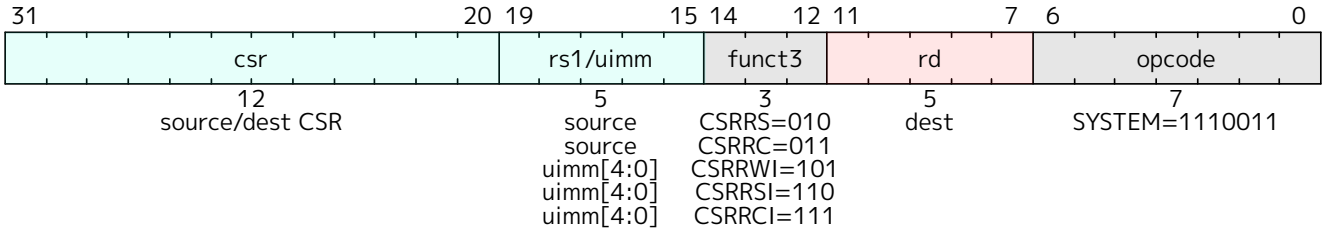
Synopsis

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

Mnemonics

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

Encoding



Description

These CSR instructions have extended functionality for accessing YLEN bit CSRs, and XLEN bit CSRs extended to YLEN bits (*Extended CSRs*).

Access to XLEN bit CSRs is as defined in Zicsr.

Zicsr rules are followed when determining whether to read or write the CSR.

Suppressed read or write actions have no side-effects on the CSR.

All writes are XLEN bits only, as determined by Zicsr, and use the semantics of the YADDRW instruction to determine the final write data.

Read data from extended CSRs is YLEN bits in *(CHERI) Capability Mode* or, if Zyhybrid is supported,

XLEN bits in *(Non-CHERI) Address Mode*.

Read data from YLEN bit CSRs is always YLEN bits.

In all cases, when writing YLEN bits of `rs1`, if any integrity check fails then set the capability tag to zero before writing to the CSR.

Permissions

Accessing CSRs may require ASR-permission.

Prerequisites

RVY, Zicsr

Included in

RVI (RVY modified behavior)

Operation

```
TBD
```

6.4.6. CSRRW (RVY)

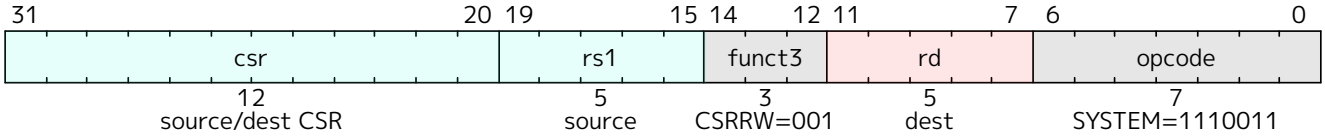
Synopsis

CSR access (CSRRW) 32-bit encodings for (RVY)

Mnemonic

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

Encoding



Description

CSRRW has extended functionality for accessing YLEN-bit CSRs, and XLEN-bit CSRs extended to YLEN bits (*Extended CSRs*).

Access to XLEN bit CSRs is as defined in Zicsr.

CSRRW accesses to YLEN bit CSRs read YLEN bits into `rd` and write YLEN bits of `rs1` into the CSR.

CSRRW accesses to extended CSRs read YLEN bits into `rd` and write YLEN bits of `rs1` into the CSR, or if Zyhybrid is supported, XLEN bit accesses are made in *(Non-CHERI) Address Mode*. The final write data is determined using semantics of the YADDRW instruction.

In all cases, when writing YLEN bits of `rs1`, if any integrity check fails then set the capability tag to zero before writing to the CSR.

Permissions

Accessing CSRs may require ASR-permission.

Prerequisites

RVY, Zicsr

Included in

RVI (RVY modified behavior)

Operation

```
TBD
```

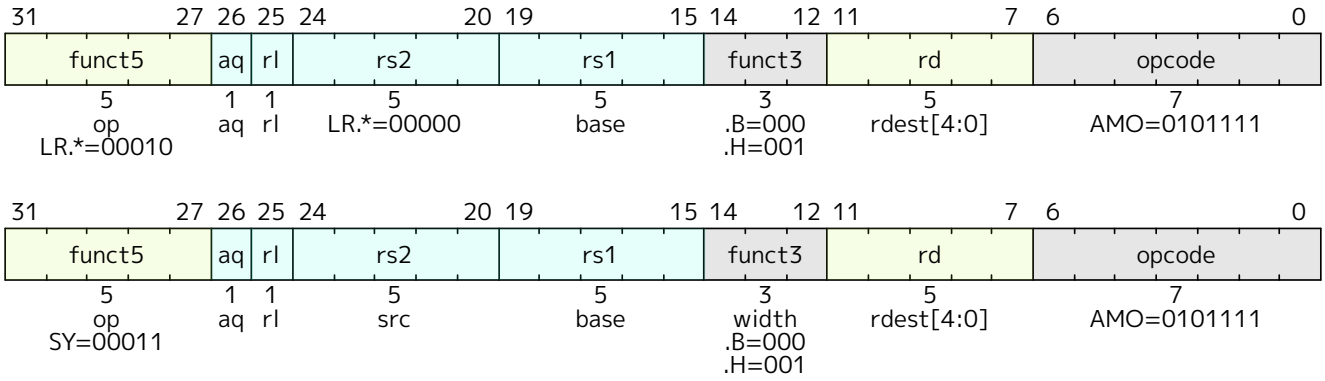
Chapter 7. "Zabhlrsc" Extension for Byte and Halfword Load Reserved/Store Conditional, Version 0.9

The Zalrsc extension offers LR/SC (load reserved/store conditional) instructions for *words* and *doublewords*. Zabhlrsc extends this by adding *byte* and *halfword* versions.

The absence of LR/SC operations for subword data types is problematic for CHERI software (RVY base architectures). Non-CHERI RISC-V software can use LR/SC on larger data types than are strictly required for the memory access to register the reservation set. RVY checks memory bounds and so it is impossible to round a subword access up to a word or larger to gain the reservation set.

7.1. Byte and Halfword Atomic Load Reserved/Store Conditional Instructions

The Zabhlrsc extension provides the `LR.[B|H]` and `SC.[B|H]` instructions.



LR.[B|H] behave analogously to LR.[W|D].

SC.[B|H] behave analogously to SC.[W|D].

All Zabhlrsc instructions sign extend the result and write it to *rd*. :leveloffset: +1

Chapter 8. Vector "V" Extension (RVY)

The Vector extension is orthogonal to RVY because the vector registers do not support capability tags.

A future extension may allow capability 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 capability tags of any copied capabilities will be set to 0 in the destination memory.*

Under RVY, vector loads and stores follow the standard rules for *active* elements:

- Only *active* elements are subject to CHERI exception checks.
- If there are no *active* elements then no CHERI exceptions will be raised.
- CHERI exceptions are only raised on fault-only-first loads if element 0 is both *active* and fails any exception checks.

Additionally, the standard RVY rule that all loads and stores where the base register is `x0` are reserved applies to all vector memory access instructions.

The approach of using indexed loads with the base register set to the value zero and XLEN-wide offsets does not work well with CHERI as the authorizing capability must cover all of memory. If the authorizing capability is specified as `x0` *then the instruction encoding is reserved.*

Chapter 9. "Zylevels1" Extension for CHERI 2-Level Information Flow Control

Zylevels1 introduces a simple *Information Flow Control* (IFC) mechanism to CHERI.

In this simple IFC system, capabilities are said to be either *global* or *local*. While the distinction between *global* and *local* is not one of *authority* (that is, the distinction is not one of *permission*), *global* capabilities may attenuate into *local* capabilities. The distinction refines the behavior of capability store and load instructions:

- Capability-write-permissive capabilities are refined to authorize stores of *any* capability or *global* capabilities only. The former may attenuate into the latter. Attempting to store a local capability through an insufficiently permissive authority clears the capability tag of the value written to memory, if any.
- Capability-load-permissive capabilities are refined to authorize loads of *any* capabilities or *local* capabilities only. Again, the former may attenuate to the latter. Attempting to load a global capability through an insufficiently permissive authority instead *attenuates* the load result as described below.

9.1. Added Architectural Permissions (AP) Bits

Permission	Type	Comment
SL-permission	Data memory permission	Used to filter the validity of stored capabilities.
LG-permission	Data memory permission	Used to filter the permissions of loaded capabilities.

Store Local Permission (SL)

This field allows limiting the propagation of *local* capabilities.

A capability without GL(obal) Flag set stored using an authorizing capability lacking SL-permission will be stored with a zero capability tag.

SL-permission is a *refinement of* C-permission and W-permission. That is, if either of the latter are clear, then SL-permission has no effect.

Load Global Permission (LG)

This field allows limiting the propagation of *global* capabilities.

When a capability is loaded through an authorizing capability that lacks LG-permission, the resulting capability value has its GL(obal) Flag bit cleared. Additionally, if the loaded capability value is *unsealed*, its LG-permission is also cleared in the result.

This permission is similar to the base LM-permission and its effects on loaded capabilities' W-permission and LM-permission (but note the difference in interaction with seals).

LG-permission is a *refinement of* C-permission and R-permission. That is, if either of the latter are clear, then LG-permission has no effect.

9.2. The Capability Global (GL) Flag

The *Capability Global* (GL) flag is a permission-like single-bit field which allows enforcing invariants on

capability propagation in combination with the LG-permission and SL-permission bits described above.

For example, the software TCB may enforce that software has access to capabilities with SL-permission only to (subsets of) its runtime stack, and may ensure that all stack pointers lack GL(obal) Flag. In such a system, capabilities without GL(obal) Flag, including all those derived from the stack, are confined to registers and stack memory. Global capabilities, say, into heap memory, may be attenuated to being local before being passed across a call; the callee will be unable to capture this pointer outside its stack. This specification defines only 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 *Capability Global* flag holds one of two values:

- 1: the capability is *global*.
- 0: the capability is *local*.

As with permissions, the *Capability Global* flag can be cleared when creating a new capability value from an existing one, but it can never be set (without deriving it from a global superset capability).

9.3. Interaction with Root Capabilities

The Root capabilities used in the system are extended thus:

- The definitions of Root Executable and Root Data capabilities are both augmented to require that GL(obal) Flag be set to *global*.
- A Root Executable capability is required to grant LG-permission.
- A Root Data capability is required to grant both LG-permission, and SL-permission.

9.4. Interaction with YPERMC and YPERMR

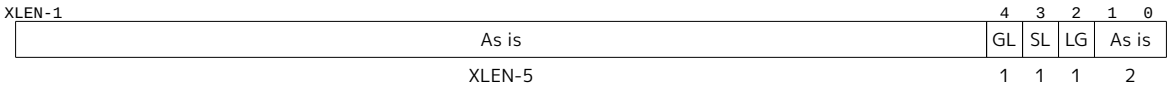


Figure 13. Extended capability permissions bit field (see Figure 6)

The GL(obal) Flag, SL-permission, and LG-permission fields are mapped into the capability permissions bitfield (Figure 6), used by YPERMC and YPERMR, as shown in Figure 13.

9.4.1. YPERMC and the Capability Global (GL) Flag

YPERMC can produce a new capability value with its GL(obal) Flag cleared, even if the source capability is sealed. This is unlike architectural and software permissions. This applies to both "implicit YPERMCs" in loads from memory and explicit YPERMC instructions.

9.4.2. Additional YPERMC rules

As mentioned, the SL-permission and LG-permission permissions are dependent on (refinements of) base permissions. YPERMC (including "implicit YPERMC" operations) and/or the capability encoding therefore clear these permissions when their dependencies clear. Specifically, we add the following rules to those of Section 2.3.10.1:

YPERMC Rule	Permission	Valid only if
Zylevels1-1	LG-permission	C-permission and R-permission
Zylevels1-2	SL-permission	C-permission and W-permission

9.5. Interaction with LY

As outlined above, Zylevels1 introduces two new constraints on capabilities loaded from memory, as part of a LY instruction (`ly rd, offset(rs1)`). Analogous requirements apply for other instructions that inherit semantics from LY. These may be phrased as "implicit YPERMC-s" performed on the loaded capability thus:

- If `rd.tag=1`, `rd` **is not** sealed, and `rs1` does not grant LG-permission, then an implicit YPERMC is performed, clearing both GL(obal) Flag and LG-permission of `rd`.
- If `rd.tag=1`, `rd` **is** sealed, and `rs1` does not grant LG-permission, then an implicit YPERMC is performed, clearing GL(obal) Flag of `rd`.

Missing LG-permission also affects the GL(obal) Flag of sealed capabilities, since notionally the latter is not a permission but rather a data flow label attached to the loaded value.

Because SL-permission is relevant only to capabilities granting W-permission, the attenuation performed by a load whose authority (`rs1`) does not grant LM-permission will necessarily also clear SL-permission.

9.6. Interaction with SY

As outlined above, Zylevels1 introduces a new constraint on capabilities stored to memory, as part of a SY instruction (`sy rs2, offset(rs1)`). Analogous requirements apply for other instructions that inherit semantics from SY. The written capability tag may be set *only if* either

- `rs2` 's GL(obal) Flag is set or
- `rs1` 's SL-permission is set.

While LG-permission attenuates by reducing GL(obal) Flag and LG-permission, SL-permission attenuates by clearing capability tags.

9.7. Interaction with YLT

Implementations of Zylevels1 must ensure that a YLT instruction `ylt rd, rs1, rs2` indicates that `rs1` is a subset of `rs2` (that is, sets `rd` to 1) only if either

- `rs2` 's GL(obal) Flag is clear or
- `rs1` 's GL(obal) Flag is set.

The existing permission subset logic applies to the new SL-permission and LG-permission.

9.8. Interaction with YBLD

A YBLD instruction `ybld rd, rs1, rs2` may yield `rd.tag=1` only if either

- `rs2` 's GL(obal) Flag is clear or
- `rs1` 's GL(obal) Flag is set.

The existing permission subset logic applies to the new SL-permission and LG-permission.

9.9. Interaction with YSUNSEAL

A YSUNSEAL instruction `ysunseal rd, rs1, rs2` must ensure that `rd` grants GL(obal) Flag only if `rs1` also grants GL(obal) Flag. (That is, `rd` grants GL(obal) Flag if and only if both the unsealing authority in `rs1` and the unsealed form of the capability in `rs2` grant GL(obal) Flag.)

The existing permission subset logic applies to the new SL-permission and LG-permission.

9.10. Summary Of System Behavior

Table 44. SL-permission effects for stored capabilities

Auth cap field			Data cap field	
W	C	SL	GL	Notes
1	1	1	X	Store data capability unmodified
		0	1	Store data capability unmodified
			0	Store data capability with capability tag cleared

SL-permission is relevant only to capabilities granting both W-permission and C-permission.

Table 45. GL(obal) Flag effects for loading capabilities

Auth cap field			Data cap field		
R	C	LG	Tag	Sealed	Action
1	1	0	1	Yes	Load data capability with its GL(obal) Flag cleared
				No	Load data capability with both its GL(obal) Flag and LG-permission cleared
		All other cases			Load data capability with both its GL(obal) Flag and LG-permission unmodified

LG-permission is relevant only to capabilities granting both R-permission and C-permission.

Chapter 10. "Zyseal" Extension for CHERI Capability (Un)Sealing

This chapter is not part of the v1.0 ratification package.

10.1. Explicit Sealing and Unsealing Operations

The RVY base architecture defines sealed capabilities. The YBLD, JALR (RVY), and YSUNSEAL instruction and the Zysentry extension allow platforms to build and consume sealed capabilities in particular ways. This extension introduces a more general, intentional (that is, capability-mediated) mechanism for the introduction and elimination of sealed capability forms, in keeping with CHERI's *principle of intentional use*.

This extension first introduces a fundamentally new *kind* of capabilities, "**type** capabilities", whose address space and borne authority range not over *memory* but rather CT-field-s. As subsequently detailed, these capabilities, and their new permissions, will serve as authorizing capabilities to new instructions which perform transformations of other capabilities' CT-field fields:

- Constructing a sealed capability with type `T` from an unsealed capability requires the authority to seal at type `T`, and
- Constructing an unsealed capability from a sealed capability with type `T` requires the authority to unseal at type `T`.

This extension does not define "type conversion" transformations directly between sealed capability types.

10.2. Usable CT-field Values Are Encoding Specified

The capabilities used to mediate (un)sealing are, like memory capabilities, associated with an XLEN-bit address space. However, capability encodings have fewer than XLEN bits devoted to storing CT-field values. As such, encodings will specify what CT-field values can be used to seal capabilities (recall that encodings *must* support representing unsealed capabilities). The remainder of the address space described by type capabilities is available for software use.

10.3. Single Address Space Encodings

Capability encodings are permitted to conflate memory and type address spaces, such that one capability may authorize both memory access to a location and (un)sealing with a type of equal numeric value. Indeed, the encoding of RV32LYmw10rc1pc/RV64LYmw14rc1ps is one such encoding. Ideally, such encodings should permit separate manipulation of (un)sealing permission and memory access permissions, so that software can segregate the address spaces even when the encoding does not do so intrinsically.

10.4. Added Architectural Permissions (AP) Bits

Table 46. Zyseal YPERMC rules.

Permission	Type	Comment
SE-permission	CT-field permission	Grants sealing authority
US-permission	CT-field permission	Grants unsealing authority

Seal Permission (SE)

Permit the bearer to YSEAL capabilities at the in-bound types of this capability.

Unseal Permission (US)

Permit the bearer to YUNSEAL capabilities at the in-bound types of this capability.

10.5. Interaction with YPERMC and YPERMR

Figure 14. Extended capability permissions bit field (see Figure 6)

The SE-permission and US-permission fields are mapped into the capability permissions bitfield (Figure 6), used by YPERMC and YPERMR, as shown in Figure 14.

10.6. Added Instructions

YSEAL

A `yseal rd, rs1, rs2` instruction will use the provided sealing authority of `rs1` to copy the *unsealed* capability in `rs2` into `rd` and seal it with type `rs1.address`, assuming `rs1` has a set capability tag, is in bounds, and grants SE-permission.

YUNSEAL

A `yunseal rd, rs1, rs2` instruction will use the provided unsealing authority of `rs1` to copy the sealed capability in `rs2` into `rd` and unseal it, so long as `rs2.ct = rs1.address`.

10.6.1. YSEAL

Synopsis

Seal a capability using a sealing capability

Mnemonic

```
yseal rd, rs1, rs2
```

Encoding

TODO

Description

Construct, into `rd`, a sealed copy of the unsealed capability in `rs2`, using the type and authority from `rs1`.

Copy `rs2` into `rd`, and then...

1. Clear the capability tag of the capability in `rd` if any of the following hold:
   - `rs2` is sealed (has a non-zero CT-field value)
   - `rs1` has a clear `capability tag`
   - `rs1` does not grant SE-permission
   - The address of `rs1` is out of bounds
   - The address of `rs1` is not a CT-field value that the capability encoding can encode on the capability in `rs2`
2. Set the CT-field of `rd` to the address of `rs1`.

YSEAL uses the (in-bounds) addresss of the authority in `rs1` as the type in the resulting capability. If the authority has a nontrivial range, software can use YADDRW to select which type should be used.

If a capability encoding also entails the presence of sentry capability types, it will be possible for software (bearing suitably permissive capabilities) to seal and unseal the sentry types that that encoding defines. This is deliberate. Software should ensure that the capabilities requisite for such operations are attenuated, confined to sufficiently trusted components, and/or destroyed.

Some capability encodings correlate non-zero CT-field values with other aspects of capabilities, notably permissions. That is, some encodings may be able to represent a valid capability with a given non-zero CT-field only if other properties of that capability hold, such as it granting, or not granting, a particular permission. As such, YSEAL may clear the capability tag of the result in `rd` depending on these other aspects of its `rs2` input, even if some capabilities can be sealed with the type called for by the address of `rs1`.

Included in

Zyseal

Operation

TODO

10.6.2. YUNSEAL

Synopsis

Unseal a capability using an unsealing capability

Mnemonic

```
yunseal rd, rs1, rs2
```

Encoding

TODO

Description

Construct, into `rd`, an unsealed copy of the capability in `rs2`, using the type and authority from `rs1`.

Copy `rs2` into `rd`, and then...

1. Clear the capability tag of the capability in `rd` if any of the following hold:
   - `rs1` has a clear `capability tag`
   - `rs1` does not grant US-permission
   - The address of `rs1` is out of bounds
   - The address of `rs1` is not equal to the CT-field of the capability in `rs2`.
2. Propagate permissions from `rs1` onto `rd`:
   - If the Zylevels1 extension is implemented, and the capability in `rs1` does not grant GL(obal) Flag, use the semantics of the YPERMC instruction to clear the GL(obal) Flag of the capbility in `rd`.

     (That is, the resulting capability in `rd` will grant GL(obal) Flag if and only if the capabilities in `rs1` and `rs2` both grant GL(obal) Flag.)
   - Other extensions may impose similar constraints.
3. Set the CT-field of the capability in `rd` to zero.

YUNSEAL requires exact equality of the authority's type, `rs1.address`, and the to-be-unsealed capability's type, `rs2.ct`. If it is desirable to unseal one of several capability types, using an authority with nontrivial range, software can use YTYPER and YADDRW to make these values match. Future extensions may specify a fused "copy type" operation, as was present in the CHERI v9 ISA.

Included in

Zyseal

Operation

TODO

Chapter 11. "Zybndsrdw" Extension for Bounding to Representable Lengths

This chapter is not part of the v1.0 ratification package.

This extension adds a single instruction, YBNDSRDW, which writes capability bounds while rounding *down* to ensure precise representability of the result. In contrast to YBNDSRW, it does not alter the requested *lower* bound (capability base).

This instruction is useful when exposing (byte) buffers across trust domains. Given a buffer, an address therein, and a length after that address of elements to be revealed to a different trust domain, software in the originating trust domain wishes to compute the largest span, starting at the cursor and up to the length of elements to be shared, that can be precisely represented with a capability. Rounding the base down and/or the length up (as with YBNDSRW) risks exposing buffered data in the buffer not suitable for exposure to the different trust domain in question. While the originating domain could instead make a series of exposures, each sufficiently small to be guaranteed to be representable, it is a significant improvement to allow capability representation itself to determine the largest safe exposure size.

11.1. YBNDSRDW

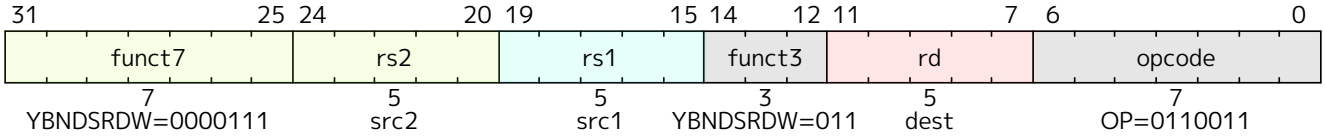
Synopsis

Write capability bounds, rounding down if required

Mnemonic

```
ybndsrdw rd, rs1, rs2
```

Encoding



Description

Copy the capability from register `rs1` to register `rd`. Set the base address of its bounds to the value of `rs1.address`.

Round *down* the requested length, in `rs2[XLEN-1:0]`, by the smallest amount necessary to guarantee that it is precisely representable given this base address.

Set `rd.tag=0` if `rs1.tag=0`, `rs1` is sealed or if `rd` 's bounds exceed `rs1` 's bounds.

Set `rd.tag=0` if `rs1` fails any integrity checks.

If the result in `rd` has its capability tag set to one, then its base is `rs1.address`, as with YBNDSW. The result in `rd` will be zero-length if and only if `rs2[XLEN-1:0]` is zero. In practice, capability encodings offer byte-granularity of bounds for capabilities of lengths up to some threshold between a few hundred and a few thousand bytes, depending on the encoding; for requested lengths below this threshold, no rounding is required, and otherwise this threshold serves to guarantee a minimum length return from YBNDSRDW.

YBNDSRDW finds the largest length `l` that is both less than or equal to the requested `rs2[XLEN-1:0]` and precisely representable given a particular base address, `b`. A length `l` is precisely representable given base `b` if a YBNDSW instruction...

- *whose `rs2` register holds `l` and*
- *whose `rs1` register holds a capability...*

— whose address is `b` and

— whose bounds cover the entire address space

produces a result in `rd` whose capability tag is set to one. This value may be efficiently found, for most capability encoding schemes, by counting trailing zeros in the desired base address and computing the length that is a maximal mantissa shifted left by that count (that is, using the number of trailing zeros in the base address as the value's exponent).

Included in

Zybndsrdw

Operation

TODO

Chapter 12. RVY Specializations for Microcontroller Systems

This chapter is not part of the v1.0 ratification package.

12.1. The Zycheriot Unprivileged ISA Extension

This section defines a series of small changes to the RVY and RVYE unprivileged base architectures that serve to specialize it for microcontroller environments. These changes are based on, but are not exactly isomorphic to, the prior CHERIoT RV32E-based ISA.

Some of these changes are intended to work in concert with their privileged counterparts found in the Smycheriot extension.

12.1.1. Required Extensions

Zycheriot assumes the presence of both the Zylevels1 and Zyseal extensions. The present specification presumes the **absence** of both the Zyhybrid and Zysentry extensions.

While Zycheriot is nominally compatible with Zyhybrid, and particular instantiations may opt to permit disabling CHERI, we have not yet found a compelling reason to formally specify this composition.

While Zycheriot is nominally compatible with Zysentry, the operating system written for CHERIoT has a security model that presumes the absence of ambient sealing, and so this specification does not define any ambiently available sentry types.

12.1.2. Refining CHERI Capabilities

12.1.2.1. Software Defined Permissions

Zycheriot defines SDPLEN, the number of software-defined permissions, to be `1`. We denote this one user permission as `U0`.

12.1.2.2. Root Permission Sets

Zycheriot defines *three* Root capability values, each of which has a set capability tag, is unsealed, has bounds that span its associated address space, and has GL(obal) Flag set. That is, they differ only in their granted *permissions*, thus:

- Its Root Executable capability value grants exactly all of X, R, C, LM, ASR, and LG.
- Its Root Data capability value grants exactly all of R, W, C, LM, LG, and SL.
- It defines a *root sealing* capability grants exactly all of SE, US, and UO.

This set of root capabilities satisfies the following properties, by construction. Any further extension which adds or modifies capability permissions must ensure that its revised or additional root capabilities do so as well. Since all capabilities in the system must trace their provenance back to a root capability, these properties will necessarily remain true through any series of YPERMC transitions.

w-nand-x
: At most one of X or W may be set.

mem-nand-ct

The SE, US, and UO collectively conflict with either of the R or W permissions.

That is, a capability may grant permissions from at most one of these two sets; this serves to partition capabilities that refer to memory addresses from those that refer to CT-field values (those granting SE and/or US) or uninterpreted integers (those granting UO). Capabilities granting no permissions from the union of these sets are not distinguished.

12.1.2.3. Permission Transition Constraints

Zycheriot requires a RVY base that is using a compressed permission scheme (that is, one has its `LY p` parameter set to `pc`).

In addition to the constraints on permission transitions defined across the base RVY ISA and the Zylevels1 extension, Zycheriot adds two additional constraints, shown in Table 47.

Table 47. Zycheriot Permission Transition Rules

YPERMC Rule	Permission	Valid only if
Zycheriot-1	X-permission	C-permission and R-permission
Zycheriot-2	SL-permission	R-permission

These constraints enable more compression of capability permissions by disallowing the least useful permission combinations. In particular, ABIs usually require code to be readable to enable PC relative access to constant data, and write-only store-local capabilities are not required given that store-local is generally used only for thread stacks.

12.1.2.4. Capability Types

Zycheriot introduces several kinds of sentry capabilities, imposing a degree of control flow integrity, by giving architectural semantics to several CT-field values. The presence of, and handling constraints for, these sentry types are important to unprivileged software. In particular, Zycheriot uses *five* CT-field values (1 through 5, inclusive) to modify the behavior of JALR (RVY):

- All of these values are defined to be sentry capability values. Thus, sealed capabilities with any of these CT-field values may be passed as inputs to JALR (RVY) and will be unsealed prior to installation into pc.
- Two values, 4 and 5, are used by JALR (RVY) and JAL (RVY) to seal the return capability when `rd` is `ra`. The choice between the two depends on privileged machine state. If `rd` is not `ra`, then the return capability is not sealed.
- JALR (RVY) is given conditional behavior based on the register selectors used, as detailed in Table 48. Prohibited combinations of register selectors and CT-field value will cause the target pc to have a clear capability tag and so raise a CHERI Instruction Access Fault.

In light of the last rule, we call capabilities granting X-permission and...

- with a CT-field value of 1, 2, or 3 "forward" sentries.
- with a CT-field value of 4 or 5 "backward" sentries.

The Smycheriot extension to *privileged* architecture draws further distinction between the various forward and backward types.

Table 48. JALR (RVY) Conditional Behavior

rs1	rd	Permitted rs1 CT-field-s	Comments
any	`ra`	0, 1, 2, 3	Function call
`ra`	`null`	4, 5	Function return
other operands		0, 1	Non-standard control flow (e.g., tail calls)

All of the sentry capability CT-field values defined herein need be encodable only if the capability grants X-permission, as required by JALR (RVY) of the capability to be installed into pc. Zycheriot imposes no architectural requirement for, or semantics upon, capabilities that do not grant X-permission and have one of these CT-field values.

The use of different sentry capability CT-field values for "forward" control flow arcs (in which JALR (RVY) writes a capability) and "backward" arcs (in which it does not) means that functions cannot return "backwards" to a (forward) function pointer and, dually, that a function pointer cannot be subsituted for a return pointer.

12.2. An RV32LYmw9e14r0as11pc Common Base Architecture

It is intended that Zycheriot and Smycheriot be instantiated (in tandem) atop a RV32Y base architecture that is at least as expressive as LYmw9e14r0as11pc and in combination with one of the capability encoding schemes discussed below (or a future, possibly vendor-specific, capability encoding). Software written for this common subset architecture will be source compatible across encoding formats and with more expressive variants of RV32Y.

The parameters set by this common architecture are:

- `mw9` and `e14`: A mantissa width of 9 and expressibility of capability exponents from 0 (inclusive) to 14 (inclusive). Together, these ensure that capabilities can precisely capture capabilities whose lengths are between 1 and 2^9 - 1 times a power of 2 between 1 (inclusive) and 2^{14} (inclusive) or is 2^{24}.

  The latter of these is sufficient to represent the entire 32-bit address space at a granularity (that is, bounds alignment) of 16MiB.

- `r0`: Software must not presume that taking a capability out of bounds, other than the "one past the end" position required by C, will leave it tagged.
- `as11`: The shift used by AUIPC (RVY) (and similar instructions) is `11`.
- `pc`: The set of capabiality permissions that can be expressed is subject to compression. For these bases, the only compression used is a consequence of the roots and permission transition rules found in the base architecture and present extensions (such as Zycheriot).

TODO: Can we refactor RV32Y's compression scheme out to something like the Zycheriot ISA perspective, too?

As a reminder, these parameters constrain the behavior of RVY instructions such as YBNDSW and YPERMC.

12.3. The RV32LYenccheriot1 CHERI Capability Encoding Scheme

This section describes an in-memory format and properties of a capability encoding intended for RV32Y. It is specifically designed to be a suitable substrate for RV32Y_Zycheriot systems.

This *is* the encoding used for the existing CHERIoT ISA. Compared to RV32LYmw10rc1pc it:

1. supports more precise bounds for a given capability length using slightly simpler hardware
2. adds features for building rich compartmentalization, such as extra permissions and sealing types.

The first of these is achieved by:

- Using two extra bits for the bounds encoding
- Giving the exponent its own field instead of embedded it in the T and B fields
- Saving space by reducing the exponent in size by one and dropping support for exponents in the range 15 to 23. This limits the precision for the bounds of large capabilities (>= 8MiB).
- dropping guaranteed out-of-bounds representability (except "one past the end")

These tradeoffs are aimed at microcontroller implementations where precise bounds on small capabilities are important to save memory on padding, large capabiliites are unlikely and microarchitectural complexity should be minimised.

12.3.1. Capability Encoding

The components of a capability, except the capability tag, are encoded as shown in Figure 15.

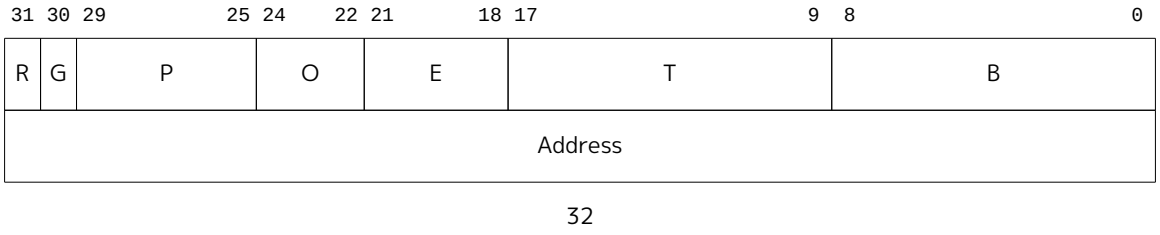


Figure 15. Capability encoding for RV32LYenccheriot1

Field	Description
R	Reserved bit
G	Global bit, as in GL.
P	Permissions, see CHERIoT permission encoding.
CT	Sealed object Type, see CHERIoT Capability Type (CT) Encoding.
E	Exponent field for bounds encoding, see CHERIoT Capability Bounds Encoding.
T	Top field for bounds encoding, see CHERIoT Capability Bounds Encoding.
B	Base field for bounds encoding, see CHERIoT Capability Bounds Encoding.
Address	The address field as per Figure 1.

The reserved bit (the R field) is defined to be 0 in the encoding of the root capabilities given below, and is not mutable by any instruction defined in RVY, and so will necessarily be 0 in all valid capabilities derived therefrom. It is therefore available for use by suitable extensions or derivative encodings.

The encoding impacts the following CHERI extensions:

Zyhybrid

This encoding does not support Zyhybrid: there is no mode bit so only purecap mode is supported.

Zylevels1

This encoding supports Zylevels1. Its GL flag is encoded in the G bit, and its GL and SL permissions are encoded in the P field.

Zyseal

This encoding supports Zyseal. The SE and US permissions are encoded in the P field.

12.3.1.1. Capability Encoding Parameter Summary

This encoding is suitable for use with RV32Y base architectures whose parametric requirements are no stronger than those given in Table 49. While not strictly required, it is recommended to use this encoding with RV32Y base architectures whose AUIPC shift (as) RVY parameter is 11 or smaller. Known extension (in)compatibilities are listed in Table 50.

Table 49. RV32LYenccheriot1 RVY Parameter Values

Parameter	Value	Comment
mw	9	Mantissa width
e	14	Exponent limit before jumping to maximum exponent
rc	0	Representable region between base and top only
p	pc	Compressed permission encoding
enc	Cheriot1	Encoding variant

Table 50. RV32LYenccheriot1 extension summary

Extension	Comment
Zyhybrid	Not supported (M-bit not defined)
Zysentry	Not compatible without other extensions (no ambient CT-field values defined herein)
Zylevels1	Compatible (recommended)
Zyseal	Compatible (recommended)
Zycheriot	Compatible (recommended)
Smycheriot	Compatible (recommended)
Zybld	Compatible (will build exclusively unsealed capabilities without other extensions)
All RVY versions of other standard extensions	Compatible if the extension is compatible with RV32E

Table 51. RV32LYenccheriot1 Feature summary

Feature	Comment
Representable region	Between base and top only (inclusive)
Permission encodings	Not all combinations can be represented

12.3.1.2. Permissions Encoding

Similarly to RV32LYmw1Orc1pc, capability permissions are encoded in a compressed form. The encoding herein exactly reflects the Root Permission Sets and Permission Transition Constraints of Zycheriot.

The permission encoding space is split into quadrants using P[3:2]. Each quadrant may include some fixed permissions (indicated with ✔) and some dependent permissions encoded using P[2:0].

- Quadrant 0 is used to encode permissions that authorize sealing (see Zyseal) and also the single software-defined permission, UO. It can also encode 'no permissions'.
- Quadrant 1 encodes executable capabilities along with the dependent permission ASR and optional permissions LM and LG.
- Quadrant 2 is subdivided into octants:
  - Octant 4 encodes combinations of R and W without C with the redundant not-R and not-W used to encode write-only with C.
  - Octant 5 encodes read-only capabilities with C and the dependent permissions LM and LG.
- Quadrant 3 encodes permissions with R, W, C and the dependent permissions SL, LM and LG. The meaning for bits [2:0] are shown in CHERIoT permission encoding.

P[4:0]				Decoded Permissions											
P[4:3]	P[2]	P[1]	P[0]	R	W	C	SL	LM	LG	X	ASR	UO	SE	US	Notes
00	UO	SE	US									P[2]	P[1]	P[0]	Sealing
01	ASR	LM	LG	✔		✔		P[1]	P[0]	✔	P[2]				Executable
10	0	0	0		✔	✔									Cap WO
10	0	R	W	P[1]	P[0]										Data RW (R and/or W)
10	1	LM	LG	✔		✔		P[1]	P[0]						Cap RO
11	SL	LM	LG	✔	✔	✔	P[2]	P[1]	P[0]						Cap RW

For example, if P[4:3] = 01 and P[2:0] = 101 then the decoded permissions are R, C, X, LG, and ASR. Note that there is no encoding for an "Infinite" capability with all permissions. In particular there is no overlap between the software and sealing permissions (`UO`, SE, US) and any other permissions (recall mem-nand-ct from Root Permission Sets) and X is mutually exclusive with W (recall w-nand-x).

To encode a set of permissions resulting from YPERMC, the first of the following rules to apply is used:

1. If the permissions include X, R and C, then encode ASR, LM and LG using the executable format (P[4:3] = 01).
2. If the permissions include R, W and C then encode SL, LM and LG using the Cap RW format (P[4:3] = 11).
3. If the permissions include R and C then encode LM and LG using the Cap RO format (P[4:2] = 101).
4. If the permissions include W and C then encode using the Cap WO format (P[4:0] = 10000).
5. If the permissions include R or W then encode using the Data RW format (P[4:0] = 100RW).
6. Encode UO, SE and US using the sealing format (P[4:3] = 00).

This procedure will automatically apply the rules defined in Permission Transition Constraints to legalize permission sets. If any of the requested permissions cannot be represented using the chosen format, then the they are dropped. For example, if the requested permissions are R, LG and X, then rule 5 applies and the resulting permissions will be just R.

The three Root capabilities defined by Zycheriot have permissions encodings as shown in CHERIoT root capabilities table.

Root name	P[4:0]	Permissions
Sealing	00111	U0, SE, US
Root Executable	01111	X, R, C, LM, LG, ASR
Root Data	11111	W, R, C, LM, LG, SL

12.3.1.3. Capability Type (CT) Encoding

Capabilities in this encoding have a 3-bit field for CT-field values. This is used to encode 15 different sealing types by distinguishing between executable and non-executable sealed capabilities as follows:

X	CT Value	Decoded Type
1	000	0 (Unsealed)
1	1 .. 7	1 .. 7
0	000	0 (Unsealed)
0	1 .. 7	9 .. 15 (CT + 8)

Attempts to seal a capability with a type not compatible with its X-permission value will yield a result with a clear capability tag.

Recall that CT-field values 1 (inclusive) through 5 (inclusive) are given semantics if the Zycheriot extension is present.

12.3.1.4. Bounds (E, B, T) Encoding

The bounds are encoded in similar, but slightly simplified way to RV64LYmw14rc1ps, with field B and T being substituted into the address at the offset given by the exponent to form the base and top. Rather than using an EF field to indicate whether the exponent is zero or contained in the low bits of B and T the exponent is stored in its own 4-bit field, E. Since this is not large enough to store the maximum exponent of 24 required to cover the full 32-bit address space, an E value of 15 decodes as 24 with exponents 15 to 23 being invalid. The decoded exponent, e, is given by:

```
e = (E == 15) ? 24 : E
```

This means that capabilities up to nearly 8 MiB are represented with alignment requirements increasing in powers of two for increasing exponents up to 14, and all larger capabilities must have bounds aligned to 16 MiB. RV32LYenccheriot2 describes an encoding variant that removes this limitation.

For a given exponent the base and top are then computed as:

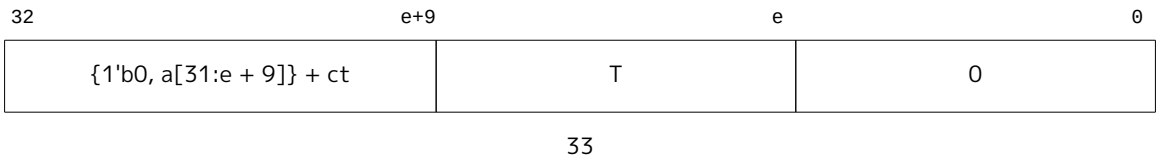


Figure 16. Decoding of the XLEN+1 wide top (t) bound

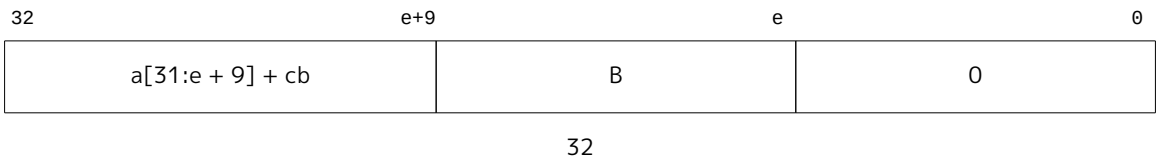


Figure 17. Decoding of the XLEN wide base (b) bound

where the corrections, c_t and c_b, are given by:

```
A_hi = (A[e + 8 : e]) < B ? 1 : 0
T_hi = (T < B) ? 1 : 0
c_b = -A_hi
c_t = A_hi - T_hi
```

These corrections work by assuming that the base is in the lower of two 2^{e+9} aligned regions, and that the top and address are always greater than or equal to the base but may be in the higher region provided they are within 2^{e+9} of base. These assumptions lead to the representable region being given by:

```
base <= address < base + (1 << (e + 9))
```

This must be checked by all operations that change the capability address. If this check fails the resulting capability will have its capability tag cleared. Note that this means that it is not possible to represent a capability with an address less than the base. Depending on the size of the capability some addresses above top may be representable, but in the worst case the highest representable address is equal to top (one byte beyond the end of the dereferenceable region).

12.3.1.4.1. Encoding bounds

When YBNDSW is used to set the bounds of a capability the E, B and T fields are computed from the desired base and length as follows:

```
// compute candidate exponent
e = 23 - count_leading_zeros(length[31 : 9])
if e > 14 then {
  e = 24
}

// extend base and top to XLEN+1 bits
base33 = 1b0 @ base
top33 = base33 + (1b0 @ length)

// extract 10-bit from base and top
b = base33[e + 9 : e]
t = top33[e + 9 : e]

// round up top if low bits are truncated
if top33[e - 1 : 0] != 0 {
  t = t + 1
```

```
}

// in case this caused length overflow use the next exponent
if t - b >= 1 << (e + 9) {
  e += 1
  if e > 14 then {
    e = 24
  }
  b = base33[e + 9 : e]
  t = top33[e + 9 : e]
  if top33[e - 1 : 0] != 0 {
    t = t + 1
  }
}

// encode E
E = (e == 24) ? 15 : e
// truncate B and T to 9 bits
B = b[8 : 0]
T = t[8 : 0]
```

12.3.2. Encoding of Special Capabilities

12.3.2.1. NULL Capability Encoding

The NULL capability is represented with 0 in all fields. This implies that it is unsealed, has no permissions and its exponent, base and top are 0.

Table 52. Field values of the NULL capability

Field	Value	Comment
Capability Tag	zero	Capability is not valid
R	zero	Reserved bit
G	zero	Not global
P	zeros	Grants no permissions
CT	zeros	Unsealed
E	zeros	Exponent
T	zeros	Top address bits
B	zeros	Base address bits
Address	zeros	Capability address

12.3.2.2. Root Capability Encoding

The encoding for the Root Executable and Root Data capabilities defined by RVY and the sealing root defined by Zycheriot have bounds that cover the entire associated address space. Root capabilities necessarily have set capability tags and are necessarily unsealed, Zylevels1 requires root capabilities to be Global, and we define our root capabilities as having a clear reserved bit (that is, an R field of `0`). The

encoded fields of root capabilities are shown in Table 53, except the P field, which depends on which root capability is being represented.

Table 53. Bounds field values of root capabilities

Field	Value	Comment
Capability Tag	one	Capability is valid
R	zero	Reserved bit
G	1	Global
P	XXXXX	Varies as per CHERIoT root capabilities table
CT	zeros	Unsealed
E	0xf	Maximum exponent
T	0x100	top = 2^{XLEN}
B	0x000	base 0
Address	zeros	Capability address

12.4. The RV32LYenccheriot2 CHERI Capability Encoding Scheme

This chapter describes a variation on the RV32LYenccheriot1 encoding to support all exponents using the same number of bits. This change is backwards compatible for software but enables support for more precise bounds on capabilities larger than 8MiB. Due to increased microarchitectural complexity and the limited need for precise alignment on such large capabilities on small systems, implementations may choose to support either encoding.

The changes are limited to the bounds encoding; all other aspects of the capability encoding are identical to RV32LYenccheriot1.

12.4.1. Capability Encoding

The components of a capability, except the capability tag, are encoded as shown in Figure 18.

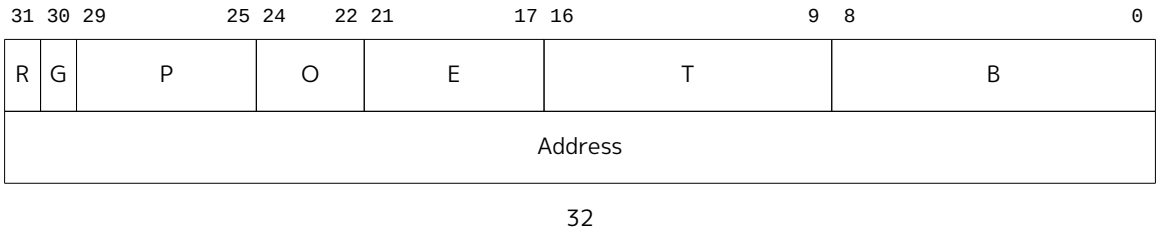


Figure 18. Capability encoding for RV32LYenccheriot2

The only difference from RV32LYenccheriot1 is that the exponent field, E, is grown to 5 bits and the T field is shrunk to 8 bits. This allows all exponents from 0 to 24 to be represented directly, with some spare exponents available for other uses. To enable the reduced T field to be decoded to a 9-bit value it is assumed that the top bit of the length, L = T - B, is one, similar to normalised floating point numbers. To encode small capabilities with exponent zero and lengths less than 256 bytes the special E value, 31, is used to mean L[8] is zero. T[8] can then be reconstructed as follows:

```
T[8] = B[8] XOR (E == 31 ? 0 : 1) XOR (T[7:0] < B[7:0])
```

The unused exponent values (25 .. 30) are reserved for future use. Other aspects of bounds decoding remain identical to CHERIoT Capability Bounds Encoding.

Encoding the bounds for a requested base and length is similar to before but without the special cases for exponents greater than 15.

```
// compute candidate exponent
e = 23 - count_leading_zeros(length[31:9])

// extend base and top to XLEN+1 bits
base33 = 1b0 @ base
top33 = base33 + (1b0 @ length)

// extract 10-bit from base and top
b = base33[e + 9 : e]
t = top33[e + 9 : e]

// round up top if low bits are truncated
if top33[e - 1 : 0] != 0 {
  t = t + 1
}

// in case this caused length overflow use the next exponent
if t - b >= 1 << (e + 9) {
  e += 1
  b = base33[e + 9 : e]
  t = top33[e + 9 : e]
  if top33[e - 1 : 0] != 0 {
     t = t + 1
  }
}

// truncate B and T to 9 bits
B = b[8 : 0]
T = t[8 : 0]
// encode E
E = (e == 0 && (T - B)[8] == 0) ? 31 : e
```

12.4.1.1. Capability Encoding Parameter Summary

This encoding is suitable for use with RV32Y base architectures whose parametric requirements are no stronger than those given in Table 54.

Table 54. RV32LYenccheriot2 parameter summary

Parameter	Value	Comment
mw	9	Mantissa width
e	a	All exponents are supported

Parameter	Value	Comment
rc	0	Representable region between base and top only
p	pc	Compressed permission encoding
enc	Cheriot2	Encoding variant

12.5. The RV32LYenccheriot3 CHERI Capability Encoding Scheme

This chapter describes a further variation on the RV32LYenccheriot2 encoding. It is equivalent in terms of representable bounds, supported features and the number of bits used, but has different microarchitectural properties. The only observable difference for software is the encoded metadata bits of capabilities in stored in memory, which is only relevant to specialised system software such as debuggers. As such, the decision of which to use should be based on microarchitectural requirements. It is included here in order to provide flexibility to implementations.

12.5.1. Capability Encoding

The components of a capability, except the capability tag, are encoded as shown in Figure 19.

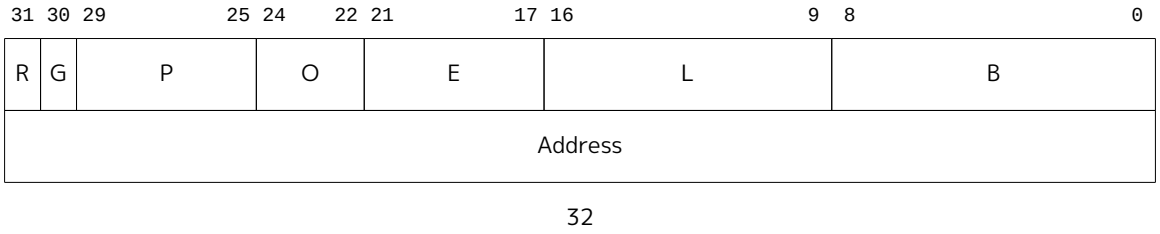


Figure 19. Capability encoding for RV32LYenccheriot3

The only difference from RV32LYenccheriot2 is that the top field, T, is replaced by a length, L. The 8-bit L is expanded to 9 bits by adding a leading implicit one bit, except in the case of the special exponent 31 where L[8] is zero. Bounds decoding can then proceed as follows:

```
// Decode L8 and exponent
e = (E == 31) ? 0 : E
L8 = (e == 31) ? 0 : 1

// Decode base (same as <<rv32y_cheriot_encoding1_name>>)
c_b = (A[e + 8 : e]) < B ? -1 : 0
base = ((A[31 : e + 9] + c_b) @ B) << e

// Compute top from base and length
top = base + ((L8 @ L) << e)
```

All other aspects of the capability encoding are identical to RV32LYenccheriot2.

12.5.1.1. Capability Encoding Parameter Summary

This encoding is suitable for use with RV32Y base architectures whose parametric requirements are no stronger than those given in Table 54. These are identical to those of RV32LYenccheriot2 except for the

encoding variant parameter.

Table 55. RV32LYenccheriot3 parameter summary

Parameter	Value	Comment
mw	9	Mantissa width
e	a	All exponents are supported
rc	0	Representable region between base and top only
p	pc	Compressed permission encoding
enc	Cheriot3	Encoding variant

Appendix A: CHERI (RV64Y) Unprivileged Appendix

A.1. RVY ISA Extension Summary

An RVY core imports *all* instructions from RVI and adds new instructions for CHERI functionality. Additionally, some RVI instruction (as well as instructions defined in other extensions) have modified behavior. The following sections detail the list of added/modified instructions per extension.

A.1.1. RVY added instructions

Table 56. RVY added instructions

Mnemonic	RV32Y	RV64Y	Function
LY	✓	✓	Load capability
SY	✓	✓	Store capability
YADD	✓	✓	Capability pointer increment
YADDI	✓	✓	Capability pointer increment by immediate
YADDRW	✓	✓	Write capability address
YTAGR	✓	✓	Read capability tag
YPERMR	✓	✓	Read capability permissions
YMV	✓	✓	Capability register copy
YPERMC	✓	✓	Clear capability permissions
SRLIY	✓	✓	Logical right shift of Y register
YHIR	✓	✓	Read capability metadata (pseudo)
PACKY	✓	✓	Pack Y register
YHIW	✓	✓	Write capability metadata and clear capability tag (pseudo)
SYEQ	✓	✓	Capability equality comparison including capability tag
YLT	✓	✓	Capability less than comparison including capability tag
YSUNSEAL	✓	✓	Unseal by superset reconstruction
YBNDSW	✓	✓	Write capability bounds
YBNDSWI	✓	✓	Write capability bounds by immediate
YBNDSRW	✓	✓	Write capability bounds, rounding up if required
YAMASK	✓	✓	Capability alignment mask
YBASER	✓	✓	Read capability base address
YLENR	✓	✓	Read capability length
YTYPER	✓	✓	Read capability type

A.1.2. RVI (RVY modified behavior)

The following RVI instructions have *modified* behavior due to adding CHERI functionality. In general, this is restricted to changing whether input/output operands read/write XLEN or YLEN bits.

Table 57. RVI (RVY modified behavior) instructions

Mnemonic	RV32Y	RV64Y	Function
AUIPC (RVY)	✔	✔	Add upper immediate to pc
JAL (RVY)	✔	✔	Immediate offset jump, and link and seal to capability register
JALR (RVY)	✔	✔	Jump to capability register, and link and seal to capability register

A.1.3. Zicsr (RVY modified behavior)

The following RVI instructions have *modified* behavior due to adding CHERI functionality. In general, this is restricted to changing whether input/output operands read/write XLEN or YLEN bits.

Table 58. Zicsr (RVY modified behavior) instructions

Mnemonic	RV32Y	RV64Y	Function
CSRRW (RVY)	✔	✔	CSR write
CSRRS (RVY)	✔	✔	CSR set
CSRRC (RVY)	✔	✔	CSR clear
CSRRWI (RVY)	✔	✔	CSR write (immediate form)
CSRRSI (RVY)	✔	✔	CSR set (immediate form)
CSRRCI (RVY)	✔	✔	CSR clear (immediate form)

A.1.4. Zysentry

Zysentry adds the YSENTRY instruction.

Table 59. Zysentry instruction extension

Mnemonic	RV32Y	RV64Y	Function
YSENTRY	✔	✔	Seal capability as a sentry

A.1.5. Zybld

Zybld adds the YBLD instruction.

Table 60. Zybld instruction extension

Mnemonic	RV32Y	RV64Y	Function
YBLD	✔	✔	Build capability

A.1.6. Zytopr

Zytopr adds the YTOPR instruction.

Table 61. Zytopr instruction extension

Mnemonic	RV32Y	RV64Y	Function
YTOPR	✔	✔	Read capability top address

A.1.7. Zybndsrdw

Zybndsrdw adds the YBNDSRDW instruction.

Table 62. Zybndsrdw instruction extension

Mnemonic	RV32Y	RV64Y	Function
YBNDSRDW	✔	✔	Write capability bounds, rounding down if required

A.1.8. C (RVY added instructions)

These 16-bit instructions are supported by any RVY core that includes the standard C extension. They are incompatible with Zcf (RV32) and Zcd (RV64).

Table 63. C (RVY added instructions) instruction extension

Mnemonic	RV32Y	RV64Y	Function
C.LYSP	✔	✔	Load capability stack pointer relative, 16-bit encoding
C.SYSP	✔	✔	Store capability stack pointer relative, 16-bit encoding
C.LY	✔	✔	Load capability, 16-bit encoding
C.SY	✔	✔	Store capability, 16-bit encoding

A.1.9. RV32 / RV32Y RVC load/store mapping summary

Table 64. 16-bit load/store instruction mapping in RV32I

Encoding		Supported Extensions				
[15:13]	[1:0]	Zca	Zcf	Zcd	Zcmp/ Zcmt	Zclsd
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	N/A
001	00	N/A	N/A	C.FLD	reserved	N/A
101	10	N/A	N/A	C.FSDSP	Zcmp/ Zcmt	N/A
001	10	N/A	N/A	C.FLDSP	reserved	N/A

Table 65. 16-bit load/store instruction mapping in RV32Y

Encoding		Supported Extensions		
[15:13]	[1:0]	Zca	Zcd	Zcmp/ Zcmt
111	00	C.SY		
111	10	C.SYSP		
011	10	C.LYSP		
011	00	C.LY		
101	00	N/A	C.FSD	reserved
001	00	N/A	C.FLD	reserved
101	10	N/A	C.FSDSP	Zcmp (RV32Y)/ Zcmt (RV32Y)
001	10	N/A	C.FLDSP	reserved

Zcf and Zclsd are incompatible with RV32Y.

A.1.10. RV64 / RV64Y RVC load/store mapping summary

Table 66. 16-bit load/store instruction mapping in RV64I

Encoding		Supported Extensions		
[15:13]	[1:0]	Zca	Zcd	Zcmp/ Zcmt
111	00	C.SD	N/A	N/A
011	00	C.LD	N/A	N/A
111	10	C.SDSP	N/A	N/A
011	10	C.LDSP	N/A	N/A
101	00	N/A	C.FSD	reserved
001	00	N/A	C.FLD	reserved
101	10	N/A	C.FSDSP	Zcmp/ Zcmt
001	10	N/A	C.FLDSP	reserved

Table 67. 16-bit load/store instruction mapping in RV64Y

Encoding		Supported Extensions
[15:13]	[1:0]	Zca
111	00	C.SD
011	00	C.LD
111	10	C.SDSP
011	10	C.LDSP
101	00	C.SY
001	00	C.LY
101	10	C.SYSP
001	10	C.LYSP

Zcd, Zcmp and Zcmt are incompatible with RV64Y.

A.1.10.1. C.LY

see C.LYSP.

A.1.10.2. C.LYSP

Synopsis

Capability loads (C.LY, C.LYSP), 16-bit encodings

These instructions have different encodings for RV64 and RV32.

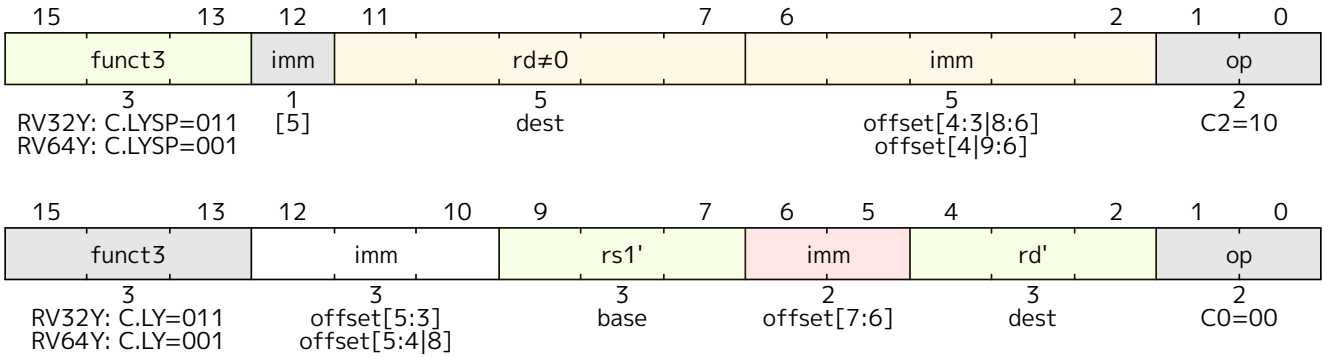
Mnemonics

```
c.ly rd', offset(rs1')
c.lysp rd', offset(sp)
```

Expansions

```
ly rd', offset(rs1')
ly rd', offset(sp)
```

Encoding



(CHERI) Capability Mode Description

Load capability instruction, authorized by the capability in `rs1`. Take a load address misaligned exception if not naturally aligned.

Exceptions

Exceptions occur when the authorizing capability fails one of the checks listed below:

Kind	Reason
CHERI Load Access Fault	Authorizing capability tag is set to 0.
CHERI Load Access Fault	Authorizing capability is sealed.
CHERI Load Access Fault	Authorizing capability does not grant the necessary permissions. Only R-permission is required.
CHERI Load Access Fault	At least one byte accessed is outside the authorizing capability bounds, or the bounds could not be decoded.
CHERI Load Access Fault	Authorizing capability failed any integrity check.

Prerequisites

C or Zca, RVY

Included in

C (RVY added instructions)

Operation (after expansion to 32-bit encodings)

See LY

A.1.10.3. C.SY

see C.SYSP.

A.1.10.4. C.SYSP

Synopsis

Capability stores (C.SY, C.SYSP), 16-bit encodings

These instructions have different encodings for RV64 and RV32.

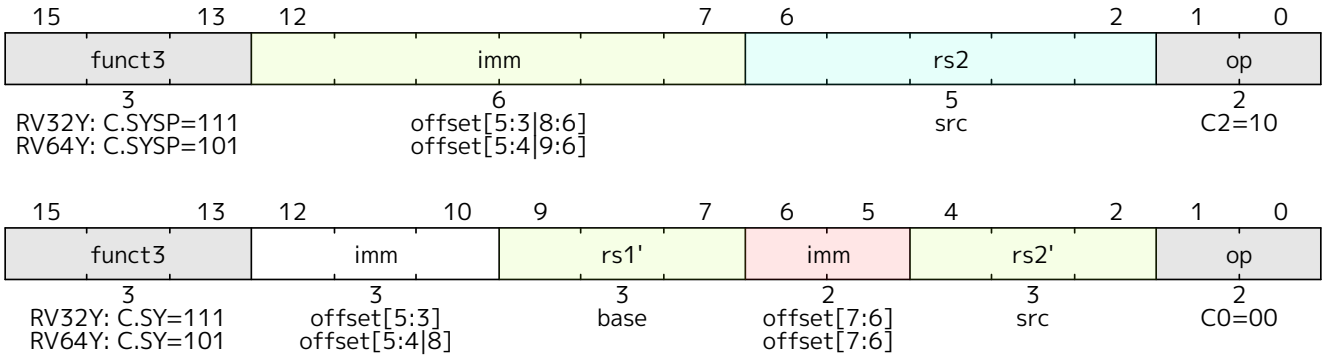
Mnemonics

```
c.sy rs2', offset(rs1')
c.sysp rs2', offset(sp)
```

Expansions

```
sy rs2', offset(rs1')
sy rs2', offset(sp)
```

Encoding



***(CHERI) Capability Mode* Description**

Store the YLEN-bit value in `rs2'` to memory. The capability in `sp` authorizes the operation. The effective address of the memory access is obtained by adding the address of `sp` to the zero-extended offset.

Capability Tag of the written capability value

The capability written to memory has the capability tag set to 0 if the capability tag of `rs2'` is 0 or if the authorizing capability (`sp`) does not grant C-permission.

Extensions may define further circumstances under which stored capabilities may have their capability tags cleared.

This instruction can propagate valid capabilities which fail integrity checks.

Exceptions

Store/AMO access fault exception when the effective address is not aligned to YLEN/8.

Store/AMO access fault if the stored capability tag is set to one and the PMA is *CHERI Capability Tag Fault*.

Exceptions occur when the authorizing capability fails one of the checks listed below:

Kind	Reason
CHERI Store/AMO Access Fault	Authorizing capability tag is set to 0.
CHERI Store/AMO Access Fault	Authorizing capability is sealed.
CHERI Store/AMO Access Fault	Authorizing capability does not grant the necessary permissions.
CHERI Store/AMO Access Fault	At least one byte accessed is outside the authorizing capability bounds, or the bounds could not be decoded.
CHERI Store/AMO Access Fault	Authorizing capability failed any integrity check.

Prerequisites

C or Zca, RVY

Included in

C (RVY added instructions)

Operation (after expansion to 32-bit encodings)

See SY

A.1.11. C (RVY modified behavior)

An RVY core which supports C also supports C (RVY modified behavior) which *modifies* the behavior of some instructions.

c.ymv is renamed from `c.mv` *to avoid ambiguity in disassembly.*

Table 68. C (RVY modified behavior) instruction extension

Mnemonic	RV32Y	RV64Y	Function
C.ADDI16SP (RVY)	✔	✔	Stack pointer increment in blocks of 16, 16-bit encoding
C.ADDI4SPN (RVY)	✔	✔	Stack pointer increment in blocks of 4, 16-bit encoding
C.YMV	✔	✔	Capability register copy, 16-bit encoding
C.JAL (RV32Y)	✔		Immediate offset jump, and link and seal to capability register, 16-bit encoding
C.JALR (RVY)	✔	✔	Jump to capability register, and link and seal to capability register, 16-bit encoding
C.JR (RVY)	✔	✔	Jump to capability register, 16-bit encoding

A.1.11.1. C.ADDI16SP (RVY)

Synopsis

Stack pointer increment in blocks of 16, 16-bit encoding

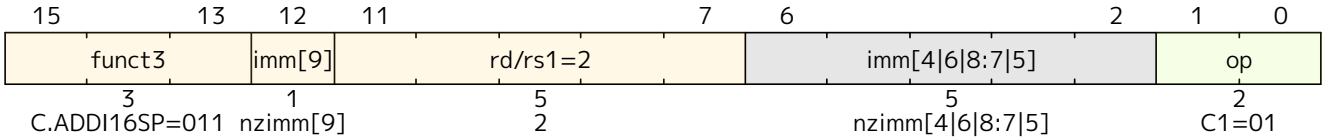
Mnemonic

```
c.addi16sp nzimm
```

Expansion

```
yaddi sp, sp, nzimm
```

Encoding



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).

Set `sp.tag=0` if `sp` is sealed.

Set `rd.tag=0` if the resulting capability cannot be represented exactly.

Set `rd.tag=0` if `sp` fails any integrity checks.

Prerequisites

C or Zca, RVY

Included in

C (RVY modified behavior)

Operation

```
execute(CADDI(sp, sp, sign_extend(nzimm)))
```

A.1.11.2. C.ADDI4SPN (RVY)

Synopsis

Stack pointer increment in blocks of 4, 16-bit encoding

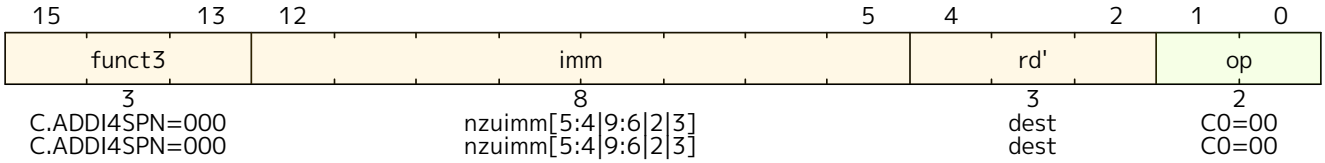
Mnemonic

```
c.addi4spn rd', nzuimm
```

Expansion

```
yaddi rd', sp, nzuimm
```

Encoding



Description

Copy `sp` to `rd'`. Add a zero-extended non-zero immediate, scaled by 4, to `rd'.address`.

Set `rd'.tag=0` if `sp` is sealed.

Set `rd'.tag=0` if the resulting capability cannot be represented exactly.

Set `rd'.tag=0` if `sp` 's bounds are malformed, or if any of the reserved fields are set.

Prerequisites

C or Zca, Zyhybrid

Included in

C (RVY modified behavior)

Operation

```
let cd = creg2reg_idx(cdc) in
execute(CADDI(cd, sp, zero_extend(nzuimm)))
```

A.1.11.3. C.YMV

Synopsis

Capability register copy, 16-bit encoding

Mnemonic

```
c.ymv rd, rs2
```

Expansion

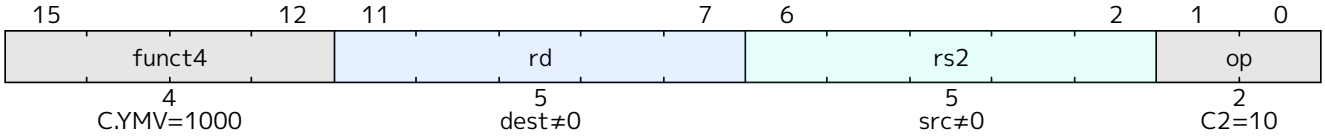
```
ymv rd, rs2
```

Suggested assembly syntax

```
ymv rd, rs2
```

`c.mv` *is remapped to copy YLEN-bit registers for RVY. The mnemonic is changed to avoid ambiguity about whether the copy is XLEN or YLEN-bits.*

Encoding



Description

Capability register `rd` is replaced with the contents of `rs2`.

This instruction can propagate valid capabilities which fail integrity checks.

Prerequisites

C or Zca, RVY

Included in

C (RVY modified behavior)

Operation (after expansion to 32-bit encoding)

See YMV

A.1.11.4. C.JR (RVY)

Synopsis

Jump to capability register, 16-bit encoding

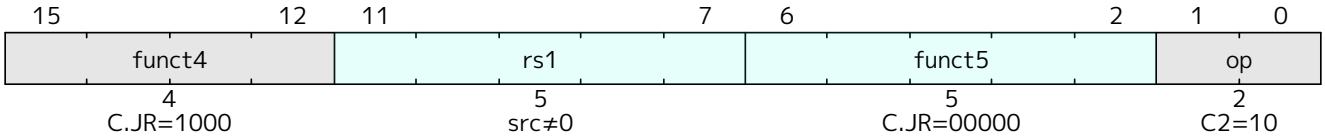
Mnemonic

```
c.jr rs1
```

Expansion

```
jalr x0, 0(rs1)
```

Encoding



***(CHERI) Capability Mode* Description**

See JALR (RVY) for execution of the expanded instruction as shown above. Note that the `offset` is zero in the expansion.

Prerequisites

C or Zca, RVY

Included in

C (RVY modified behavior)

Operation (after expansion to 32-bit encodings)

See JALR (RVY)

A.1.11.5. C.JAL (RV32Y)

Synopsis

Immediate offset jump, and link and seal to capability register, 16-bit encoding

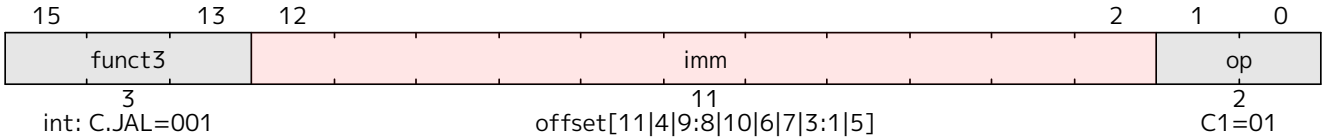
Mnemonic (RV32Y)

```
c.jal x1, offset
```

Expansion (RV32Y)

```
jal x1, offset
```

Encoding (RV32Y)



Description

Link the next linear pc to `rd` and seal. Jump to pc.address+offset.

Prerequisites

C or Zca, RVY

Included in

C (RVY modified behavior)

Operation (after expansion to 32-bit encodings)

See JAL (RVY)

A.1.11.6. C.JALR (RVY)

Synopsis

Jump to capability register, and link and seal to capability register, 16-bit encoding

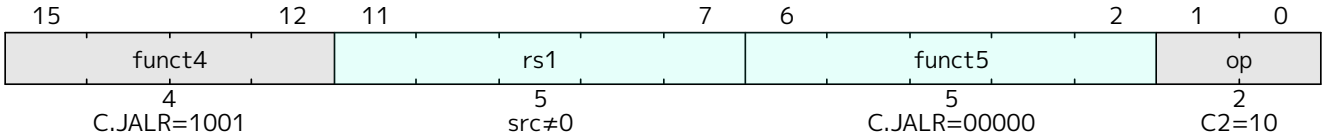
Mnemonic

```
c.jalr x1, rs1
```

Expansion

```
jalr x1, 0(rs1)
```

Encoding



Description

See JALR (RVY) for execution of the expanded instruction as shown above. Note that the `offset` is zero in the expansion.

Exceptions

See JALR (RVY)

Prerequisites

C or Zca, RVY

Included in

C (RVY modified behavior)

Operation (after expansion to 32-bit encodings)

See JALR (RVY)

A.1.12. Zalrsc (RVY added instructions)

Specifying RVY and Zalrsc adds atomic capability load and store instructions.

Table 69. Zalrsc (RVY added instructions) instruction extension

Mnemonic	RV32Y	RV64Y	Function
LR.Y	✔	✔	Load Reserved capability
SC.Y	✔	✔	Store Conditional capability

A.1.12.1. LR.Y

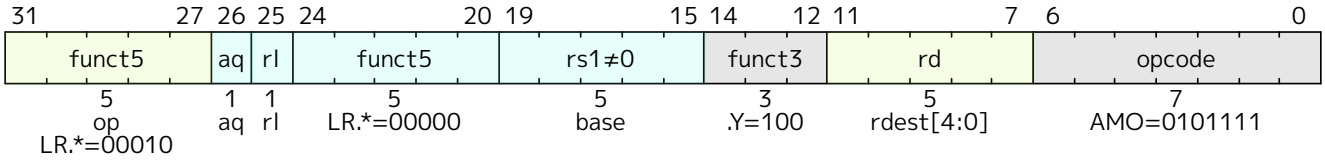
Synopsis

Load Reserved capability

Mnemonic

```
lr.y rd, 0(rs1)
```

Encoding



Any instance of this instruction with `rs1`=`x0` *will raise an exception, as* `x0` *is defined to always hold a NULL capability. As such, the encodings with* `rs1`=`x0` *are RESERVED for use by future extensions.*

Description

Calculate the effective address of the memory access by adding `rs1.address` to the sign-extended 12-bit offset.

Authorize the memory access with the capability in `rs1`.

Load a naturally aligned YLEN-bit data value from memory.

If the PMA is *CHERI Capability Tag* then load the associated capability tag, otherwise set the capability tag to zero.

Set the reservation as for LR.W/D.

Use the YLEN-bit data and the capability tag to determine the value of `rd` as specified by the LY instruction.

This instruction can propagate valid capabilities which fail integrity checks.

Exceptions

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

Exceptions occur when the authorizing capability fails one of the checks listed below:

Kind	Reason
CHERI Load Access Fault	Authorizing capability tag is set to 0.
CHERI Load Access Fault	Authorizing capability is sealed.
CHERI Load Access Fault	Authorizing capability does not grant the necessary permissions. Only R-permission is required.
CHERI Load Access Fault	At least one byte accessed is outside the authorizing capability bounds, or the bounds could not be decoded.
CHERI Load Access Fault	Authorizing capability failed any integrity check.

Prerequisites

RVY, and A or Zalrsc

Included in

Zalrsc (RVY added instructions)

Operation

```
TBD
```

A.1.12.2. SC.Y

Synopsis

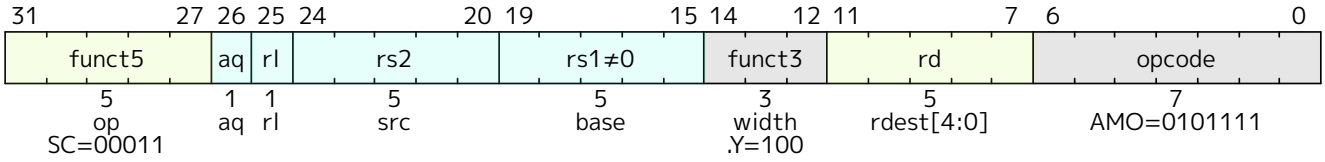
Store Conditional capability

Any instance of this instruction with `rs1`=`x0` will raise an exception, as `x0` is defined to always hold a NULL capability. As such, the encodings with `rs1`=`x0` are RESERVED for use by future extensions.

Mnemonic

```
sc.y rd, rs2, 0(rs1)
```

Encoding



Description

Calculate the effective address of the memory access by adding `rs1.address` to the sign-extended 12-bit offset.

Authorize the memory access with the capability in `rs1`.

Conditionally store, following the same rules as SC.W, a naturally aligned YLEN-bit data value in `rs2` to memory and the associated capability tag in `rs2`.

Set rd to 1 for success or 0 for failure.

The written capability tag may be cleared following the same modification rules as SY.

This instruction can propagate valid capabilities which fail integrity checks.

Exceptions

Store/AMO access fault exception when the effective address is not aligned to YLEN/8.

Store/AMO access fault if the stored capability tag is set to one and the PMA is *CHERI Capability Tag Fault*.

Exceptions occur when the authorizing capability fails one of the checks listed below:

Kind	Reason
CHERI Store/AMO Access Fault	Authorizing capability tag is set to 0.
CHERI Store/AMO Access Fault	Authorizing capability is sealed.
CHERI Store/AMO Access Fault	Authorizing capability does not grant the necessary permissions.
CHERI Store/AMO Access Fault	At least one byte accessed is outside the authorizing capability bounds, or the bounds could not be decoded.
CHERI Store/AMO Access Fault	Authorizing capability failed any integrity check.

Prerequisites

RVY, and A or Zalrsc

Included in

Zalrsc (RVY added instructions)

Operation

```
TBD
```

A.1.13. Zaamo (RVY added instructions)

Specifying RVY and Zaamo gives Zaamo (RVY added instructions) functionality, which adds atomic capability swap.

Table 70. Zaamo (RVY added instructions) instruction extension

Mnemonic	RV32Y	RV64Y	Function
AMOSWAP.Y	✔	✔	Atomic swap of capabilities

A.1.13.1. AMOSWAP.Y

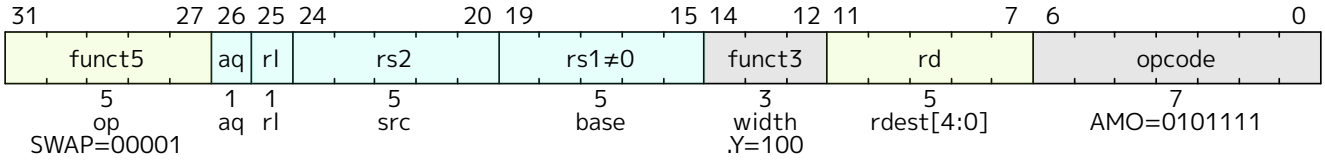
Synopsis

Atomic swap of capabilities

Mnemonic

```
amoswap.y rd, rs2, 0(rs1)
```

Encoding



Any instance of this instruction with `rs1`=`x0` *will raise an exception, as* `x0` *is defined to always hold a NULL capability. As such, the encodings with* `rs1`=`x0` *are RESERVED for use by future extensions.*

Description

Atomic swap of capability type, authorized by the capability in `rs1`.

The operation is equivalent to an atomically executed sequence of:

```
ly rd, 0(rs1)
```

```
sy rs2, 0(rs1)
```

With the proviso that `rd` is only updated if no exceptions are raised.

Permissions

Requires R-permission and W-permission in the authorizing capability.

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

Exceptions

If the address is not naturally aligned raise a *Store/AMO address misaligned* exception or a *Store/AMO access fault* exception. See "Zaamo" for details on which one is raised.

Exceptions occur when the authorizing capability fails one of the checks listed below:

Kind	Reason
CHERI Store/AMO Access Fault	Authorizing capability tag is set to 0.
CHERI Store/AMO Access Fault	Authorizing capability is sealed.
CHERI Store/AMO Access Fault	Authorizing capability does not grant the necessary permissions. W-permission and R-permission are both required.
CHERI Store/AMO Access Fault	At least one byte accessed is outside the authorizing capability bounds, or the bounds could not be decoded.

Prerequisites

RVY, and A or Zaamo

Included in

Zaamo (RVY added instructions)

Operation

TODO

A.1.14. Zba (RVY added instructions)

Specifying RVY and Zba gives Zba (RVY added instructions) functionality, which adds more instructions.

Table 71. Zba (RVY added instructions) instruction extension

Mnemonic	RV32Y	RV64Y	Function
YSH1ADD	✔	✔	shift and add, representability check
YSH2ADD	✔	✔	shift and add, representability check
YSH3ADD	✔	✔	shift and add, representability check
YSH4ADD (RV64Y)		✔	shift and add, representability check
YSH1ADD.UW (RV64Y)		✔	shift and add unsigned word, representability check
YSH2ADD.UW (RV64Y)		✔	shift and add unsigned word, representability check
YSH3ADD.UW (RV64Y)		✔	shift and add unsigned word, representability check
YSH4ADD.UW (RV64Y)		✔	shift and add unsigned word, representability check

There is no RVY equivalent for `add.uw` *as only having the integer version is sufficient.*

A.1.14.1. YSH1ADD

See YSH4ADD (RV64Y).

A.1.14.2. YSH2ADD

See YSH4ADD (RV64Y).

A.1.14.3. YSH3ADD

See YSH4ADD (RV64Y).

A.1.14.4. YSH4ADD (RV64Y)

Synopsis

Shift by *n* and add for address generation (YSH1ADD, YSH2ADD, YSH3ADD, YSH4ADD)

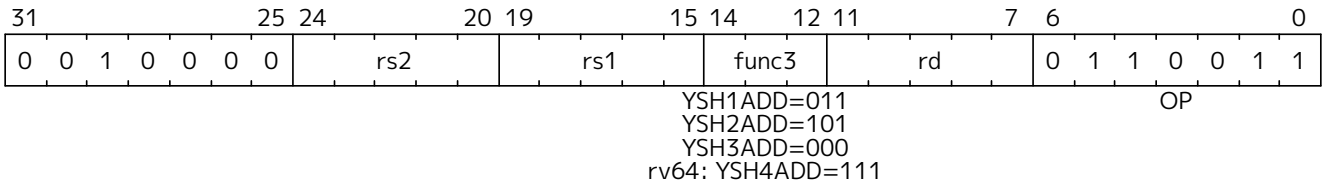
Mnemonics (RVY)

```
ysh1add rd, rs1, rs2
ysh2add rd, rs1, rs2
ysh3add rd, rs1, rs2
```

Mnemonics (RV64Y)

```
ysh4add rd, rs1, rs2
```

Encoding



Description

Copy the capability in `rs2` to `rd`.

Increment `rd.address` by `rs1` shifted left by *n* bit positions.

Set `rd.tag=0` if `rs2` is sealed.

Set `rd.tag=0` if the resulting capability cannot be represented exactly.

Set `rd.tag=0` if `rs2` fails any integrity checks.

Included in

Zba (RVY added instructions)

Operation

```
let rs1_val = X(rs1);
let cs2_val = C(cs2);
let shamt : range(0,3) = match op {
  RISCV_SH1ADD => 1,
```

```
  RISCV_SH2ADD => 2,
  RISCV_SH3ADD => 3,
};
let result = incCapAddrChecked(cs2_val, rs1_val << shamt);
C(cd) = result;
RETIRE_SUCCESS
```

A.1.14.5. YSH1ADD.UW (RV64Y)

See YSH4ADD.UW (RV64Y).

A.1.14.6. YSH2ADD.UW (RV64Y)

See YSH4ADD.UW (RV64Y).

A.1.14.7. YSH3ADD.UW (RV64Y)

See YSH4ADD.UW (RV64Y).

A.1.14.8. YSH4ADD.UW (RV64Y)

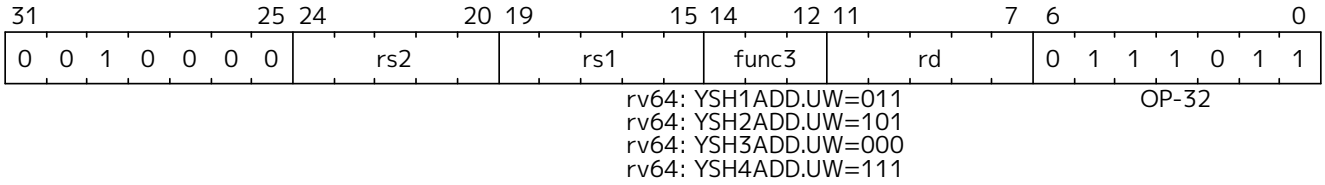
Synopsis

Shift by *n* and add unsigned words for address generation (YSH1ADD.UW, YSH2ADD.UW, YSH3ADD.UW, YSH4ADD.UW)

Mnemonics (RV64Y)

```
ysh1add.uw rd, rs1, rs2
ysh2add.uw rd, rs1, rs2
ysh3add.uw rd, rs1, rs2
ysh4add.uw rd, rs1, rs2
```

Encoding



Description

Copy the capability in `rs2` to `rd`.

Increment `rd.address` by the unsigned word `rs1` shifted left by *n* bit positions.

Set `rd.tag=0` if `rs2` is sealed.

Set `rd.tag=0` if the resulting capability cannot be represented exactly.

Set `rd.tag=0` if `rs2` fails any integrity checks.

Included in

Zba (RVY added instructions)

Operation

```
let rs1_val = X(rs1);
let cs2_val = C(cs2);
let shamt : range(0,3) = match op {
  RISCV_ADDUW    => 0,
  RISCV_SH1ADDUW => 1,
```

```
  RISCV_SH2ADDUW => 2,
  RISCV_SH3ADDUW => 3,
};
let result = incCapAddrChecked(cs2_val, zero_extend(rs1_val[31..0]) <<
shamt);
C(cd) = result;
RETIRE_SUCCESS
```

A.1.15. Zicbom (RVY modified behavior)

Specifying RVY and Zicbom gives Zicbom (RVY modified behavior) functionality, which extends the checking.

Table 72. Zicbom (RVY modified behavior) instruction extension

Mnemonic	RV32Y	RV64Y	Function
CBO.INVAL (RVY)	✔	✔	Cache block invalidate (implemented as clean)
CBO.CLEAN (RVY)	✔	✔	Cache block clean
CBO.FLUSH (RVY)	✔	✔	Cache block flush

A.1.15.1. CBO.CLEAN (RVY)

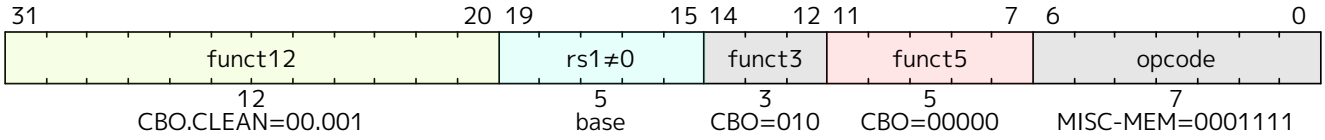
Synopsis

Perform a clean operation on a cache block

Mnemonic

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

Encoding



Description

A CBO.CLEAN instruction performs a clean operation on the cache block whose effective address is the base address specified in `rs1`. The authorizing capability for this operation is `rs1`.

Exceptions

Kind	Reason
CHERI Store/AMO Access Fault	Authorizing capability tag is set to 0.
CHERI Store/AMO Access Fault	Authorizing capability is sealed.
CHERI Store/AMO Access Fault	Authorizing capability does not grant the necessary permissions. W-permission and R-permission are both required.
CHERI Store/AMO Access Fault	None of the bytes accessed are within the bounds, or the bounds could not be decoded.
CHERI Store/AMO Access Fault	Authorizing capability failed any integrity check.

Prerequisites

Zicbom, RVY

Included in

Zicbom (RVY modified behavior)

Operation

```
TBD
```

A.1.15.2. CBO.FLUSH (RVY)

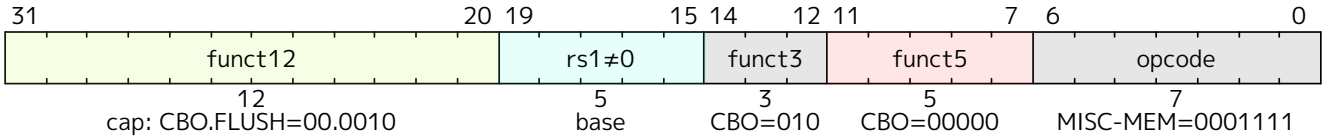
Synopsis

Perform a flush operation on a cache block

Mnemonic

```
cbo.flush 0(rs1)
```

Encoding



Description

A CBO.FLUSH instruction performs a flush operation on the cache block whose effective address is the base address specified in `rs1`. The authorizing capability for this operation is `rs1`.

Exceptions

Kind	Reason
CHERI Store/AMO Access Fault	Authorizing capability tag is set to 0.
CHERI Store/AMO Access Fault	Authorizing capability is sealed.
CHERI Store/AMO Access Fault	Authorizing capability does not grant the necessary permissions. W-permission and R-permission are both required.
CHERI Store/AMO Access Fault	None of the bytes accessed are within the bounds, or the bounds could not be decoded.
CHERI Store/AMO Access Fault	Authorizing capability failed any integrity check.

Prerequisites

Zicbom, RVY

Included in

Zicbom (RVY modified behavior)

Operation

```
TBD
```

A.1.15.3. CBO.INVAL (RVY)

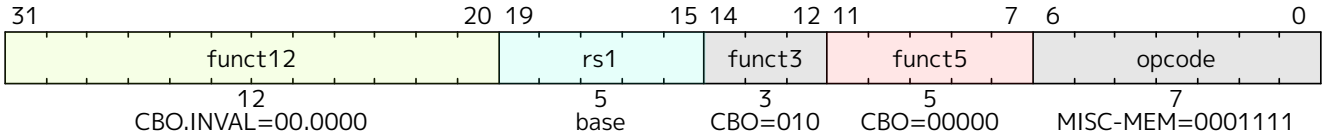
Synopsis

Perform an invalidate operation on a cache block

Mnemonic

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

Encoding



Description

A CBO.INVAL instruction performs an invalidate operation on the cache block whose effective address is the base address specified in `rs1`. The authorizing capability for this instruction is `rs1`.

Exceptions

Kind	Reason
Illegal instruction	pc does not grant ASR-permission.
CHERI Store/AMO Access Fault	Authorizing capability tag is set to 0.
CHERI Store/AMO Access Fault	Authorizing capability is sealed.
CHERI Store/AMO Access Fault	Authorizing capability does not grant the necessary permissions. W-permission, R-permission are both required.
CHERI Store/AMO Access Fault	At least one byte accessed is outside the authorizing capability bounds, or the bounds could not be decoded.
CHERI Store/AMO Access Fault	Authorizing capability failed any integrity check.

CSR state controls whether CBO.INVAL performs cache block flushes instead of invalidations for less privileged modes.

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 as a minimum, and a sensible implementation choice for CHERI systems is to always execute CBO.INVAL as CBO.FLUSH.

Prerequisites

Zicbom, RVY

Included in

Zicbom (RVY modified behavior)

Operation

```
TBD
```

A.1.16. Zicboz (RVY modified behavior)

Specifying RVY and Zicboz gives Zicboz (RVY modified behavior) functionality, which extends the checking.

Table 73. Zicboz (RVY modified behavior) instruction extension

Mnemonic	RV32Y	RV64Y	Function
CBO.ZERO (RVY)	✔	✔	Cache block zero

A.1.16.1. CBO.ZERO (RVY)

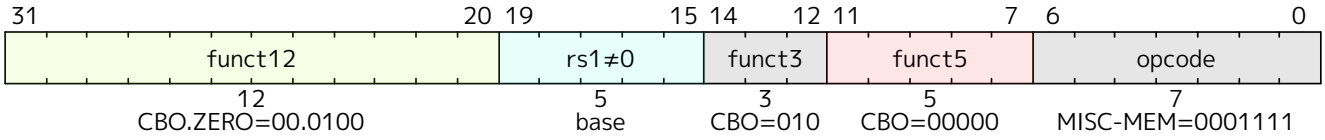
Synopsis

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

Mnemonic

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

Encoding



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 `rs1`. An implementation may or may not update the entire set of bytes atomically although each individual write must atomically clear the capability tag bit of the corresponding aligned YLEN-bit location. The authorizing capability for this instruction is `rs1`.

Exceptions

Store/AMO access fault exception when the effective address is not aligned to YLEN/8.

Store/AMO access fault if the stored capability tag is set to one and the PMA is *CHERI Capability Tag Fault*.

Exceptions occur when the authorizing capability fails one of the checks listed below:

Kind	Reason
CHERI Store/AMO Access Fault	Authorizing capability tag is set to 0.
CHERI Store/AMO Access Fault	Authorizing capability is sealed.
CHERI Store/AMO Access Fault	Authorizing capability does not grant the necessary permissions.
CHERI Store/AMO Access Fault	At least one byte accessed is outside the authorizing capability bounds, or the bounds could not be decoded.
CHERI Store/AMO Access Fault	Authorizing capability failed any integrity check.

Prerequisites

Zicboz, RVY

Included in

Zicboz (RVY modified behavior)

Operation

```
TBD
```

A.1.17. Zicbop (RVY modified behavior)

Specifying RVY and Zicbop gives Zicbop (RVY modified behavior) functionality, which extends the checking.

Table 74. Zicbop (RVY modified behavior) instruction extension

Mnemonic	RV32Y	RV64Y	Function
PREFETCH.I (RVY)	✔	✔	Cache Block Prefetch for Instruction Fetch
PREFETCH.R (RVY)	✔	✔	Cache Block Prefetch for Data Read
PREFETCH.W (RVY)	✔	✔	Cache block Prefetch for Data Write

A.1.17.1. PREFETCH.I (RVY)

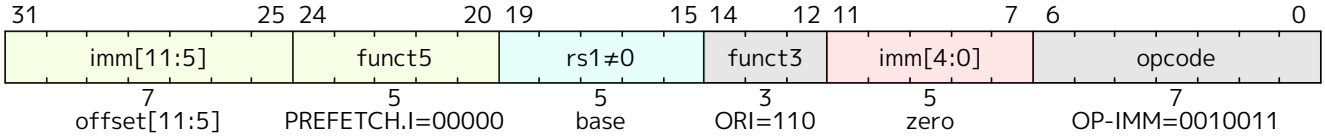
Synopsis

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

Mnemonic

```
prefetch.i offset(rs1)
```

Encoding



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 authorizing capability for this operation is `rs1`. This instruction does not throw any exceptions. However, following the rules from Chapter 2, this instruction does not perform a prefetch if it is not authorized by `rs1`.

PREFETCH.I does not perform a memory access if one or more of the following conditions of the authorizing capability are met:

- The capability tag is not set
- The sealed bit is set
- No bytes of the cache line requested is in bounds
- X-permission is not set
- Any integrity check fails

If the checks above pass, an implementation may opt to cache a copy of the cache block in a cache accessed by an instruction fetch in order to improve memory access latency, but this behavior is not required.

Prerequisites

Zicbop, RVY

Included in

Zicbop (RVY modified behavior)

Operation

```
TODO
```

A.1.17.2. PREFETCH.R (RVY)

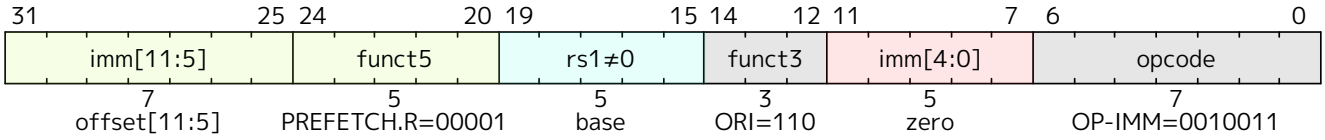
Synopsis

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

Mnemonic

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

Encoding



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 authorizing capability for this operation is `rs1`. This instruction does not throw any exceptions. However, following the rules from Chapter 2, this instruction does not perform a prefetch if it is not authorized by `rs1`.

PREFETCH.R does not perform a memory access if one or more of the following conditions of the authorizing capability are met:

- The capability tag is not set
- The sealed bit is set
- No bytes of the cache line requested is in bounds
- R-permission is not set
- Any integrity check fails

If the checks above pass, an implementation may opt to cache a copy of the cache block in a cache accessed by a data read in order to improve memory access latency, but this behavior is not required.

Prerequisites

Zicbop, RVY

Included in

Zicbop (RVY modified behavior)

Operation

```
TODO
```

A.1.17.3. PREFETCH.W (RVY)

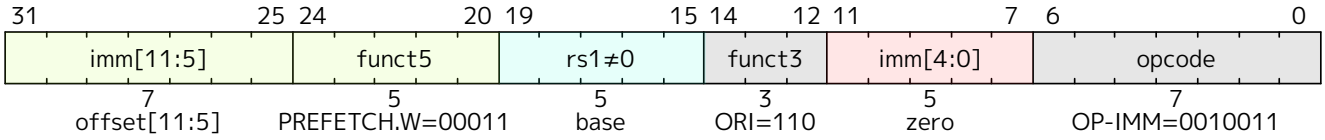
Synopsis

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

Mnemonic

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

Encoding



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 authorizing capability for this operation is `rs1`. This instruction does not throw any exceptions. However, following the rules from Chapter 2, this instruction does not perform a prefetch if it is not authorized by `rs1`.

PREFETCH.W does not perform a memory access if one or more of the following conditions of the authorizing capability are met:

- The capability tag is not set
- The sealed bit is set
- No bytes of the cache line requested is in bounds
- W-permission is not set
- Any integrity check fails

If the checks above pass, an implementation may opt to cache a copy of the cache block in a cache accessed by a data write in order to improve memory access latency, but this behavior is not required.

Prerequisites

Zicbop, RVY

Included in

Zicbop (RVY modified behavior)

Operation

```
TODO
```

A.1.18. Zyhybrid

An RVY core which supports Zyhybrid adds the instructions in Table 75.

Table 75. Zyhybrid instruction extension

Mnemonic	RV32Y	RV64Y	Function
YMODEW	✔	✔	Set capability execution mode
YMODER	✔	✔	Read capability mode
YMODESWY	✔	✔	Switch execution to *(CHERI) Capability Mode*
YMODESWI	✔	✔	Switch execution to *(Non-CHERI) Address Mode*

A.1.19. "Zcmp", "Zcmt" (RVY)

This chapter is not part of the v1.0 ratification package.

A.1.20. "Zcmp" Standard Extension For Code-Size Reduction

The push (CM.PUSH (RV32Y)) and pop (CM.POP (RV32Y), CM.POPRET (RV32Y), CM.POPRETZ (RV32Y)) instructions are redefined in *(CHERI) Capability Mode* to save/restore capability data.

The double move instructions (CM.MVSA01 (RV32Y), CM.MVA01S (RV32Y)) are redefined in *(CHERI) Capability Mode* to move capability data between registers. The saved register mapping is as shown in Table 76.

Table 76. saved register mapping for Zcmp

saved register specifier	xreg	integer ABI	RV32Y ABI
0	x8	s0	s0
1	x9	s1	s1
2	x18	s2	s2
3	x19	s3	s3
4	x20	s4	s4
5	x21	s5	s5
6	x22	s6	s6
7	x23	s7	s7

A.1.20.1. CM.PUSH (RV32Y)

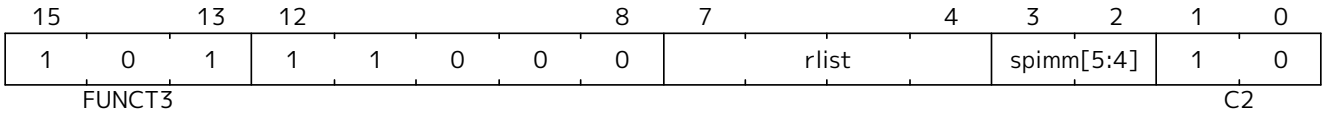
Synopsis

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 encoding.

Mnemonic

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

Encoding



Assembly Syntax:

```
cm.push {reg_list},  -stack_adj
cm.push {xreg_list}, -stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32Y:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32Y:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
```

```
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

rlist values 0 to 3 are reserved for a future EABI variant

Description

Create stack frame, store capability registers as specified in *creg_list* using SY semantics.

Optionally allocate additional multiples of 16-byte stack space in `sp`.

All accesses are authorized against `sp`.

Prerequisites

C or Zca, RVY, Zcmp

Operation

```
TBD
```

A.1.20.2. CM.POP (RV32Y)

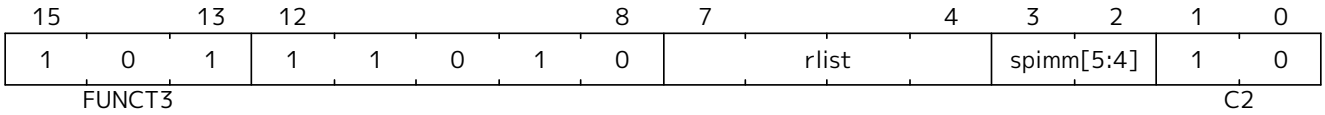
Synopsis

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.

Mnemonic

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

Encoding



Assembly Syntax:

```
cm.pop {reg_list},  stack_adj
cm.pop {xreg_list}, stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32Y:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32Y:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
```

```
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

rlist values 0 to 3 are reserved for a future EABI variant

Description

Load capability registers as specified in *creg_list* using LY semantics.

Deallocate stack frame.

All accesses are authorized by `sp`.

Prerequisites

C or Zca, RVY, Zcmp

Operation

```
TBD
```

A.1.20.3. CM.POPRET (RV32Y)

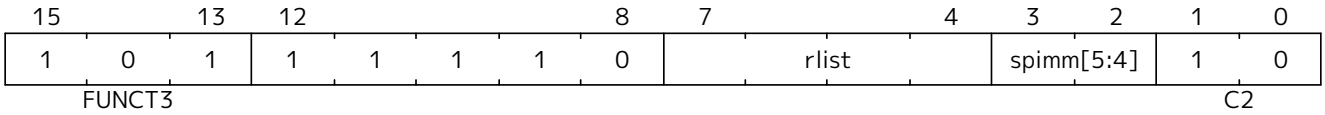
Synopsis

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.

Mnemonic

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

Encoding



Assembly Syntax:

```
cm.popret {reg_list},  stack_adj
cm.popret {xreg_list}, stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32Y:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32Y:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
```

```
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

rlist values 0 to 3 are reserved for a future EABI variant

Description

Load capability registers as specified in *creg_list* using LY semantics.

Deallocate stack frame.

Return by calling JALR (RVY) to `ra`.

All data accesses are authorized by `sp`.

The return destination is authorized by `ra`.

Prerequisites

C or Zca, RVY, Zcmp

Operation

```
TBD
```

A.1.20.4. CM.POPRETZ (RV32Y)

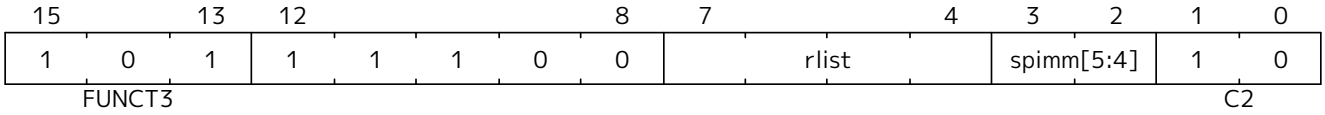
Synopsis

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 encoding.

Mnemonic

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

Encoding



Assembly Syntax:

```
cm.popretz {reg_list},  stack_adj
cm.popretz {xreg_list}, stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32Y:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32Y:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
```

```
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

rlist values 0 to 3 are reserved for a future EABI variant

Description

Load capability registers as specified in *creg_list* using LY semantics.

Deallocate stack frame.

Move zero into `a0`.

Return by calling JALR (RVY) to `ra`.

All data accesses are authorized by `sp`.

The return destination is authorized by `ra`.

Prerequisites

C or Zca, RVY, Zcmp

Operation

```
TBD
```

A.1.20.5. CM.MVSA01 (RV32Y)

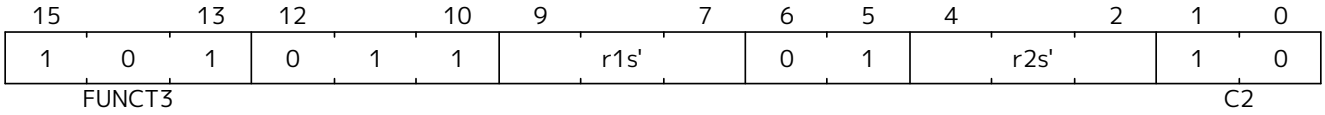
Synopsis

CM.MVSA01: Move argument registers 0 and 1 into two saved registers. 16-bit encoding.

Mnemonic

```
cm.mvsa01 c1s', c2s'
```

Encoding



The encoding uses sreg number specifiers instead of xreg number specifiers to save encoding space. The saved register encoding is shown in Table 76.

Description

Atomically move two saved capability registers `s0-s7` into `a0` and `a1`.

Prerequisites

C or Zca, RVY, Zcmp

Operation

```
TBD
```

A.1.20.6. CM.MVA01S (RV32Y)

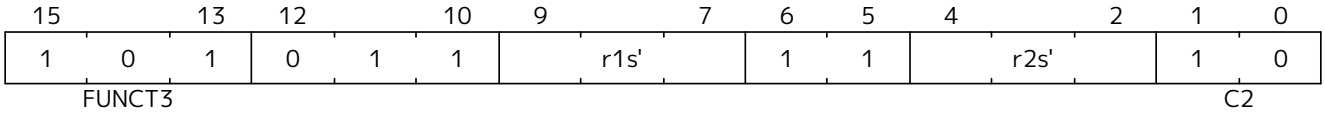
Synopsis

Move two saved registers into argument registers 0 and 1. 16-bit encoding.

Mnemonic

```
cm.mva01s c1s', c2s'
```

Encoding



The encoding uses sreg number specifiers instead of xreg number specifiers to save encoding space. The saved register encoding is shown in Table 76.

Description

Atomically move two capability registers `a0` and `a1` into `s0-s7`.

Prerequisites

C or Zca, RVY, Zcmp

Operation

```
TBD
```

A.1.21. "Zcmt" Standard Extension For Code-Size Reduction

The table jump instructions (CM.JT (RV32Y), CM.JALT (RV32Y)) are *not* redefined in *(CHERI) Capability Mode* to have capabilities in the jump table. This is to prevent the code-size growth caused by doubling the size of the jump table.

In the future, new jump table modes or new encodings can be added to have capabilities in the jump table.

The jump vector table CSR jvt (RVY) is a full capability so that it can only be configured to point to accessible memory. All accesses to the jump table are checked against jvt (RVY) in *(CHERI) Capability Mode*, and against pc bounds in *(Non-CHERI) Address Mode*. This allows the jump table to be accessed when the pc bounds are set narrowly to the local function only in *(CHERI) Capability Mode*.

In (CHERI) Capability Mode the instruction fetch bounds check is authorized by two different capabilities - jvt (RVY) for the table access and pc for the CM.JALT (RV32Y)/CM.JT (RV32Y) instruction, and target instruction.

In (CHERI) Capability Mode the implementation doesn't need to expand and bounds check against jvt (RVY) on every access, it is sufficient to decode the valid accessible range of entries after every write to jvt (RVY), and then check that the accessed entry is in that range.

A.1.21.1. Jump Vector Table CSR (jvt)

The Zcmt jvt CSR is extended to be a full capability.

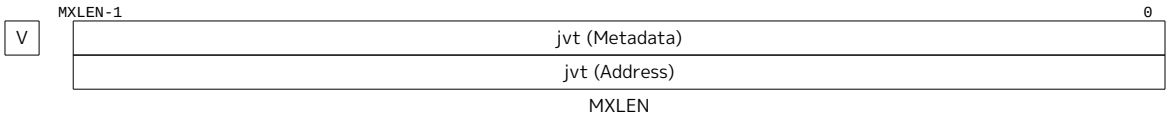


Figure 20. Jump Vector Table Capability register

All instruction fetches from the jump vector table are checked against jvt (RVY) in *(CHERI) Capability Mode*. In *(Non-CHERI) Address Mode* the address field gives the base address of the table, and the access is checked against pc bounds.

See CM.JALT (RV32Y), CM.JT (RV32Y).

If the access to the jump table succeeds, then the instructions execute as follows:

- CM.JT (RV32Y) executes as J or AUIPC+JR
- CM.JALT (RV32Y) executes as JAL or AUIPC+JALR

As a result the capability metadata is retained in pc during execution.

A.1.21.2. CM.JALT (RV32Y)

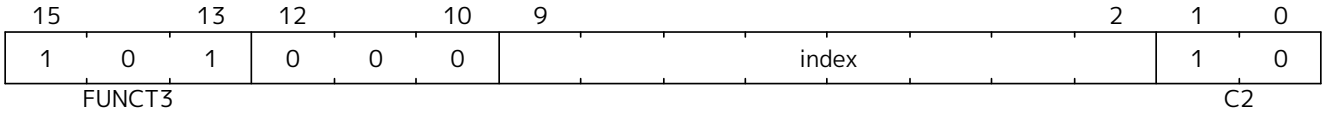
Synopsis

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

Mnemonic (RV32)

`cm.jalt` *`index`*

Encoding



For this encoding to decode as CM.JALT (RV32Y), index≥32, otherwise it decodes as CM.JT (RV32Y).

Description (RV32Y)

Redirect instruction fetch via the jump table defined by the indexing via `jvt.address+ index*XLEN/8`, checking every byte of the jump table access against jvt (RVY) bounds (not against pc) and requiring X-permission. Link to `cra`.

The target pc is calculated by replacing the current pc address with the value read from the jump table, and is updated using the semantics of the YADDRW instruction.

If the jvt (RVY) check fails, then clear the capability tag of the target pc.

If Zcherihybrid is implemented and the CHERI execution mode is (Non-CHERI) Address Mode then the table access is checked against pc bounds.

Permissions (RV32Y)

Requires jvt (RVY) to have its capability tag set, not be sealed, have X-permission and for the full XLEN-wide table access to be in jvt (RVY) bounds.

Prerequisites for (RV32Y)

C or Zca, RVY, Zcmt

Operation

```
TBD
```

A.1.21.3. CM.JT (RV32Y)

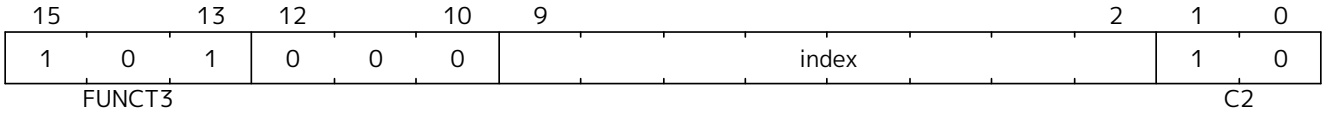
Synopsis

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

Mnemonic (RV32)

`cm.jt` *`index`*

Encoding



For this encoding to decode as CM.JT (RV32Y), index<32, otherwise it decodes as CM.JALT (RV32Y).

Description (RV32Y)

Redirect instruction fetch via the jump table defined by the indexing via `jvt.address+ index*XLEN/8`, checking every byte of the jump table access against jvt (RVY) bounds (not against pc) and requiring X-permission.

The target pc is calculated by replacing the current pc address with the value read from the jump table, and is updated using the semantics of the YADDRW instruction.

If the jvt (RVY) check fails, then clear the capability tag of the target pc.

If Zcherihybrid is implemented and the CHERI execution mode is (Non-CHERI) Address Mode then the table access is checked against pc bounds.

Permissions (RV32Y)

Requires jvt (RVY) to have its capability tag set, not be sealed, have X-permission and for the full XLEN-wide table access to be in jvt (RVY) bounds.

Prerequisites for (RV32Y)

C or Zca, RVY, Zcmt

Operation

```
TBD
```

A.2. ISA changes since 0.9.5

Many mnemonics have been renamed since v0.9.5 of the specification as shown in Table 77.

Table 77. ISA renames since 0.9.5

Mnemonic	Old mnemonic	Extension
LY	LC	RVY
SY	SC	RVY
C.LYSP	C.LCSP	C (RVY added instructions)

Mnemonic	Old mnemonic	Extension
C.SYSP	C.SCSP	C (RVY added instructions)
C.LY	C.LC	C (RVY added instructions)
C.SY	C.SC	C (RVY added instructions)
AUIPC (RVY)	AUIPCC	RVI (RVY modified behavior)
YADD	CADD	RVY
YADDI	CADDI	RVY
YADDRW	SCADDR	RVY
YTAGR	GCTAG	RVY
YPERMR	GCPERM	RVY
YMV	CMV	RVY
YHIR	GCHI	RVY
YHIW	SCHI	RVY
SYEQ	SCEQ	RVY
YSENTRY	SENTRY	Zysentry
YLT	SCSS	RVY
YBLD	CBLD	Zybld
YBNDSW	SCBNDS	RVY
YBNDSWI	SCBNDSI	RVY
YBNDSRW	SCBNDSR	RVY
YBNDSRDW	SCBNDSRD	Zybndsrdw
YAMASK	CRAM	RVY
YBASER	GCBASE	RVY
YLENR	GCLEN	RVY
YTYPER	GCTYPE	RVY
YTOPR	GCTOP	Zytopr
YMODEW	SCMODE	Zyhybrid
YMODER	GCMODE	Zyhybrid
YMODESWY	MODESW.CAP	Zyhybrid
YMODESWI	MODESW.INT	Zyhybrid
C.ADDI16SP (RVY)	C.CADDI16SP	C (RVY modified behavior)
C.ADDI4SPN (RVY)	C.CADDI4SPN	C (RVY modified behavior)
C.YMV	C.CMV	C (RVY modified behavior)
C.JAL (RV32Y)	C.CJAL	C (RVY modified behavior)
JAL (RVY)	CJAL	RVI (RVY modified behavior)
JALR (RVY)	CJALR	RVI (RVY modified behavior)
C.JALR (RVY)	C.CJALR	C (RVY modified behavior)
C.JR (RVY)	C.CJR	C (RVY modified behavior)
LR.Y	LR.C	Zalrsc (RVY added instructions)
SC.Y	SC.C	Zalrsc (RVY added instructions)

Mnemonic	Old mnemonic	Extension
AMOSWAP.Y	AMOSWAP.C	Zaamo (RVY added instructions)
HLV.Y	HLV.C	H Extension (RVY added instructions)
HSV.Y	HSV.C	H Extension (RVY added instructions)

Some instructions have been added as shown in Table 78.

Table 78. Instructions added since 0.9.5

Mnemonic	Old mnemonic	Extension
YPERMC	N/A	RVY
SRLIY	N/A	RVY
PACKY	N/A	RVY
YSUNSEAL	N/A	RVY
YSH1ADD	N/A	Zba (RVY added instructions)
YSH2ADD	N/A	Zba (RVY added instructions)
YSH3ADD	N/A	Zba (RVY added instructions)
YSH4ADD (RV64Y)	N/A	Zba (RVY added instructions)
YSH1ADD.UW (RV64Y)	N/A	Zba (RVY added instructions)
YSH2ADD.UW (RV64Y)	N/A	Zba (RVY added instructions)
YSH3ADD.UW (RV64Y)	N/A	Zba (RVY added instructions)
YSH4ADD.UW (RV64Y)	N/A	Zba (RVY added instructions)
YSEAL	N/A	Zyseal
YUNSEAL	N/A	Zyseal

YSEAL and YUNSEAL are not included in the v1.0 ratification package.

1. *PACKY and SRLIY are actual instructions. YHIW and YHIR are pseudoinstructions.*
2. *ACPERM was replaced by YPERMC. The difference is that the mask is used to clear, not retain, permission bits.*
    a. *Clearing bits makes it much simpler to form the necessary constant compared to retaining bits, and so gives better code-size.*
3. *0.9.5 had SH[123]ADD, and the .UW forms, replaced by capability versions.*
    a. *This is no longer the case, so now the capability versions have new encodings.*
4. *There is no longer an SH4ADD instruction (i.e., the integer version).*
5. *The YSENTRY instruction is now in a separate extension Zysentry.*

The following changes are for forwards compatibility with Zycheriot:

1. *Capability encodings and extensions are now the naming authorities for capability types (only 0/unsealed exists in the base architecture).*
2. *Capability encoding formats are now in separate base parameterizations.*
3. *JALR (RVY) has been given explicit hooks for sentry handling (especially for future forward/backward arc distinction).*

A.3. Placeholder references to the unprivileged spec

This chapter only exists for the standalone document to allow references to resolve.

RV32I

See Chapter *RV32I Base Integer Instruction Set* in (RISC-V, 2023).

RV32E and RV64E

See Chapter *RV32E and RV64E Base Integer Instruction Sets* in

General purpose registers

See Chapter *RV32I Base Integer Instruction Set* in (RISC-V, 2023).

Load and Store Instructions

See Chapter *RV32I Base Integer Instruction Set* in (RISC-V, 2023).

Integer Register-Immediate Instructions

See Chapter *RV32I Base Integer Instruction Set* in (RISC-V, 2023).

Control Transfer Instructions

See Chapter *RV32I Base Integer Instruction Set* in (RISC-V, 2023).

Atomics

See Chapter *"A" Extension for Atomic Instructions* in (RISC-V, 2023).

Zba

See Chapter *"B" Extension for Bit Manipulation* in (RISC-V, 2023).

Zicbom

See Chapter *"CMO" Extensions for Base Cache Management Operation ISA* in (RISC-V, 2023).

Zcmt

See Chapter *"Zc*" Extension for Code Size Reduction* in (RISC-V, 2023).

Zcmp

See Chapter *"Zc*" Extension for Code Size Reduction* in (RISC-V, 2023).

jvt

See Chapter *"Zc*" Extension for Code Size Reduction* in (RISC-V, 2023).

Zaamo

See Chapter *"A" Extension for Atomic Instructions* in (RISC-V, 2023).

"Zalrsc" for RVY

See Chapter *"A" Extension for Atomic Instructions* in (RISC-V, 2023).

"Zaamo" for RVY

See Chapter *"A" Extension for Atomic Instructions* in (RISC-V, 2023).

Chapters for the privileged specification

Chapter 13. "Machine/Supervisor-Level ISA (RVY)" Extensions, Version 1.0

This chapter will appear in the priv spec. Exact location TBD.

This chapter describes integration of RVY with the RISC-V privileged architecture.

13.1. Machine-Level CSRs added or extended by RVY

RVY extends some M-mode CSRs to hold capabilities or otherwise add new functions. ASR-permission in the pc is always required for access to privileged CSRs.

13.1.1. Machine Trap Vector Base Address Capability Register (mtvec)

The mtvec register is extended to hold a code capability. Its reset value is nominally a Root Executable capability.

mtvec (RVY) exists in all CHERI implementations, and so may be used as a source of a Root Executable capability after reset.

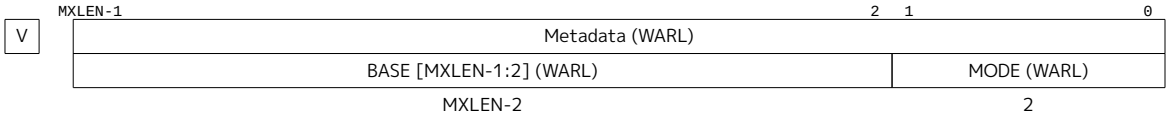


Figure 21. Machine-mode trap-vector base-capability register

The fields in the metadata are WARL as many fields can be implemented as constants.

Examples of WARL behavior include always setting X-permission to 1 and setting the reserved fields to zero, otherwise the capability is unusable. Another example is to partially or fully restrict the bounds to constant values.

Care must be taken however that suitable root capabilities are available to software after reset if this CSR does not represent one.

When traps are taken into machine mode, the pc is updated following the standard `mtvec` behavior. The capability tag and metadata from mtvec (RVY) are also written to the pc.

Following the standard `mtvec` behavior, the value of `mtvec.address` can be viewed with a range of different addresses:

1. The MODE field is included in `mtvec.address[1:0]` but it does not form part of the trap vector address.
2. When MODE=Vectored, the trap vector address is incremented by four times the interrupt number.
3. CSR reads include MODE in `mtvec.address[1:0]`.

`HICAUSE` is defined to be the largest interrupt cause value that the implementation can write to `xcause` when an interrupt is taken.

Therefore the minimum observable address is `mtvec.address & ~3` and the maximum is `(mtvec.address & ~3) + 4 x HICAUSE`.

All possible observable values must be in the Representable Range. Software must ensure this is true when

writing to mtvec (RVY), and the hardware sets the capability tag to zero if any values are out of the Representable Range.

Modifying the address of any capability outside of the Representable Range without clearing the capability tag causes a security hole as the interpretation of the bounds changes. Therefore requiring that all possible observable addresses are representable but not necessary in bounds is the minimum security requirement.

mtvec (RVY) is always updated using the semantics of the YADDRW instruction and so writing a sealed capability will cause the capability tag to be set to zero.

The capability in mtvec (RVY) is not unsealed when it is written to pc, unlike other executing from other CSRs such as mepc (RVY).

mtvec (RVY) follows the rule from `mtvec` about not needing to be able to hold all possible invalid addresses (see Invalid address conversion).

13.1.2. Machine Scratch Capability Register (mscratch)

The mscratch register is extended to hold a capability.

The reset value of the capability tag of this CSR is zero, the reset values of the metadata and address fields are UNSPECIFIED.

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

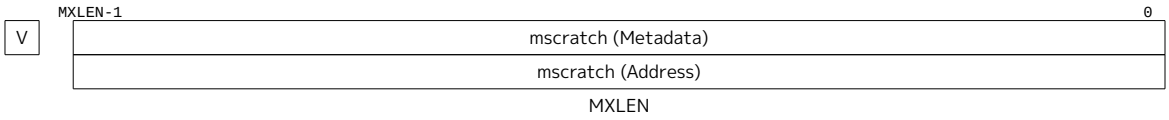


Figure 22. Machine-mode scratch capability register

13.1.3. Machine Exception Program Counter Capability (mepc)

The mepc is extended to hold a capability. Its reset value is nominally a Root Executable capability.

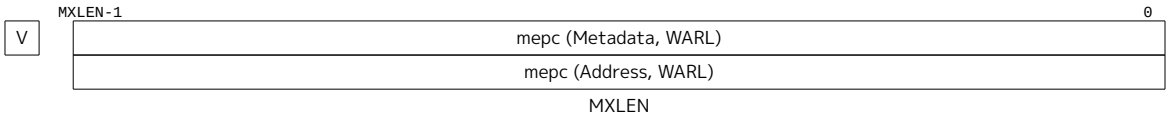


Figure 23. Machine exception program counter capability register

`mepc.address` is the `mepc` CSR, and so the follows the standard rules meaning that:

1. `mepc.address[0]=0`, and
2. `mepc.address[1]=0` when IALIGN is fixed to 32
3. `mepc.address[1]` reads as zero when IALIGN is programmable and is set to 32

As listed above for mtvec (RVY), this means that `mepc.address` can represent multiple different values. Therefore software must ensure that all possible values are in the Representable Range on writing, otherwise the hardware sets the written capability tag to zero.

Sealed capabilities may be written to mepc (RVY). The capability tag is set to zero on writing if:

1. `mepc.address[0]=1`, or

2. `mepc.address[1]=1` when IALIGN=32

In the following case the value of the capability tag observable in the CSR depends on the value of IALIGN:

1. mepc (RVY) is sealed, the capability tag is set, and
2. `mepc.address[1]=1` and IALIGN=16 when writing the CSR

The capability tag is zero then IALIGN=32 when reading the CSR, or executing MRET (RVY), and the capability tag is one when IALIGN=16.

When a trap is taken into M-mode, the pc is written to `mepc.address` following the standard behavior. The capability tag and metadata of the pc are also written to mepc (RVY).

On execution of an MRET (RVY) instruction, the capability value from mepc (RVY) is unsealed and written to pc.

mepc (RVY) follows the rule from `mepc` about not needing to be able to hold all possible invalid addresses (see Invalid address conversion).

13.1.4. Machine Thread Identifier Capability (mtidc)

The mtidc register is used to identify the current software thread in machine mode, using the method defined in the section for the unprivileged utidc CSR. On reset the capability tag of mtidc will be set to zero and the remainder of the data is UNSPECIFIED.

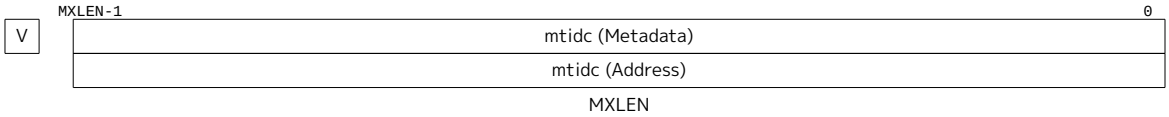


Figure 24. Machine thread identifier capability register

13.2. Machine-Level CSRs modified by RVY

13.2.1. Machine Status Registers (mstatus and mstatush)

The **mstatus** and **mstatush** registers have two additional restrictions:

- The MXL, SXL and UXL fields that control the value of XLEN for S-mode and U-mode must be read-only and equal to MXL in RVY implementations. Only 1 and 2 are supported values.
- The MBE, SBE, and UBE fields that control the memory system endianness for M-mode, S-mode, and U-mode must be read-only in RVY implementations. 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.

Changing XLEN or endianness would change the interpretation of all in-memory capabilities, so allowing these fields to change at runtime is prohibited.

These restrictions may be relaxed by a future extension. Such an extension is likely to enforce the constraint that any privilege level with XLEN less than MXLEN has CHERI disabled.

MXR has no effect on the CHERI permission checking.

CHERI does not need to make use execute only memory for security reasons, and so MXR has no relevance. Additionally the 32-bit encoding format does not allow X-permission to be

encoded without R-permission.

13.2.2. Machine Cause Register (mcause)

RVY adds new exception codes for CHERI exceptions that mcause must be able to represent. The new exception codes and priorities are listed in Machine cause (`mcause`) register values after trap and Table 79 respectively.

Table 79. Synchronous exception priority in decreasing priority order for RVY.

Priority	Exc.Code	Description
Highest	3	Instruction address breakpoint
	32	Prior to instruction address translation: CHERI Instruction Access Fault due to pc checks (capability tag, sealed, execute permission, bounds[1])
	12, 1	During instruction address translation: First encountered page fault or access fault
	1	With physical address for instruction: Instruction access fault
	2 32 0 8,9,11 3 3	Illegal instruction CHERI Instruction Access Fault due to pc ASR-permission clear Instruction address misaligned Environment call Environment break Load/store/AMO address breakpoint
	33,34	Prior to address translation for an explicit memory access: CHERI Load Access Fault, CHERI Store/AMO Access Fault due to capability checks (capability tag, sealed, permissions, bounds)
	4,6	Load/store/AMO capability address misaligned
	4,6	Optionally: Load/store/AMO address misaligned
	36, 13, 15, 5, 7	During address translation for an explicit memory access: First encountered CHERI Store/AMO Page Fault, 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	35	CHERI Load Capability Fault[2]

[1] pc bounds are checked against all bytes of fetched instructions. If the instructions could not be decoded to determine the length, then the pc bounds check is made against the minimum sized instruction supported by the implementation which can be executed, when prioritizing against Instruction Access Faults.

[2] CHERI Load Capability Fault is the lowest priority as determining whether to raise the exception may include checking the loaded capability tag.

The full details of CHERI Instruction Access Fault, CHERI Load Access Fault and CHERI Store/AMO Access Fault are in Table 80.

13.2.3. Machine Trap Delegation Register (medeleg)

Bits 32,33,34,35,36 of medeleg refer to valid CHERI exceptions and so can be used to delegate CHERI exceptions to supervisor mode.

13.2.4. "Smstateen/Ssstateen" Integration

The TID bit in `mstateen0` controls access to the stidc, vstidc, and utidc CSRs.

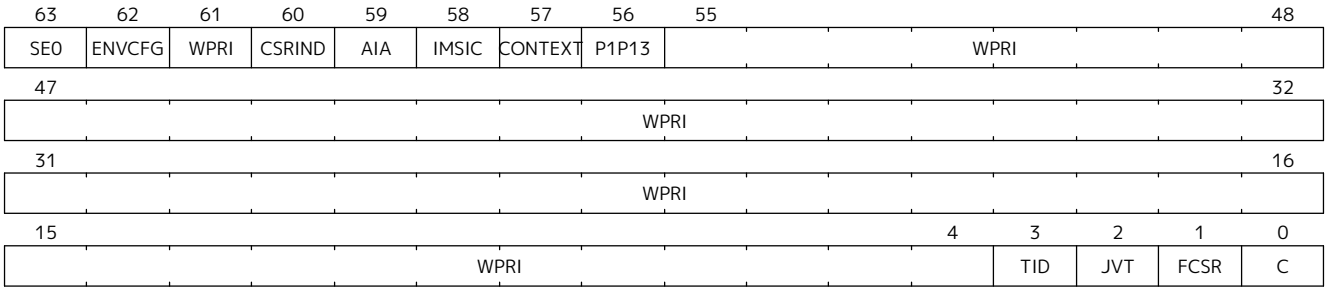


Figure 25. Machine State Enable 0 Register (`mstateen0`)

13.3. Supervisor-Level CSRs added or extended by RVY

RVY extends some of the existing RISC-V CSRs to be able to hold capabilities or with other new functions. ASR-permission in the pc is required for access to all privileged CSRs.

13.3.1. Supervisor Trap Vector Base Address Capability Register (stvec)

The stvec register is extended to hold a capability.

When the S-mode execution environment starts, the value is nominally the Root Executable capability.

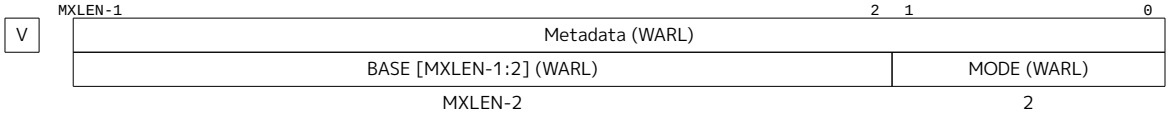


Figure 26. Supervisor trap-vector base-capability register

The handling of stvec (RVY) is otherwise identical to mtvec (RVY), but in supervisor mode.

13.3.2. Supervisor Scratch Capability Register (sscratch)

The sscratch register is extended to hold a capability.

At the start of the S-mode execution environment, the value of the capability tag of this CSR is zero and the values of the metadata and address fields are UNSPECIFIED.

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

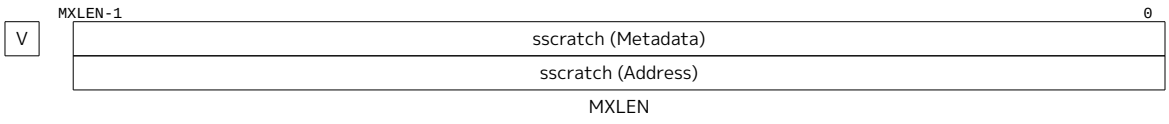


Figure 27. Supervisor scratch capability register

13.3.3. Supervisor Exception Program Counter Capability (sepc)

The sepc register is extended to hold a capability.

When the S-mode execution environment starts, the value is nominally the Root Executable capability.

As shown in Table 97, sepc (RVY) is a code capability, so it does not need to be able to hold all possible invalid addresses (see Invalid address conversion). Additionally, the capability in sepc (RVY) is unsealed when it is written to pc on execution of an SRET (RVY) instruction. The handling of sepc (RVY) is otherwise identical to mepc (RVY), but in supervisor mode.

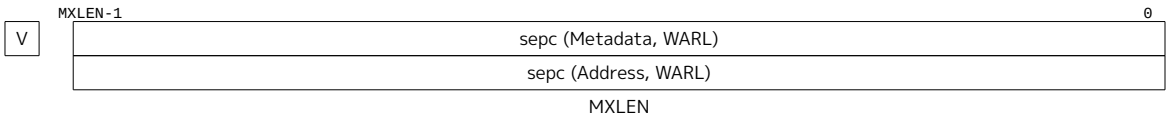


Figure 28. Supervisor exception program counter capability register

13.3.4. Supervisor Thread Identifier Capability (stidc)

The stidc register is used to identify the current software thread in supervisor mode, using the method defined in the section for the unprivileged utidc CSR.

At the start of the S-mode execution environment, the value of the capability tag of this CSR is zero and the values of the metadata and address fields are UNSPECIFIED.

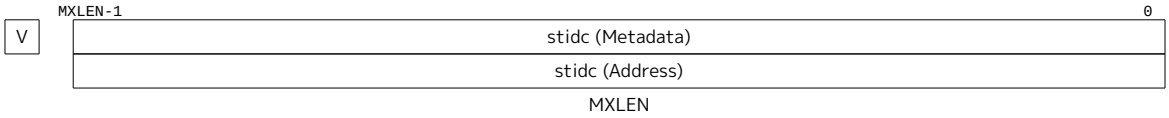


Figure 29. Supervisor thread identifier capability register

13.4. Supervisor-Level CSRs modified by RVY

13.4.1. Supervisor Cause Register (scause)

RVY adds new exception codes for CHERI exceptions that scause must be able to represent. The new exception code is listed in .Supervisor cause (`scause`) register values after trap. The behavior and usage of scause otherwise remains as described in scause.

See mcause (RVY) for the new exceptions priorities when RVY is implemented.

13.4.2. "Smstateen/Ssstateen" Integration

The TID (thread ID) bit in `sstateen0` controls access to the utidc CSR. See utidc for a description of the usage.

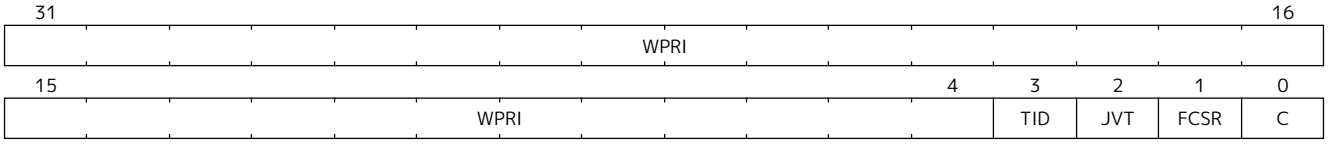


Figure 30. Supervisor State Enable 0 Register (`sstateen0`)

13.5. CHERI Exception handling

CHERI faults are typically higher priority than standard RISC-V faults. E.g., CHERI faults on the pc are higher priority than any other fault effecting the program counter such as instruction access fault.

`auth_cap` *is* `rs1`*, unless in (Non-CHERI) Address Mode when it is ddc (if Zyhybrid is implemented).*

Table 80. Valid CHERI exception combination description

Instructions	Xcause	Description	Check
All instructions have these exception checks first			
All	32	CHERI Instruction Access Fault	pc capability tag is zero
All	32	CHERI Instruction Access Fault	pc is sealed
All	32	CHERI Instruction Access Fault	pc does not grant X-permission
All	32	CHERI Instruction Access Fault	Any byte of current instruction out of pc bounds[1]
All	32	CHERI Instruction Access Fault	pc failed any integrity check.
CSR/Xret additional exception check			
CSR*, MRET (RVY), SRET (RVY), CBO.INVAL (RVY)	2	Illegal instruction	pc does not grant ASR-permission when required for CSR access or execution of MRET (RVY), SRET (RVY) or CBO.INVAL (RVY)
Load additional exception checks			
All loads	33	CHERI Load Access Fault	`auth_cap` capability tag is zero
All loads	33	CHERI Load Access Fault	`auth_cap` is sealed
All loads	33	CHERI Load Access Fault	`auth_cap` does not grant R-permission
All loads	33	CHERI Load Access Fault	Any byte of load access out of `auth_cap` bounds[1]
All loads	33	CHERI Load Access Fault	`auth_cap` failed any integrity check.
Capability loads	5[2]	Load access fault	Misaligned capability load
Store/atomic/cache-block-operation additional exception checks			
All stores, all atomics, all CBOs	34	CHERI Store/AMO Access Fault	`auth_cap` capability tag is zero
All stores, all atomics, all CBOs	34	CHERI Store/AMO Access Fault	`auth_cap` is sealed
All stores, CBO.ZERO	34	CHERI Store/AMO Access Fault	`auth_cap` does not grant W-permission
All atomics, CBO.CLEAN, CBO.FLUSH, CBO.INVAL	34	CHERI Store/AMO Access Fault	`auth_cap` does not grant both R-permission and W-permission
All stores, all atomics	34	CHERI Store/AMO Access Fault	any byte of access out of `auth_cap` bounds[1]
CBO.ZERO, CBO.INVAL	34	CHERI Store/AMO Access Fault	any byte of cache block out of `auth_cap` bounds[1]
CBO.CLEAN, CBO.FLUSH	34	CHERI Store/AMO Access Fault	all bytes of cache block out of `auth_cap` bounds[1]
All stores, all atomics, all CBOs	34	CHERI Store/AMO Access Fault	`auth_cap` failed any integrity check.

Instructions	Xcause	Description	Check
Capability stores	7[2]	Store access fault	Misaligned capability store

[1] The bounds checks include the cases where the bounds could not be decoded.

[2] Misaligned capability accesses raise access faults instead of misaligned faults since they cannot be emulated in software.

CBO.ZERO (RVY) is performed as a cache block wide store. All CMOs operate on the cache block which contains the address. Prefetch instructions check that the authorizing capability is has its capability tag set, is not sealed, has the required 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.

13.6. 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.

13.7. Physical Memory Attributes (PMA)

Typically, only parts of the entire memory space need to support CHERI capability tags. Therefore, it is desirable that harts supporting RVY extend PMAs with Physical Memory Attributes indicating whether a memory region allows storing CHERI capability tags. If they are not supported, then what the behavior is when attempting to access them.

There are three levels of support:

Table 81. CHERI PMAs

PMA	Load Behavior	Store Behavior	Comment
CHERI Capability Tag	Load capability tag	Store capability tag	Tagged memory supporting capability tags
CHERI Capability Tag Strip	Load zero capability tag	Ignore stored capability tag	No support for capability tags, ignore them
CHERI Capability Tag Fault	Load zero capability tag	Store/AMO Access Fault on capability tag[1]	No support for capability tags, trap on storing one

[1] The access fault is triggered on all capability stores or atomics such as SY or AMOSWAP.Y when C-permission and W-permission are granted and the to-be-stored capability tag is set to one. For SC.Y, the access fault is raised if the to-be-stored capability tag is set in rs2 even when the store fails.

Memory regions that do not have the *CHERI Capability Tag* PMA do not require storage for capability tags.

13.8. Modified Trap-Return Instructions Behavior

When the RVY base ISA is implemented, the trap-return instructions (`MRET` and `SRET`) read the full YLEN bits of the mepc (RVY)/sepc (RVY) register and unseal it prior to exception return if it is a sentry capability.

13.8.1. SRET (RVY)

See MRET (RVY).

13.8.2. MRET (RVY)

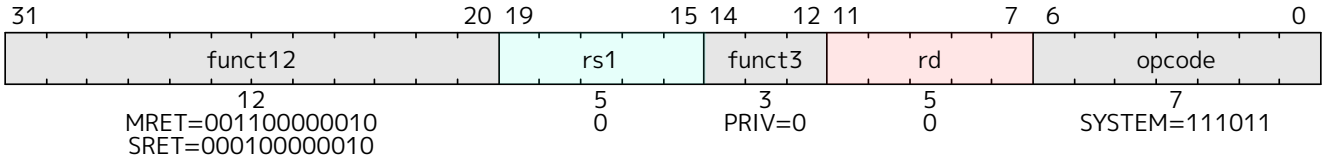
Synopsis

Trap Return (MRET, SRET)

Mnemonics

```
mret
sret
```

Encoding



Description

Return from machine mode (MRET (RVY)) or supervisor mode (SRET (RVY)) trap handler. MRET unseals mepc (RVY) and writes the result into pc. SRET unseals sepc (RVY) and writes the result into pc.

Exceptions

An illegal instruction fault is raised when pc does not grant ASR-permission because MRET (RVY) and SRET (RVY) require access to privileged CSRs.

Prerequisites (MRET)

Machine-Level ISA, RVY

Prerequisites (SRET)

Supervisor-Level ISA, RVY

Operation

```
TBD
```

Chapter 14. "Zyhybrid for Privileged Architectures" Extension, Version 1.0

When using a system with Zyhybrid, it may be desirable to disable CHERI register and instruction access to some (or all) privilege levels such that they operate as a RV32I/RV64I system without any observable presence of CHERI features. Zyhybrid includes functions to disable explicit access to CHERI registers and instructions (hereafter referred to as *disabling CHERI*).

When CHERI is disabled for a specific privilege level, pc and ddc bounds are still enforced.

The Zyhybrid extension makes the Y bit of misa, menvcfg, and senvcfg writable to allow CHERI to be disabled.

The effective CHERI-enable for the current privilege is:

- Machine: `misa.Y`
- Supervisor: `misa.Y & menvcfg.Y`
- User: `misa.Y & menvcfg.Y & senvcfg.Y`

On reset CHERI is always disabled for backwards compatibility (misa.Y resets to zero, ddc and pc bounds are nominally root capabilities (see Root).

The following occurs when executing code in a privilege mode that has CHERI disabled:

- Instructions from RVY and Zyhybrid cause illegal instruction exceptions.
- Executing CSR instructions accessing any natively YLEN CSR causes an illegal instruction exception.
- Executing CSR instructions accessing any CSR extended to YLEN only allows XLEN access (this is identical to *(Non-CHERI) Address Mode* access).

Disabling CHERI has no effect on implicit accesses or security checks. The last capability written to pc and ddc before disabling CHERI will be used to authorize instruction execution and data memory accesses.

Disabling CHERI prevents low-privileged (Non-CHERI) Address Mode software from interfering with the correct operation of higher-privileged (Non-CHERI) Address Mode software that does not perform ddc switches on trap entry and return.

Disabling CHERI allows harts supporting CHERI to be fully compatible with standard RISC-V, so CHERI instructions, such as YAMASK, that do not change any CHERI state, raise exceptions. This is the default behavior on reset.

Table 82 summarizes the behavior of a hart in connection with the effective CHERI enable and the CHERI Execution Mode while in a privilege other than debug mode.

Table 82. Hart's behavior depending on the effective CHERI enable and CHERI Execution Mode

	Y[1]=0	Y=1, M-bit=1	Y=1, M-bit=0
Authorizing capability for data memory accesses	ddc	ddc	capability in `rs1`
natively YLEN CSR Access Width	✘	YLEN	YLEN
Extended YLEN CSR Access Width	XLEN	XLEN	YLEN
CHERI Instructions Allowed	✘	✔[3]	✔

	Y[1]=0	Y=1, M-bit=1	Y=1, M-bit=0
Summary	*Fully RISC-V compatible*[2]	*(Non-CHERI) Address Mode*	*(CHERI) Capability Mode*

[1] *Y* represents the effective CHERI enable for the current privilege mode.

[2] The hart is fully compatible with standard RISC-V when Y=0 provided that pc, Xtvec, Xepc and ddc have not been changed from the default reset state (i.e., hold Root Executable and Root Data capabilities).

[3] The compressed instructions operating on capability data are unavailable as their encoding will revert to non-CHERI standard behavior.

Chapter 15. "Supervisor-Level ISA for Virtual Memory (RV64Y)" Extension, Version 1.0 for RV64Y

CHERI checks are made on the effective address according to the current translation scheme. I.e., on the virtual address if translation is enabled or the physical address if translation is disabled.

Implicit memory accesses made by the page table walker are not subject to CHERI checks.

A future extension may add CHERI checks to the page table walker.

15.1. 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*

15.2. CHERI Store/AMO Page Fault

RV64Y adds the the following new fault type:

- CHERI Store/AMO Page Fault (cause value 36)

It is prioritized against other fault types as shown in Table 79.

15.3. The *pte.rvy* field for capability flow control

As RV32Y has no field allocated in the PTE, capability loads and stores operate as normal. Therefore the RVY field is reserved in the RV64 PTE formats only (e.g. Figure 65) for use by RV64Y extensions. The principle is that extensions, such as Svyrg may redefine the usage of the RVY field by adding an enable bit into `Xstatus`.

The definition of the RVY field is as follows:

Table 83. Definition of the pte.rvy field

Bit	Name	Comment
pte.rvy[3]	*pte.y*	Enable capability (Y) access
pte.rvy[2]	*reserved*	*reserved*

Bit	Name	Comment
pte.rvy[1]	*reserved*	*reserved*
pte.rvy[0]	*reserved*	*reserved*

If *pte.y*=0 then:

- All capability loads set the loaded capability tag to zero
- All capability stores with the to-be-stored capability tag set raise a CHERI Store/AMO Page Fault.

If *pte.y*=1 then:

- All capability loads and capability stores operate as normal.

15.4. Invalid Virtual Address Handling

When address translation is in effect for RV64Y, the upper bits of virtual memory addresses must match for the address to be valid.

The CSRs shown in Table 97, as well as the 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.

However, the bounds encoding of capabilities depends on the address value if the bounds are not infinite.

Therefore implementations must not convert invalid addresses to other arbitrary invalid addresses in an unrestricted manner if the bounds are not infinite.

If the bounds could not be decoded due to the address being invalid, then a *CHERI Instruction Access Fault*, *CHERI Load Access Fault* or *CHERI Store/AMO Access Fault* exception is raised as appropriate.

In all cases, if the authorizing capability has bounds that cover all addresses, then the behavior is identical to the normal RISC-V behavior without CHERI.

Not requiring the implementation to decode the bounds for invalid addresses reduces the size of bounds comparators from 64-bits to the supported virtual address width.

15.4.1. Updating CSRs

A CSR may be updated to hold a capability with an invalid address, due to:

- executing instructions, such as CSRRW (RVY)
- hardware updates to CSRs such as storing the pc into mepc (RVY)/sepc (RVY) etc. when taking an exception.

To ensure that the bounds of a valid capability cannot be corrupted:

- If the new address is invalid and the capability bounds do not cover all addresses, then set the capability tag to zero before writing to the CSR.

When the capability's address is invalid and happens to match an invalid address which the CSR can hold, then it is implementation-defined whether to set the capability tag to zero.

15.4.2. Branches and Jumps

If the effective target address of the jump or branch is invalid, and the authorizing capability's bounds do not cover all addresses, then set the capability tag of the target pc to zero. This will cause a CHERI Instruction Access Fault exception when executing the target instruction.

RISC-V harts that do not support RVY normally raise an instruction access fault or page fault after jumping or branching to an invalid address. Therefore, RVY aims to preserve that behavior to ensure that harts supporting RVY and Zyhybrid are fully compatible with RISC-V harts provided that pc and ddc are set to Root Executable and Root Data capabilities, respectively.

15.4.3. Memory Accesses

If the effective address of the memory access is invalid, and the authorizing capability's bounds do not cover all addresses, then raise a CHERI Load Access Fault or CHERI Store/AMO Access Fault exception because the bounds cannot be reliably decoded.

15.5. Integrating RVY with Debug

15.5.1. Integrating RVY with Sdext

This chapter will appear in the priv spec. Exact location TBD.

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

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

- The `hartinfo` register must be implemented.
- All harts which support RVY must provide `hartinfo.nscratch` of at least 1 and implement the dscratch0 (RVY) register.
- All harts which support RVY 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 `x1` *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  x2, dscratch0, x2
yhir   x2, x1
csrw   0xBF0, x2
csrrw  x2, dscratch0, x2
ebreak
```

```
# Read the capability tag into data0
csrrw  x2, dscratch0, x2
ytagr  x2, x1
csrw   0xBF0, x2
csrrw  x2, dscratch0, x2
ebreak

# Write the high MXLEN bits from data0-data1
csrrw  x2, dscratch0, x2
csrr   x2, 0xBF0
yhiw   x1, x1, x2
csrrw  x2, dscratch0, x2
ebreak

# Write the capability tag (if nonzero)
csrrw   x2, dscratch0, x2
csrr    x2, drootc
ybld    x1, x2, x1
csrrw   x2, dscratch0, x2
ebreak
```

The low `MXLEN` *bits of a capability are read and written using normal Access Register abstract commands. If dscratch0 (RVY) 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.*

15.5.1.1. 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 the *RISC-V Debug Specification* apply.

15.5.1.2. Core Debug Registers

RVY 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 99.

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

This specification extends the following registers from the *RISC-V Debug Specification*.

Debug Program Counter (dpc)

dpc is a DXLEN-bit register used as the PC saved when entering debug mode.

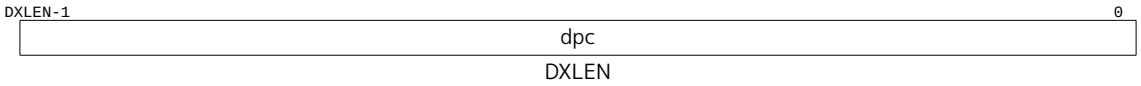


Figure 31. Debug program counter

Debug Scratch Register 1 (dscratch1)

dscratch1 is an optional DXLEN-bit scratch register that can be used by implementations which need it.

Figure 32. Debug scratch 0 register

Debug Scratch Register 1 (dscratch1)

dscratch1 is an optional DXLEN-bit scratch register that can be used by implementations which need it.

Figure 33. Debug scratch 1 register

15.5.1.3. Debug Program Counter Capability (dpc)

The dpc register is extended to hold a capability.

The reset value of the capability tag of this CSR is zero, the reset values of the metadata and address fields are UNSPECIFIED.

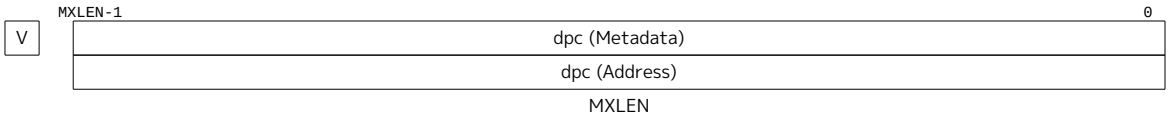


Figure 34. Debug program counter capability

Upon entry to debug mode, the *RISC-V Debug Specification*, 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 pc, which refers to all jumps, conditional branches and AUIPC (RVY). The exceptions are YMODESWY and YMODESWI, which *are* supported if Zyhybrid is implemented, see drootc for details.

As a result, the value of pc is UNSPECIFIED in debug mode according to this specification. The pc metadata has no architectural effect in debug mode. Therefore ASR-permission is implicitly granted for access to all CSRs for instruction execution.

On debug mode entry, dpc (RVY) is updated with the capability in pc whose address field is set to the address of the next instruction to be executed upon debug mode exit as described in the *RISC-V Debug Specification*.

When leaving debug mode, an unsealed capability value is copied from the value in dpc (RVY) and written into pc. A debugger may write dpc (RVY) to change where the hart resumes and its mode, permissions, sealing or bounds.

The legalization of dpc (RVY) follows the same rules described for mepc (RVY).

15.5.1.4. Debug Scratch Register 0 (dscratch0)

The dscratch1 register is extended to hold a capability.

The reset value of the capability tag of this CSR is zero, the reset values of the metadata and address fields are UNSPECIFIED.

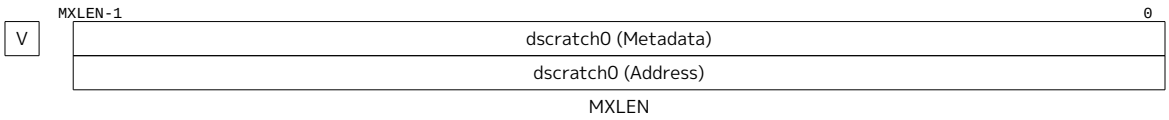


Figure 35. Debug scratch 0 capability register

15.5.1.5. Debug Scratch Register 1 (dscratch1)

The dscratch1 register is extended to hold a capability.

The reset value of the capability tag of this CSR is zero, the reset values of the metadata and address fields are UNSPECIFIED.

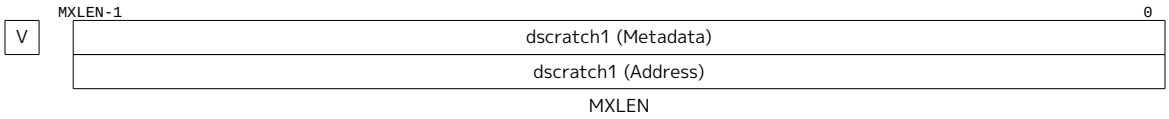


Figure 36. Debug scratch 1 capability register

15.5.1.6. Debug Root Capability Selector (drootcsel)

drootcsel is a debug mode accessible integer CSR. The address and access details are shown in Table 98.

It selects which Root capability is exposed through drootc. The reset value is `0`, which must cause drootcsel to expose a Root Executable capability.

Other capability values may be defined for exposure through drootc by the capability encoding, and may be selected by having the debugger write to this register. Writes are WARL, so the debugger may confirm that its selection has been applied.

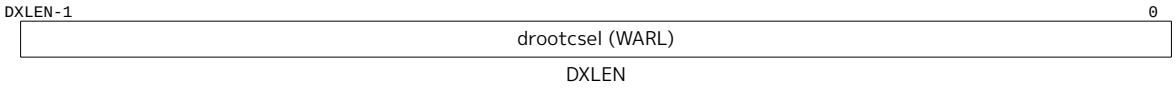


Figure 37. Debug root capability register

15.5.1.7. Debug Root Capability Register (drootc)

drootc is a debug mode accessible capability CSR. The address and access details are shown in Table 98. It exposes the capability selected by drootcsel.

If Zyhybrid is implemented, the Root Executable exposed when drootcsel is `0` is further specified as follows:

- The M-bit is reset to *(Non-CHERI) Address Mode* (1).
- The debugger can set the M-bit to *(CHERI) Capability Mode* (0) by executing YMODESWY from the program buffer.
  - Executing YMODESWY causes execution of subsequent instructions from the program buffer, starting from the next instruction, to be executed in *(CHERI) Capability Mode*. It also sets the CHERI execution mode to *(CHERI) Capability Mode* on future entry into debug mode.
  - Therefore to enable use of a CHERI debugger, a single YMODESWY only needs to be executed once from the program buffer after resetting the core.

- The debugger can also execute YMODESWI to change the mode back to *(Non-CHERI) Address Mode*, which also affects the execution of the next instruction in the program buffer, updates the M-bit of this capability and controls which CHERI execution mode to enter on the next entry into debug mode.

The M-bit of this capability is *only* updated by executing YMODESWY or YMODESWI from the program buffer.

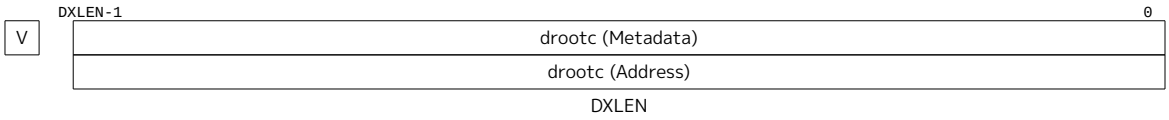


Figure 38. Debug root capability register

15.5.1.8. Modified Trap-Return Instruction Behavior

The `DRET` instruction reads the full YLEN bits of the mepc (RVY)/sepc (RVY) register and unseals it prior to exception return if it is a sentry capability.

15.5.1.8.1. DRET (RVY)

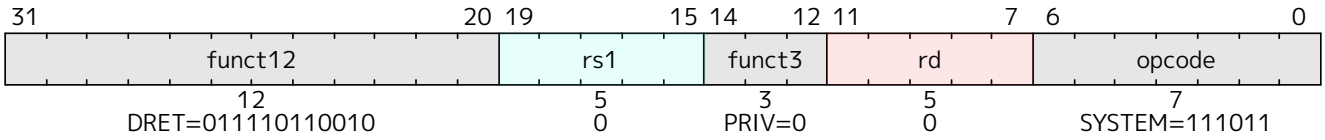
Synopsis

Debug Return (DRET)

Mnemonic

`dret`

Encoding



Description

DRET (RVY) returns from debug mode. It unseals dpc (RVY) and writes the result into pc.

> *The DRET (RVY) instruction is the recommended way to exit debug mode. However, it is a pseudoinstruction to return that technically does not execute from the program buffer or memory. It currently does not require the pc to grant ASR-permission so it never raises an exception.*

Prerequisites

Sdext, RVY

Operation

```
TBD
```

15.5.2. Integrating Zyhybrid with Sdext

A new debug default data capability (dddc) CSR is added at the CSR number shown in Table 42.

Zyhybrid allows YMODESWY and YMODESWI to execute in debug mode.

When entering debug mode, whether the core enters *(Non-CHERI) Address Mode* or *(CHERI) Capability Mode* is controlled by the M-bit in the drootc capability selected by drootcsel value 0.

The current mode can be read by setting drootcsel to `0` and then reading drootc.

The following sequence executed from the program buffer will write 0 for *(CHERI) Capability Mode* and 1 for *(Non-CHERI) Address Mode* to `x1`:

```
csrr   x1, drootc
ymoder x1, x1
```

> *There is no CHERI enable/disable bit for debug mode, so CHERI register and instruction access is always permitted in debug mode.*

15.5.2.1. Debug Default Data Capability CSR (dddc)

dddc is a debug mode accessible capability CSR. The address is shown in Table 42.

The reset value of the capability tag of this CSR is zero, the reset values of the metadata and address fields are UNSPECIFIED.

This CSR is only implemented if Zyhybrid is implemented.

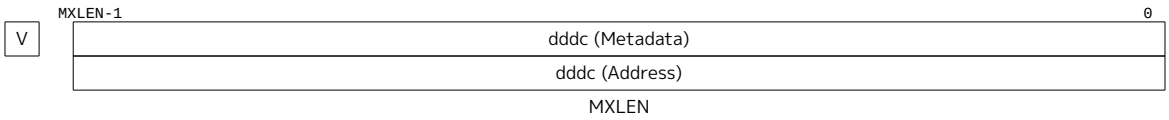


Figure 39. Debug default data capability

Upon entry to debug mode, ddc is saved in dddc. ddc is set to a Root Data capability such that ddc's address remains unchanged.

When debug mode is exited by executing DRET (RVY), 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 97, dddc is a data pointer, so it does not need to be able to hold all possible invalid addresses (see Invalid address conversion).

15.5.3. "Sdtrig (RVY)", Integrating RVY with Sdtrig

This chapter will appear in the priv spec. Exact location TBD.

The Sdtrig extension is orthogonal to RVY. However, the priority of synchronous exceptions and where triggers fit is adjusted as shown in Table 84.

Debug triggers are higher priority than CHERI exceptions to allow debug.

Table 84. Synchronous exception priority in decreasing priority order.

Priority	Exception 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
	32	Prior to instruction address translation: CHERI Instruction Access Fault due to pc checks (capability tag, sealed, execute permission, bounds[1])	
	12, 20, 1	During instruction address translation: First encountered page fault, guest-page fault, or access fault	
	1	With physical address for instruction: Instruction access fault	

Priority	Exception Code	Description	Trigger
	3		mcontrol/mcontrol6 execute data before
	2 32 22 0 8, 9, 10, 11 3 3	Illegal instruction CHERI Instruction Access Fault due to pc ASR-permission clear Virtual instruction Instruction address misaligned Environment call Environment break Load/Store/AMO address breakpoint	mcontrol/mcontrol6 load/store address before, store data before
	33,34	Prior to address translation for an explicit memory access: CHERI Load Access Fault, CHERI Store/AMO Access Fault due to capability checks (capability tag, sealed, permissions, bounds)	
	4,6	Load/store/AMO capability address misaligned	
	4, 6	Optionally: Load/Store/AMO address misaligned	
	36, 13, 15, 21, 23, 5, 7	During address translation for an explicit memory access: First encountered CHERI Store/AMO Page Fault, page fault, guest-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	
	35	CHERI Load Capability Fault	
Lowest	3		mcontrol/mcontrol6 load data before

See the notes beneath Synchronous exception priority in decreasing priority order for details about CHERI load page fault priority.

Chapter 16. Pointer Masking (Ssnpm, Smnpm, Smmpm, Sspm, Supm) (RV64Y)

This chapter is not part of the v1.0 ratification package.

Whenever pointer masking is enabled, all bounds decoding, representable range checks and bounds checks are affected.

The suggestion in this section is based on the pointer masking approach from Morello but with changes to sign extension and to address the dynamic nature of bit masking.

When bounds are encoded or decoded, a masked but *not* sign-extended address is used. Changing how many bits are masked can therefore change the interpretation of the bounds of a capability, both for the purpose of implicit accesses via bounds checks and any instructions that report the bounds of a capability. Apart from treating the PMLEN high address bits as zero, there are no other changes to bounds decoding, which is still based on XLEN, not the new effectively addressable space. That is, the maximum length of a capability does not change, and it is not invalid to have a capability that covers a longer range than could actually be addressed with pointer masking enabled (such as one covering the entire XLEN address space). For the representable range check, both the original and new addresses are masked. Bounds setting instructions also mask the address in the same way.

Because dynamically changing the number of masked bits changes the interpretation of a capability, software must take the same care when sharing capabilities between address spaces with differing pointer masking as it generally must when sharing capabilities between address spaces with different page mappings.

Any address that is checked against a capability is also first subject to the same masking as bounds decode (masking without extension). After any CHERI operations, the final access address is still subject to as much sign extension as the pointer masking extensions mandate.

In summary, for data accesses only:

- When setting bounds (YBNDSW/YBNDSWI/YBNDSRW), bits [XLEN-1:XLEN-PMLEN] of the address are set to zero and therefore the resulting capability will have a base with the PMLEN high address bits set to zero.
- When decoding bounds, the address used for decoding has bits [XLEN-1:XLEN-PMLEN] set to zero.
- When checking the representable range for YADDI/YADD/YADDRW, the old address and new addresses both have bits [XLEN-1:XLEN-PMLEN] set to zero **before** the check.

Also note that:

- pc bounds decoding isn't affected by pointer masking.
- Any representable range check triggered by a write to a CSR (e.g. mtvec (RVY)) is not affected by pointer masking.

Chapter 17. "Svyrg" Extension, Version 1.0 for RV64Y

This extension specifies use of the RVY field from the PTE formats (e.g., see Figure 65).

Svyrg implies Sv39.

The Svyrg extension is enabled when the `sstatus.YRGE` bit is set.

The extension adds the ability for supervisor-mode software to quickly enable trapping loads of capabilities from all pages of memory, incrementally allow loads of capabilities from such pages, and track stores of capabilities to all pages of memory. Applied to userspace pages, this has been shown to allow the operating system to implement capability revocation schemes, which allow userspace memory allocators to deterministically guard against use-after-reallocation (Filardo et al., 2024). Applied to kernel pages, they can similarly be used against use-after-reallocation within the kernel itself.

The 4-bit *pte.rvy* field is subdivided into four 1-bit fields, two controlling capability loads (including the capability loaded during an AMO) and two controlling capability stores (again, including the capability stored as part of an AMO).

Table 85. Breakdown of the 4-bit pte.rvy field.

Bit	Name	Comment
Capability load/AMO fields		
pte.rvy[0]	*pte.yr*	Capability readable
pte.rvy[1]	*pte.yrg*	Capability read generation
Capability store/AMO fields		
pte.rvy[2]	*pte.yw*	Capability writable
pte.rvy[3]	*pte.yd*	Capability dirty

When all of the following are true, Svyrg adds two related architectural features, CHERI Load Capability Faults and Capability Dirty Tracking:

1. `sstatus.YRGE` is set.
2. The authorizing capability has C-permission.
3. The PMA is set to *CHERI Capability Tag*.

17.1. CHERI Load Capability Faults

Svyrg defines a new fault type:

- CHERI Load Capability Fault (cause value 35)

In the case of AMOs that could trigger both a CHERI Store/AMO Page Fault, due to storing a valid capability tag, and a CHERI Load Capability Fault, the CHERI Store/AMO Page Fault takes priority as shown in Table 79

pte.yr and *pte.yrg* are used to enable capability loads or AMOs to write a capability tag to rd.

When *pte.yr* is clear, *pte.yrg* controls whether capabilities can write a capability tag to rd. When clear capability tags cannot be written to rd, and when set, they can.

When *pte.yr* is set, *pte.yrg* can be used to trap on capability loads or AMOs when it does not match the

Capability Read Generation value that is represented by the value of `sstatus.UYRG` for userspace pages (*pte.u=1*) or `sstatus.SYRG` for kernel pages (*pte.u=0*).

The implementation raises a CHERI Load Capability Fault when, in addition to the rules above, all of the following are true:

1. A capability load or AMO is executed.
2. *pte.yr* is set.
3. if *pte.u=1*, *pte.yrg* does not equal `sstatus.UYRG`.
4. if *pte.u=0*, *pte.yrg* does not equal `sstatus.SYRG`.
5. Optionally, the loaded capability tag is set[1].
6. Any other platform specific rules have not forced the loaded capability tag to be clear.

[1]Checking the value of the capability tag requires taking data dependent exceptions on loaded capabilities for loads or AMOs. Ideally all implementations would trap precisely (taking the capability tag into account in all cases) rather than conservatively (trapping more often, potentially on every loaded capability). However, the loaded capability tag may not be available in all implementations when determining whether to raise the exception, and therefore flexibility is permitted. As a result, the software is required to be tolerant of raising the trap when the capability tag is not set, potentially resulting in spurious traps from pages that have pte.yr=1, and so are likely to store valid capabilities.

Implementations that already take synchronous traps on loaded data, such as ECC faults, are recommended to check the loaded capability tag when determining whether to raise the fault.

Table 86. Summary of capability load pte.yr and pte.yrg behavior in the PTEs

pte.yr	*pte.yrg*	tag[1]	Load Capability Behavior
0	0	X	Clear loaded capability tag
0	1	X	Normal operation
1	≠ `sstatus.xYRG`[2]	0	Implementation defined choice of CHERI Load Capability Fault or normal operation
1	≠ `sstatus.xYRG`[2]	1	CHERI Load Capability Fault
1	= `sstatus.xYRG`[2]	X	Normal operation

[1] The loaded capability tag.

[2] if *pte.u=1*, `sstatus.UYRG`, else `sstatus.SYRG`.

CHERI Load Capability Faults may be used to implement the load-barrier primitive from (Filardo et al., 2024).

17.2. Capability Dirty Tracking

When *pte.yw* is clear, capability stores or AMOs where the to-be-stored capability tag is set will raise a CHERI Store/AMO Page Fault fault.

When *pte.yw* is set, capability stores to the virtual page are permitted. In addition, the *pte.yd* bit indicates that a capability was stored to the virtual page since the last time the *pte.yd* bit was cleared.

This is akin to the pte.d bit indicating that a store occurred through this PTE since the last

time pte.d was cleared.

Capability dirty tracking behavior is enabled when, in addition to the rules above, all of the following are true:

1. A capability store or AMO instruction is executed.
2. The to-be-stored capability tag is set.
3. *pte.yw* is set.
4. *pte.yd* is clear.

Two schemes for capability dirty tracking are permitted, and the scheme in use is determined by whether the *Svade* or *Svadu* extensions are enabled.

- For *Svade*, take a CHERI Store/AMO Page Fault.
- For *Svadu*, do a hardware update that sets *pte.yd=1*, following the same rules as setting *pte.d*.
  - When setting *pte.yd*, the hardware update also necessarily sets (or leaves set) *pte.a* and *pte.d*.

Table 87. Summary of capability store pte.yw and pte.yd behavior in the PTEs

pte.yw	*pte.yd*	tag[1]	Store Capability Behavior
0	X	0	Normal operation
0	X	1	CHERI Store/AMO Page Fault
1	0	0	Normal operation
1	0	1	CHERI Store/AMO Page Fault (*Svade*) or hardware *pte.yd* update (*Svadu*)
1	1	X	Normal operation

[1] The to-be-stored capability tag.

The capability dirty tracking is resolved during memory translation, but there are cases where it is not known if there will be a capability tag stored to memory or not at this point. Capability dirty tracking *must* be triggered when *pte.yw*=1 and *pte.yd*=0 and the to-be-stored capability tag is set, and *may* be triggered in the following case where it is not known if there will be a stored capability tag during translation:

- SC.Y triggers capability dirty tracking if the capability tag is set in rs2, even if the store fails. This matches the semantics of SC.* with regard to *pte.d*.

> *pte.yd must always be set when a capability with a valid capability tag is written to a virtual page so that software knows which pages have had capabilities stored to them. It may be set too often, which may cause software to examine the page to check for capabilities when none are present. This is a situation software is required to handle anyway, as it is always possible for all capabilities in a page to be overwritten by non-capability data. In this case the pte.yd bit would still be set.*
>
> 
>
> *Future AMOs fall into this category:*
>
> - *For future AMOCAS.Y, it is not known whether the store will happen until the load has executed, and the compare has been done. Therefore, AMOCAS.Y is likely to trigger capability dirty tracking if the capability tag is set in rs2.*
> - *For future AMOADD.Y, the stored capability tag depends upon the loaded capability tag which is not known during translation and on the execution of the YADD. Therefore,*

AMOADD.Y, is likely to always trigger capability dirty tracking.

Checking the stored capability tag is less of a burden to the implementation than checking the loaded capability tag for CHERI Load Capability Fault, which is why checking the loaded capability tag is optional behavior. However, a future extension may reduce the burden further by removing the check on the to-be-stored capability tag.

Capability dirty tracking may be used to implement the store-barrier primitive from (Filardo et al., 2024).

The minimum level of PTE support is to set pte.yr=1, pte.yd=1, pte.yw=1 and pte.yrg=0 in all PTEs intended for storing capabilities (e.g., private anonymous mappings) and set `sstatus.UYRG`*=0 and* `sstatus.SYRG`*=0 on all harts, which will enable capabilities to be loaded and stored successfully.*

17.3. UYRG CSR field

The `mstatus`, `sstatus` and `vsstatus` CSRs include the Userspace Capability Read Generation (UYRG) bit, the Supervisor version (SYRG) and the enable bit (YRGE).

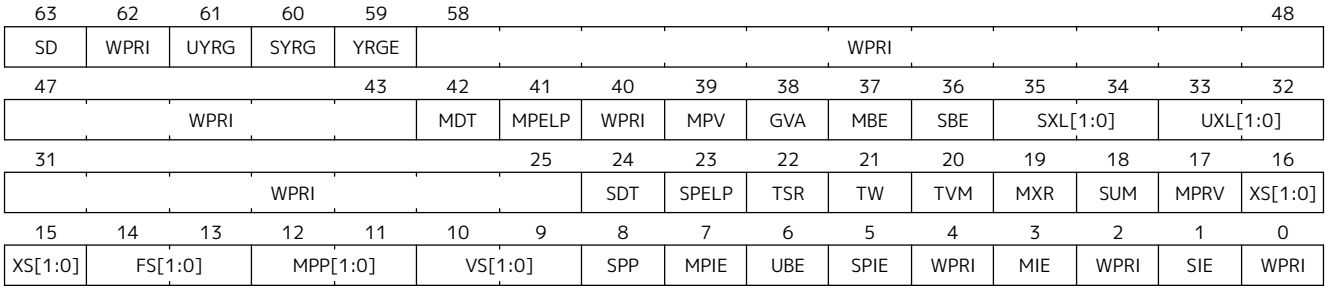


*Figure 40. Machine-mode status (**mstatus**) register for RV64Y*

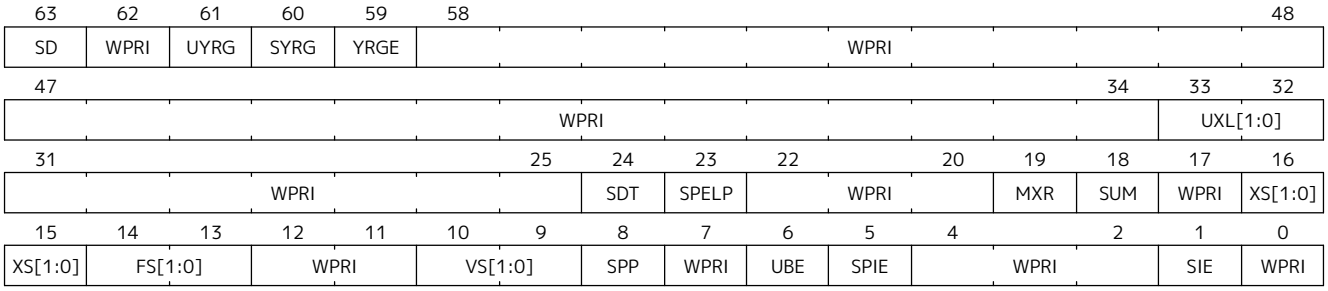


*Figure 41. Supervisor-mode status (**sstatus**) register when SXLEN=64*

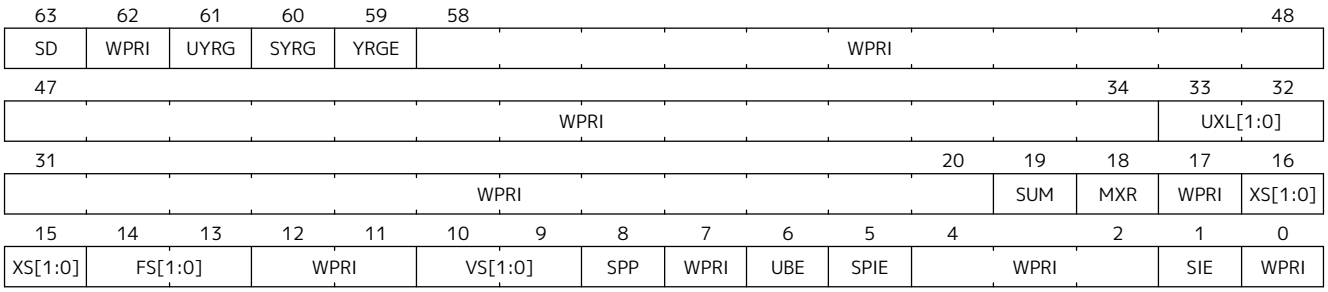


*Figure 42. Virtual Supervisor-mode status (**vsstatus**) register when VSXLEN=64*

Chapter 18. Hypervisor "H" Extension (RVY)

This chapter is not part of the v1.0 ratification package.

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 RVY and Zyhybrid, are that address CSRs added for hypervisors are extended to YLEN size. The remainder of this chapter describes these changes in detail.

18.1. Hypervisor Status Register (hstatus)

The hstatus register operates as described above 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 RVY (but not Zyhybrid), then hstatus's VSXL must be read-only as described in mstatus for mstatus.SXL. When the implementation supports both RVY and Zyhybrid, then VSXL behaves as described in mstatus (RVY) for mstatus.SXL.

The VSBE field 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 RVY.

18.2. Hypervisor Environment Configuration Register (henvcfg)

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

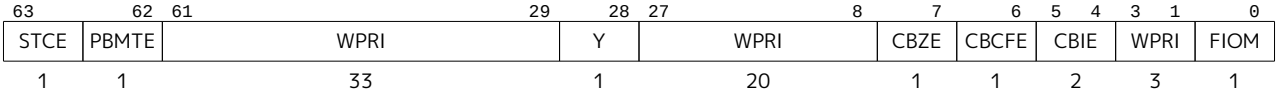


*Figure 43. Hypervisor environment configuration register (**henvcfg**)*

The Y bit controls whether explicit access to CHERI registers is permitted when V=1. When henvcfg(RVY).Y=1 and menvcfg.Y=1 and misa.Y=1, CHERI can be enabled by VS-mode and VU-mode. When henvcfg(RVY).Y=0, CHERI is disabled in VS-mode and VU-mode as described in Chapter 14.

The Y bit is reset to 0 for compatibility, so that non-CHERI aware code can run unmodified.

18.3. Hypervisor Exception Delegation Register (hedeleg)

Bits 32,33,34,35,36 of hedeleg refer to valid CHERI exceptions and so can be used to delegate CHERI exceptions to supervisor mode.

18.4. Virtual Supervisor Status Register (vsstatus)

The vsstatus register operates as described above 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 RVY (but not Zyhybrid), then vsstatus.UXL must be read-only as described in mstatus for mstatus.UXL. When the implementation supports both RVY and Zyhybrid, then UXL behaves as described in mstatus (RVY) for mstatus.UXL.

18.5. Virtual Supervisor Trap Vector Base Address Capability Register (vstvec)

The `vstvec` register is extended to hold a capability.

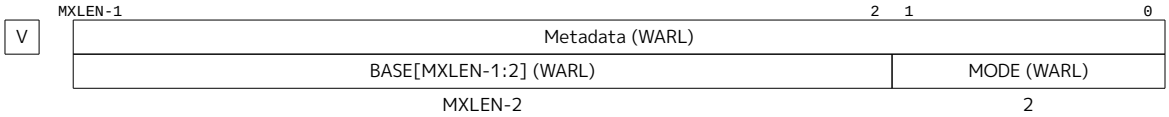


Figure 44. Virtual supervisor trap vector base address capability register

The handling of vstvec (RVY) is otherwise identical to mtvec (RVY), but in virtual supervisor mode.

18.6. Virtual Supervisor Scratch Register (vsscratch)

The `vsscratch` register is extended to hold a capability.

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

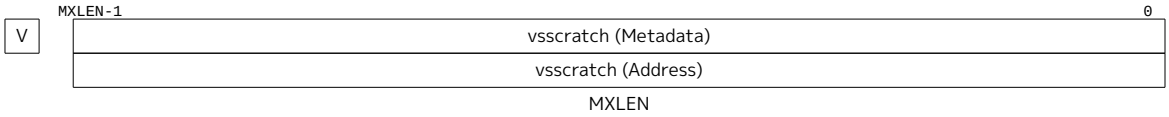


Figure 45. Virtual supervisor scratch capability register

18.7. Virtual Supervisor Exception Program Counter Capability (vsepc)

The `vsepc` register is extended to hold a capability.

As shown in Table 97, vsepc (RVY) is a code capability, so it does not need to be able to hold all possible invalid addresses (see Invalid address conversion). Additionally, the capability in vsepc (RVY) is unsealed when it is written to pc on execution of an SRET (RVY) instruction when V=1. The handling of vsepc (RVY) is otherwise identical to mepc (RVY), but in VS-mode.

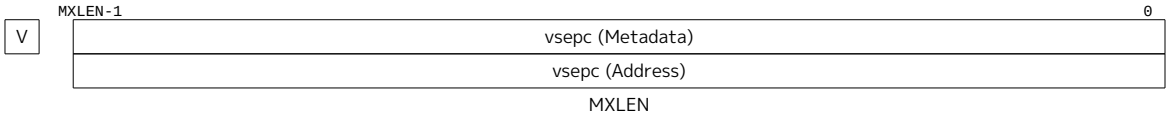


Figure 46. Virtual supervisor exception program counter capability

18.8. 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 and CHERI page faults that are delegated to VS-mode.

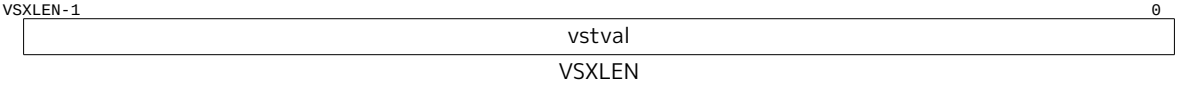


Figure 47. Virtual supervisor trap value register

18.9. Virtual Supervisor Thread Identifier Capability (vstidc)

The vstidc register is used to identify the current software thread in virtual supervisor mode. As with other Virtual Supervisor registers when V=1, vstidc substitutes for stidc, so that instructions that normally read or modify stidc actually access vstidc instead. When V=0, vstidc does not directly affect the behavior of the machine.

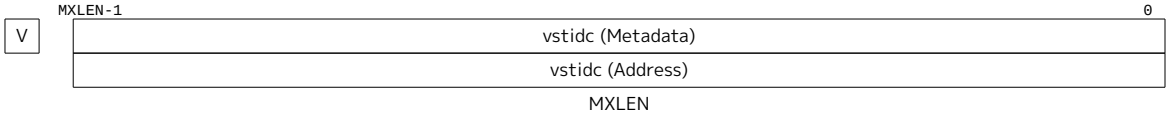


Figure 48. Virtual supervisor thread identifier capability register

18.10. "Smstateen/Ssstateen" Integration

The new TID bit controls access to the stidc (really vstidc) CSR.

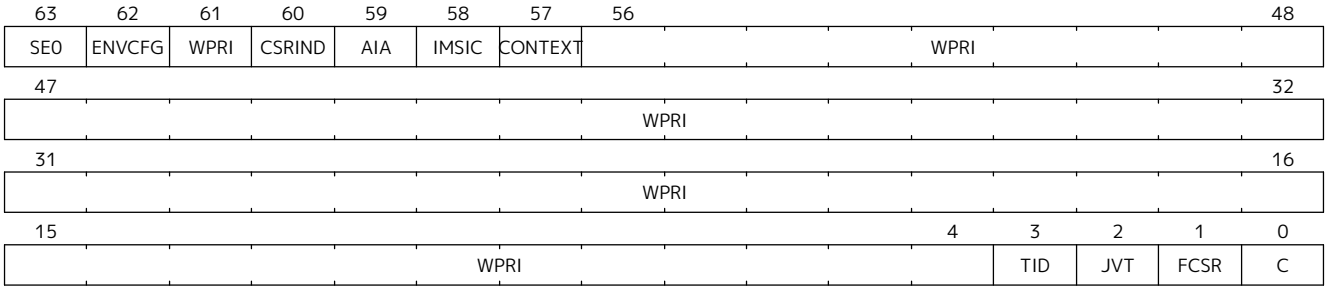


Figure 49. Hypervisor State Enable 0 Register (`hstateen0`)

18.11. Hypervisor Load and Store Instructions For Capability Data

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

When in *(CHERI) Capability Mode*, the hypervisor load and store capability instructions behave as described in Section 2.9. In *(Non-CHERI) Address Mode*, the instructions use the low XLEN bits of the base register as the effective address for the memory access and the capability authorizing the memory access is ddc.

18.11.1. HLV.Y

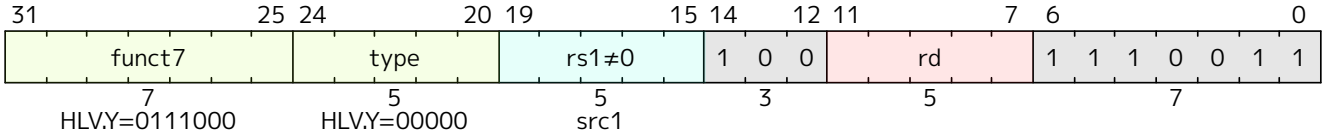
Synopsis

Hypervisor virtual-machine load capability

Mnemonic

```
hlv.y rd, rs1
```

Encoding



Any instance of this instruction with `rs1`=`x0` *will raise an exception, as* `x0` *is defined to always hold a NULL capability. As such, the encodings with* `rs1`=`x0` *are RESERVED for use by future extensions.*

Description

Execute LY as though V=1, following the same pattern as HLV.W but with capability data.

Prerequisites

RVY, H

Operation

TBD

18.11.2. HSV.Y

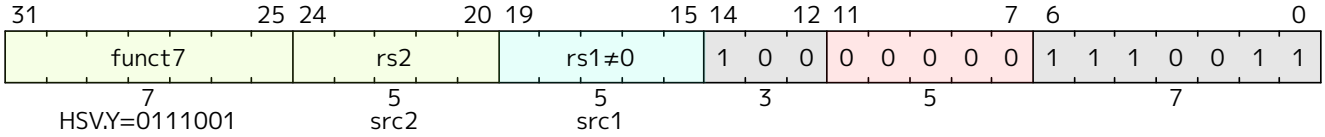
Synopsis

Hypervisor virtual-machine store capability

Mnemonic

```
hsv.y rs2, rs1
```

Encoding



Any instance of this instruction with `rs1`=`x0` *will raise an exception, as* `x0` *is defined to always hold a NULL capability. As such, the encodings with* `rs1`=`x0` *are RESERVED for use by future extensions.*

Description

Execute SY as though V=1; following the same pattern as HSV.W but with capability data.

Prerequisites for *(CHERI) Capability Mode*

RVY, H

Operation

TBD

Chapter 19. The "Smycheriot" Privileged ISA Extension

This section defines a series of small changes to the Machine-Level ISA (RVY) privileged architecture that serve to specialize it for microcontroller environments. These changes are based on, but are not exactly isomorphic to, the prior CHERIoT RV32E-based ISA.

19.1. Required Extensions

Smycheriot requires its unprivileged counterpart, Zycheriot, and builds on Machine-Level ISA (RVY).

The present specification presumes the **absence** of both

- any execution mode other than M-mode (such as VS, HS, VU, S, or U), and
- the Zyhybrid privileged extensions.

While Smycheriot is nominally compatible with Zyhybrid for Privileged Architectures, and particular instantiations may opt to permit disabling CHERI, we have not yet found a compelling reason to formally specify this composition.

19.2. CSR Reset States

The RVY base privileged ISA tends to define M-mode CSRs' reset values either as Root Executable capabilities or as otherwise unspecified values with clear capability tags. To make available Zycheriot's multiple root capability values, we redefine two CSRs' reset values:

mtidc
: The mtidc register's reset value is changed to be Zycheriot's Root Data capability value (with set capability tag), rather than the unspecified value with zero capability tag of RVY.

mscratch
: The mscratch register's reset value is changed to be Zycheriot's *sealing* capability value (with set capability tag), rather than the unspecified value with zero capability tag of RVY.

19.3. Additional CSR Legalization Requirements

All of pc, mtvec, and mepc will clear their capability tag if the capability they would come to hold after an update would be sealed or would not grant X-permission.

The Machine-Level ISA (RVY) requires that mtvec and mepc nominally reset to Root Executable capabilities, and pc must be unsealed and grant X-permission for successful execution. Neither architecture nor software expects to be able to place capabilities thereby prohibited in these CSRs.

19.4. Capability Types

Zycheriot introduces several sentry CT-field values and discusses some aspects of their interaction with `JALR` and `JAL` instructions. Smycheriot further equips these CT-field values with privileged architectural side-effects when used with `JALR`. In particular, Smycheriot uses these CT-field values to enforce structured interrupt control, by atomically capturing and changing the value of `xstatus.xIE` in `JALR`. The purpose of this is to enable calls to short per-core atomic sequences without granting the right to modify `xstatus.xIE` arbitrarily. Functions that are called with a new interrupt stance can return to the caller's stance. Non-

standard control flow transfers such as tail calls and calls to compiler outlined functions using a non-standard link register are still possible but cannot change the interrupt status. Beyond the `JALR` behaviors specified by Zycheriot, Smycheriot further requires that...

- Upon successfully retiring, a `JALR` instruction whose `rs1` holds such a capability updates the hart's `xstatus.xIE` bit as per Table 88.
- A successfully retiring `JALR` or `JAL` whose `rd` is `ra` will seal its written-back return capability based on the hart's `xstatus.xIE` bit as of the *prior* instruction's retirement. If that bit is `0`, the return capability is sealed with 4; otherwise, 5 is used.

Combining these semantics with those of Zycheriot lets us give convenient mnemonics for the particular CT-field values that these extensions use, as shown in Table 89.

Table 88. Additional `JALR` Architectural Semantics

CT-field	IRQs at Retirement (`xstatus.xIE`)
1	Unchanged (as is)
2, 4	Deferred (`0`)
3, 5	Enabled (`1`)

Table 89. Sentry type mnemonics

CT-field	Name
1	Forward interrupt-inheriting
2	Forward interrupt-deferring
3	Forward interrupt-enabling
4	Backward interrupt-deferring
5	Backward interrupt-enabling

The degree of control flow integrity provided by the rules of Zycheriot ensures that a leaf function entered via a forward interrupt-disabling sentry cannot be induced, by a malicious caller, to return to its own entry vector (indeed, to return via any forward sentry). This ensures that granting untrusted code such a sentry is not tantamount to handing it the ability to wedge the machine.

19.5. Stack High Watermark CSRs

Smycheriot introduces two new XLEN CSRs for each privilege level: `xshwm` and `xshwmb`. Both are freely read but require ASR-permission for explicit writes from software. Writes are WARL, with legal values being addresses with capability alignment (that is, multiples of YLEN bits in octet bytes). Store instructions targeting addresses between the values held in `xshwm` and `xshwmb` cause `xshwm` to be updated to the lowest targeted address (rounded down to capability alignment).

The intended software use of these CSRs is, as the section title suggests, to track the "high watermark" of a thread's C stack (that is, the lowest address written to, because "stacks grow down"). The thread context switching code should context switch these registers, having initialized `xshwmb` to the lower bound of the thread's stack capability and `xshwm` to its upper bound. Privileged stack zeroing code can be used to lower `xshwm`, so that all bytes between the addresses held in `xshwm` and `xshwmb` are known to be zero.

19.6. Capability Load Filter and The Revocation Bitmap

This section is non-normative, as it introduces no new behavior not already permitted by the RVY unprivileged architecture. It is meant to be informative detail about how CHERIoT platforms avail themselves of a particular architectural affordances therein.

As part of providing *heap temporal* safety, CHERIoT platforms may, as permitted by RVY, clear tags being transported from memory into CPU registers by a `ly` instruction. CHERIoT platforms mediate capability loads with a *capability load filter*. Software dynamic memory allocators can use this load filter to ensure that additional copies of capabilities to deallocated objects cannot be constructed (into register files and, so, into memory, too). This facilitates particularly straightforward global revocation of pointers to freed heap objects.

In implementations to date, this capability load filter is instantiated by pairing each memory block(s) in which software's dynamic allocation heap(s) are to reside with a per-memory-block *revocation bitmap*, a bit-vector wherein each bit corresponds to a capability-sized memory granule. (That is, with each capability-sized and -aligned region of the primal memory.) Such memory block(s) are said to be *revokable*. These revocation bitmaps are also exposed (as memory) to software. When `ly` transfers a valid capability (one with a set capability tag) from memory (revokable or not) into a hart's register file, it checks whether the *base* (lower bound) of that capability is within a revokable memory block and, if so, fetches the corresponding bit in the associated revocation bitmap. If that bit is set, then the capability tag stored in the hart's register file as part of this transfer is cleared.

It is up to each particular CHERIoT platform to define...

- the number, location(s), and size(s) of revokable memory blocks,
- the location(s) of their associated revocation bitmaps, and
- the mapping function between primal memory address and revocation bitmap address and bit index.

Appendix B: CHERI (RV64Y) Privileged Appendix

B.1. RVY Privileged Extensions Summary

B.1.1. H Extension (RVY added instructions)

Specifying RVY and "H" gives H Extension (RVY added instructions) functionality, which adds virtualized capability load and store instructions.

While HLVX.* only requires execute permission in the PTE, the authorizing CHERI capability must grant R-permission.

Table 90. H Extension (RVY added instructions) instruction extension

Mnemonic	RV32Y	RV64Y	Function
HLV.Y	✔	✔	Hypervisor virtual machine load capability
HSV.Y	✔	✔	Hypervisor virtual machine store capability

B.1.2. Machine level ISA for RVY

Table 91. Machine level ISA, modified instructions for RVY

Mnemonic	RV32Y	RV64Y	Function
MRET (RVY)	✔	✔	Return from machine mode handler, sets pc from mtvec (RVY) , needs ASR-permission

B.1.3. Supervisor level ISA for RVY

Table 92. Supervisor level ISA, modified instructions for RVY

Mnemonic	RV32Y	RV64Y	Function
SRET (RVY)	✔	✔	Return from supervisor mode handler, sets pc from stvec (RVY), needs ASR-permission

B.1.4. Sdext for RVY

Table 93. Sdext extension, modified instructions for RVY

Mnemonic	RV32Y	RV64Y	Function
DRET (RVY)	✔	✔	Return from debug mode, sets ddc from dddc and pc from dpc (RVY)

B.2. RVY YLEN CSR Summary

this section includes debug CSRs

Table 94. CSRs extended to YLEN

YLEN CSR	Prerequisites
dpc (RVY)	Sdext

YLEN CSR	Prerequisites
dscratch0 (RVY)	Sdext
dscratch1 (RVY)	Sdext
mtvec (RVY)	M-mode
mscratch (RVY)	M-mode
mepc (RVY)	M-mode
stvec (RVY)	S-mode
sscratch (RVY)	S-mode
sepc (RVY)	S-mode
vstvec (RVY)	H
vsscratch (RVY)	H
vsepc (RVY)	H
jvt (RVY)	Zcmt

Table 95. Action taken on writing to extended CSRs

YLEN CSR	Action on XLEN write	Action on YLEN write
dpc (RVY)	Apply Invalid address conversion. Always update the CSR with YADDRW 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
dscratch0 (RVY)	Update the CSR using YADDRW.	direct write
dscratch1 (RVY)	Update the CSR using YADDRW.	direct write
mtvec (RVY)	Apply Invalid address conversion. Always update the CSR with YADDRW 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 YADDRW even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.
mscratch (RVY)	Update the CSR using YADDRW.	direct write
mepc (RVY)	Apply Invalid address conversion. Always update the CSR with YADDRW 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
stvec (RVY)	Apply Invalid address conversion. Always update the CSR with YADDRW 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 YADDRW even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.
sscratch (RVY)	Update the CSR using YADDRW.	direct write
sepc (RVY)	Apply Invalid address conversion. Always update the CSR with YADDRW 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
vstvec (RVY)	Apply Invalid address conversion. Always update the CSR with YADDRW 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 YADDRW even if the address didn't change, including the MODE field in the address for simplicity. Vector range check * if vectored mode is programmed.

YLEN CSR	Action on XLEN write	Action on YLEN write
vsscratch (RVY)	Update the CSR using YADDRW.	direct write
vsepc (RVY)	Apply Invalid address conversion. Always update the CSR with YADDRW 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
jvt (RVY)	Apply Invalid address conversion. Always update the CSR with YADDRW even if the address didn't change.	Apply Invalid address conversion and update the CSR with the result if the address changed, direct write if address didn't change

* The vector range check is to ensure that vectored entry to the handler is within bounds of the capability written to `xtvec`. 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 YLEN-wide CSRs are written when executing CSRRWI (RVY), CSRRC (RVY), CSRRS (RVY), CSRRCI (RVY) or CSRRSI (RVY) regardless of the CHERI execution mode. When using CSRRW (RVY), YLEN bits are written when the CHERI execution mode is *(CHERI) Capability Mode* and XLEN bits are written when the mode is *(Non-CHERI) Address Mode*; therefore, writing XLEN bits with CSRRW (RVY) is only possible when Zyhybrid is implemented.

Table 96. Action taken on writing to YLEN-wide CSRs

YLEN CSR	Action on XLEN write	Action on YLEN write
dddc	Apply Invalid address conversion. Always update the CSR with YADDRW 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
ddc	Apply Invalid address conversion. Always update the CSR with YADDRW 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
drootcsel	Ignore	Ignore
drootc	Ignore	Ignore
utidc	Update the CSR using YADDRW.	direct write
stidc	Update the CSR using YADDRW.	direct write
vstidc	Update the CSR using YADDRW.	direct write
mtidc	Update the CSR using YADDRW.	direct write

XLEN bits of YLEN-wide CSRs added in Zyhybrid are written when executing CSRRWI (RVY), CSRRC (RVY), CSRRS (RVY), CSRRCI (RVY) or CSRRSI (RVY) regardless of the CHERI execution mode. YLEN bits are always written when using CSRRW (RVY) regardless of the CHERI execution mode.

Implementations which allow misa.C to be writable need to legalize `xepc` 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 97. YLEN-wide CSRs storing code pointers or data pointers

YLEN CSR	Code Pointer	Data Pointer	Unseal On Execution
dpc (RVY)	✔		✔
mtvec (RVY)	✔		
mepc (RVY)	✔		✔
stvec (RVY)	✔		
sepc (RVY)	✔		✔

YLEN CSR	Code Pointer	Data Pointer	Unseal On Execution
vstvec (RVY)	✔		
vsepc (RVY)	✔		✔
jvt (RVY)	✔		
dddc		✔	
ddc		✔	

Some CSRs store code pointers or data pointers as shown in Table 97. These are WARL CSRs that do not need to store full 64-bit addresses on RV64, and so need not be capable of holding all possible invalid addresses. Prior to writing an invalid address to these CSRs, the address must be converted to another invalid address that the CSR is capable of holding. CSRs that store fewer address bits are also subject to the invalid address check in Invalid address conversion on writing.

The tables below show all YLEN-wide CSRs.

Table 98. All YLEN-wide CSRs.

YLEN CSR	Prerequisites	Address	Permissions	Reset Value	Description
dpc (RVY)	Sdext	0x7b1	DRW	tag=0, otherwise specified by the platform	Debug Program Counter Capability
dscratch0 (RVY)	Sdext	0x7b2	DRW	tag=0, otherwise specified by the platform	Debug Scratch Capability 0
dscratch1 (RVY)	Sdext	0x7b3	DRW	tag=0, otherwise specified by the platform	Debug Scratch Capability 1
mtvec (RVY)	M-mode	0x305	MRW, ASR-permission	Nominally Root Executable	Machine Trap-Vector Base-Address Capability
mscratch (RVY)	M-mode	0x340	MRW, ASR-permission	tag=0, otherwise specified by the platform	Machine Scratch Capability
mepc (RVY)	M-mode	0x341	MRW, ASR-permission	Nominally Root Executable	Machine Exception Program Counter Capability
stvec (RVY)	S-mode	0x105	SRW, ASR-permission	Nominally Root Executable	Supervisor Trap-Vector Base-Address Capability
sscratch (RVY)	S-mode	0x140	SRW, ASR-permission	tag=0, otherwise specified by the platform	Supervisor Scratch Capability
sepc (RVY)	S-mode	0x141	SRW, ASR-permission	Nominally Root Executable	Supervisor Exception Program Counter Capability
vstvec (RVY)	H	0x205	HRW, ASR-permission	Nominally Root Executable	Virtual Supervisor Trap-Vector Base-Address Capability
vsscratch (RVY)	H	0x240	HRW, ASR-permission	tag=0, otherwise specified by the platform	Virtual Supervisor Scratch Capability
vsepc (RVY)	H	0x241	HRW, ASR-permission	Nominally Root Executable	Virtual Supervisor Exception Program Counter Capability

YLEN CSR	Prerequisites	Address	Permissions	Reset Value	Description
jvt (RVY)	Zcmt	0x017	URW	tag=0, otherwise specified by the platform	Jump Vector Table Capability
dddc	Zyhybrid, Sdext	0x7bc	DRW	tag=0, otherwise specified by the platform	Debug Default Data Capability (saved/restored on debug mode entry/exit)
ddc	Zyhybrid	0x416	URW	nominally Root Data	User Default Data Capability
drootcsel	Sdext	0x7ba	DRW	0	Multiplexing selector for drootc
drootc	Sdext	0x7bd	DRW	nominally Root Executable	Source of authority in debug mode, writes are ignored
utidc	RVY	0x480	Read: U, Write: U, ASR-permission	tag=0, otherwise specified by the platform	User thread ID
stidc	RVY	0x580	Read: S, Write: S, ASR-permission	tag=0, otherwise specified by the platform	Supervisor thread ID
vstidc	RVY	0xA80	Read: VS, Write: VS, ASR-permission	tag=0, otherwise specified by the platform	Virtual supervisor thread ID
mtidc	RVY	0x780	Read: M, Write: M, ASR-permission	tag=0, otherwise specified by the platform	Machine thread ID

Where reset values are specified in Table 98, they are typically a maximum possible value. For example, a Root Executable as specified for mtvec (RVY) is the maximum, the platform may reset this CSR to a smaller memory range, or to have fewer permissions.

Machine-Level ISA (RVY) and Supervisor-Level ISA (RVY) extend the CSRs listed in Table 99, Table 100, Table 101, Table 102 and Table 103 from the base RISC-V ISA and its extensions.

If Zyhybrid is supported then the CHERI Execution Mode determines whether YLEN or XLEN bits are returned (see CSRRW (RVY)).

Table 99. Extended debug-mode CSRs in RVY

RVY CSR	Address	Prerequisites	Permissions	Description
dpc (RVY)	0x7b1	Sdext	DRW	Debug Program Counter Capability
dscratch0 (RVY)	0x7b2	Sdext	DRW	Debug Scratch Capability 0
dscratch1 (RVY)	0x7b3	Sdext	DRW	Debug Scratch Capability 1

Table 100. Extended machine-mode CSRs in RVY

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

`mconfigptr` *is not extended, despite representing an address, as it is solely for use by low-level system software and is not interpreted by hardware. Such software can be expected to hold*

suitable Root capabilities from which it can derive a capability to the address in this register.

Table 101. Extended supervisor-mode CSRs in RVY

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

Table 102. Extended virtual supervisor-mode CSRs in RVY

RVY CSR	Address	Prerequisites	Permissions	Description
vstvec (RVY)	0x205	H	HRW, ASR-permission	Virtual Supervisor Trap-Vector Base-Address Capability
vsscratch (RVY)	0x240	H	HRW, ASR-permission	Virtual Supervisor Scratch Capability
vsepc (RVY)	0x241	H	HRW, ASR-permission	Virtual Supervisor Exception Program Counter Capability

Table 103. Extended user-mode CSRs in RVY

RVY CSR	Address	Prerequisites	Permissions	Description
jvt (RVY)	0x017	Zcmt	URW	Jump Vector Table Capability

B.3. CHERI System Implications

Unclear if this chapter will appear in the priv spec. May just be in the standalone spec.

CHERI processors need memory systems which support the capability 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 capability tag along with the data. This is typically done using user-defined bits in the protocol.
    a. These busses will read capability tags from memory (if capability tags are present in the target memory) and return them to the requestor.
    b. These busses will write the capability tag to memory as an extension of the data write.
2. Non-capability tag-aware busses, i.e., current non-CHERI-aware busses.
    a. Reads of tagged memory will not read the capability tag.
    b. Writes to tagged memory will set the capability tag to zero of any YLEN-aligned YLEN-wide memory location where any byte matches the memory write.

The fundamental rule for any CHERI system is that the capability tag and data are always accessed atomically. For every naturally aligned YLEN-wide memory location, it must never be possible to:

1. Update any data bytes without also writing the capability tag
    a. This implies setting the capability tag to zero if a non-CHERI aware bus master overwrites a

capability in memory

2. Read a capability tag value with mismatched (stale or newer) data
3. Set the capability tag without also writing the data.

Clearing capability tags in memory does not necessarily require updating the associated data.

B.3.1. Small CHERI system example

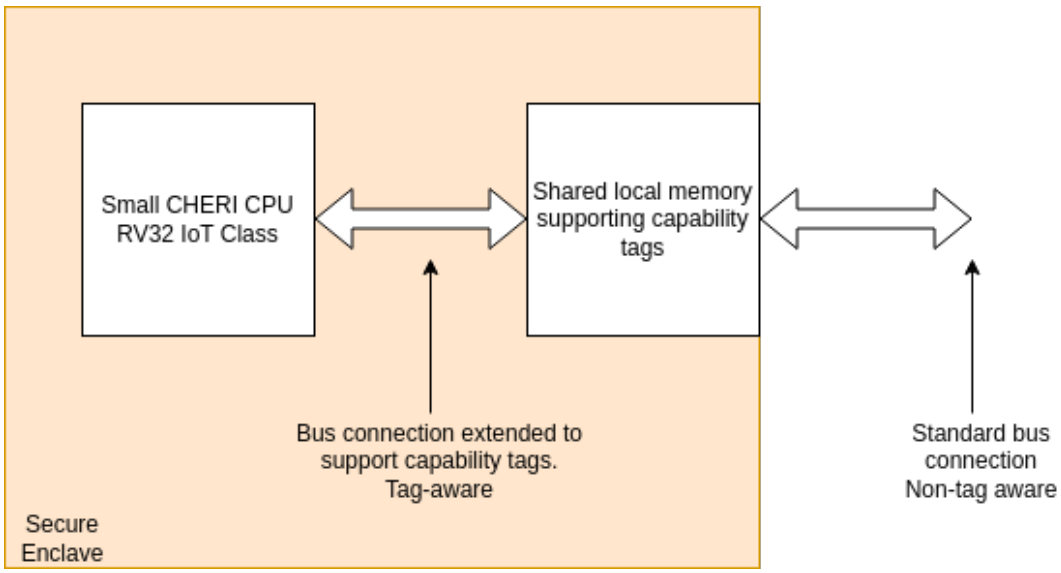


Figure 50. Example small CHERI system with local capability tag storage

This example shows a minimum-sized system where only the local memory is extended to support capability tags. The capability tag-aware region is highlighted. All capability 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 capability tag-aware.

Therefore the connection between CPU and memory is tag-aware, and the connection to the system is not capability tag-aware.

All writes from the system port to the memory must clear any memory capability tags to follow the rules from above.

B.3.2. Large CHERI system example

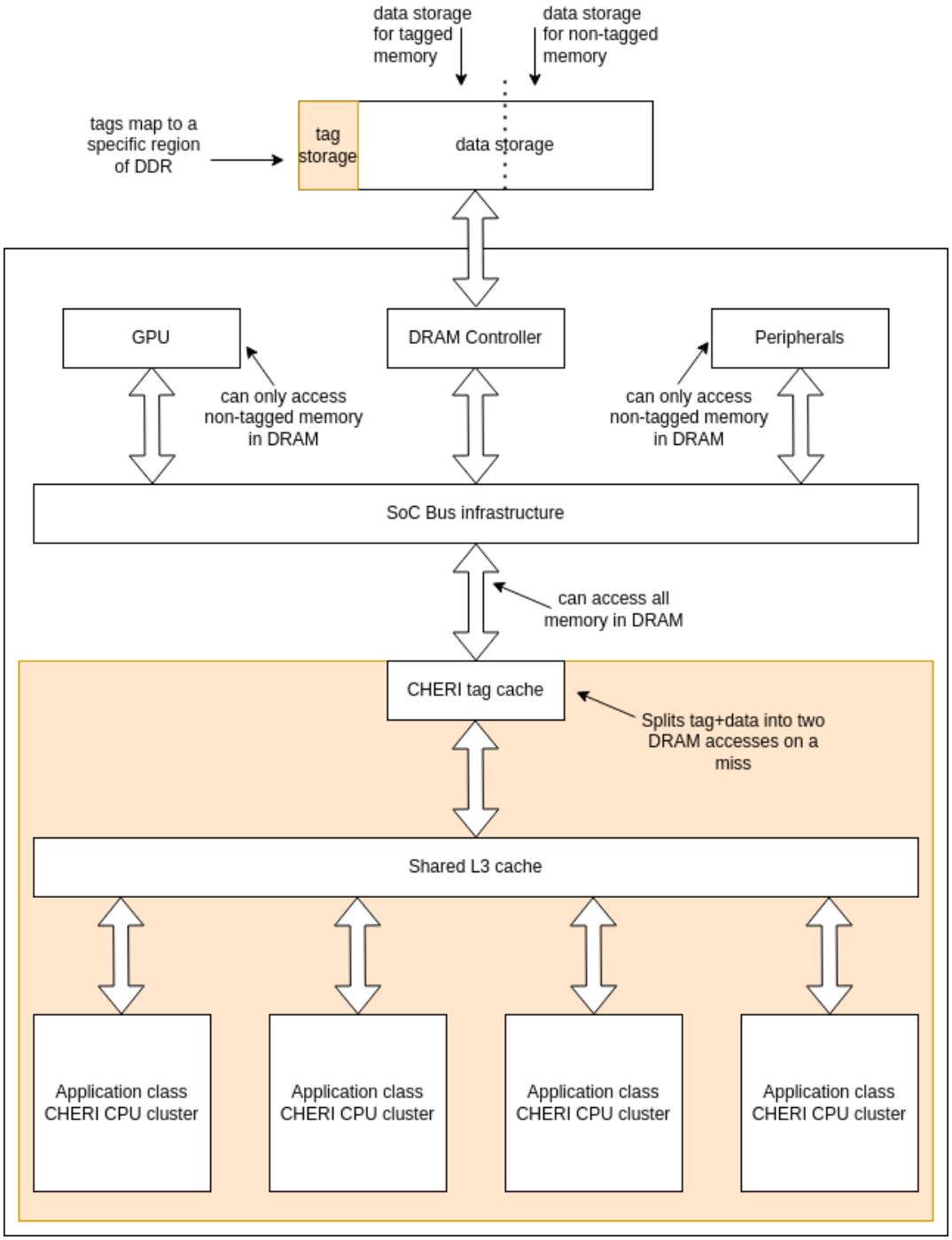


Figure 51. Example large CHERI system with capability tag cache

In the case of a large CHERI SoC with caches, all the cached memory visible to the CHERI CPUs must support capability tags. All memory is backed up by DRAM, and standard DRAM does not offer the extra bit required for CHERI capability tag storage and so a typical system will have a capability tag cache IP.

A region of DRAM is typically reserved for CHERI capability tag storage.

The capability tag cache sits on the boundary of the capability tag-aware and non-tag-aware memory domains, and it provides the bridge between the two. It stores capability 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 capability 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 capability tag.

The key property of the capability tag cache is to preserve the atomic access of data and capability tags in

the memory system so that all CPUs have a consistent view of capability tags and data.

The region of DRAM reserved for capability tag storage must be accessible only by the capability tag cache, therefore no bus initiators should be able to write to the DRAM without the transactions passing through the capability tag cache.

Therefore the GPUs and peripherals cannot write to the capability tag storage in the DRAM or the capability tag supporting 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 capability tag supporting memory data storage region of the DRAM, if required.

It would be possible to allow a DMA to access the capability 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 capability tags, and so the root-of-trust will need to authenticate and encrypt the transfer, with anti-rollback protection.

For further information on the capability tag cache see (*Efficient Tagged Memory*, 2017).

B.3.3. Large CHERI pure-capability system example

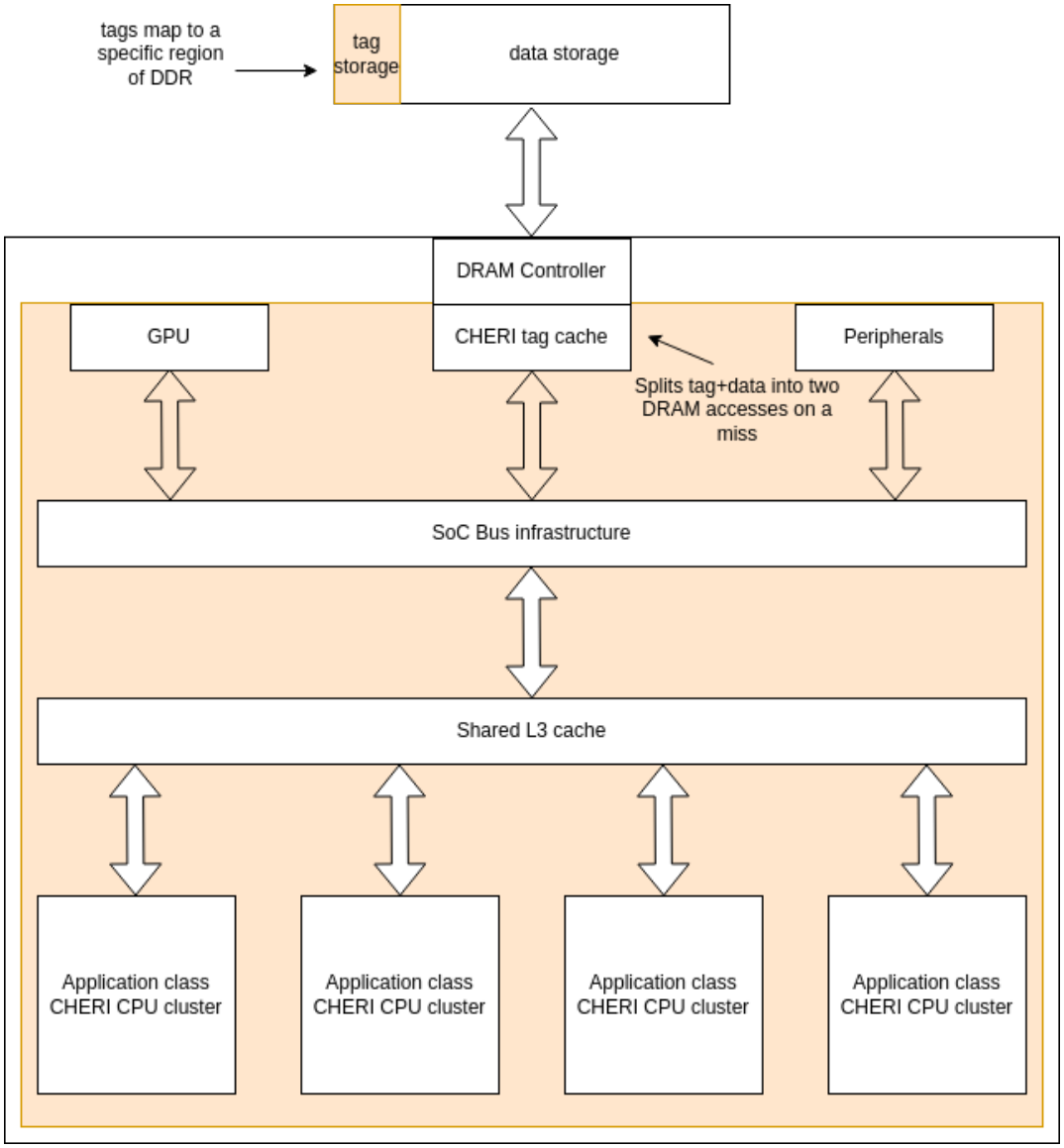


Figure 52. Example large CHERI system with only tag-aware bus masters

In this example every DRAM access passes through the capability tag cache, and so *all* bus masters are capability tag-aware and can access the capability tags associated with memory if permitted by the network-on-chip.

The system topology is simpler than in Figure 51.

There is likely to be a performance difference between the two systems. The main motivation for Figure 51 is to avoid the GPU DRAM traffic needing to look-up every capability tag in the capability tag cache, potentially adding overhead to every transaction.

B.4. Placeholder references to privileged spec

This chapter only exists for the standalone document to allow references to resolve.

Control and Status Registers (CSRs) overview

See Chapter *Control and Status Registers (CSRs)* in (RISC-V, 2023).

Machine Status Registers (mstatus and mstatush)

Base ISA Control in `mstatus` Register

Endianness Control in `mstatus` and `mstatush` Registers

See `mtatus` in Chapter *Machine-Level ISA, Version 1.13* in (RISC-V, 2023).

Machine Scratch Register (mscratch)

See `mscratch` in Chapter *Machine-Level ISA, Version 1.13* in (RISC-V, 2023).

Figure 53. Machine-mode scratch register

Machine Cause (`mcause`) Register

Synchronous exception priority in decreasing priority order

Machine cause (`mcause`) register values after trap

See `mcause` in Chapter *Machine-Level ISA, Version 1.13* in (RISC-V, 2023).

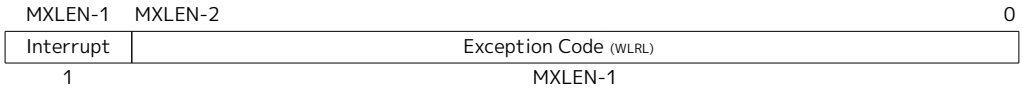


Figure 54. Machine Cause (`mcause`) register.

Machine Trap-Vector Base-Address (`mtvec`) Register

See `mtvec` in Chapter *Machine-Level ISA, Version 1.13* in (RISC-V, 2023).

Figure 55. Machine-mode trap-vector base-address register

Machine Exception Program Counter (mepc)

See `mepc` in Chapter *Machine-Level ISA, Version 1.13* in (RISC-V, 2023).

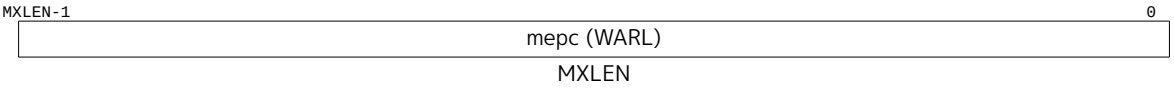


Figure 56. Machine exception program counter register

Machine Trap Delegation Register (medeleg)

See `medeleg` in Chapter *Machine-Level ISA, Version 1.13* in (RISC-V, 2023).

Machine Trap Value Register (mtval)

See `mtval` in Chapter *Machine-Level ISA, Version 1.13* in (RISC-V, 2023).

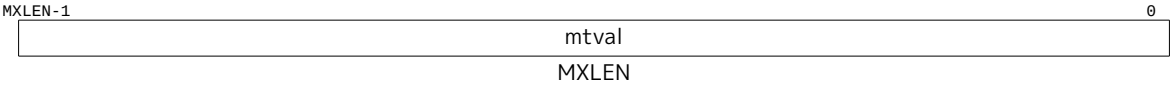


Figure 57. Machine trap value register

Machine ISA (`misa`) Register

See `misa` in Chapter *Machine-Level ISA, Version 1.13* in (RISC-V, 2023). RVY sets the Y bit to be 1, and I or E are set to show how many X registers are present.

If Zyhybrid is implemented, then Y is writable.

Machine Environment Configuration (`menvcfg`) Register

See `menvcfg` in Chapter *Machine-Level ISA, Version 1.13* in (RISC-V, 2023). Zyhybrid adds a new Y bit.

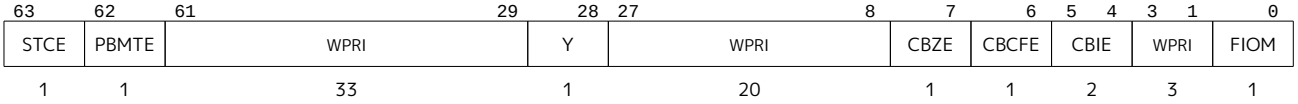


Figure 58. Machine environment configuration register (menvcfg)

Trap-Return Instructions

See Trap-Return Instructions in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

Supervisor Trap Vector Base Address (`stvec`) Register

See `stvec` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

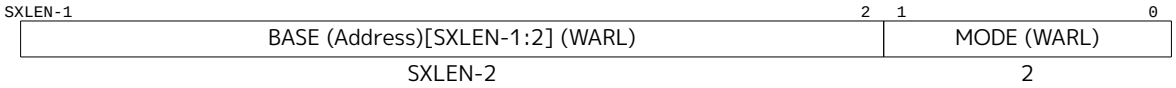


Figure 59. Supervisor trap vector base address (`stvec`) register.

Supervisor Scratch (`sscratch`) Register

See `sscratch` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

Figure 60. Supervisor-mode scratch register

Supervisor Exception Program Counter (`sepc`) Register

See `sepc` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

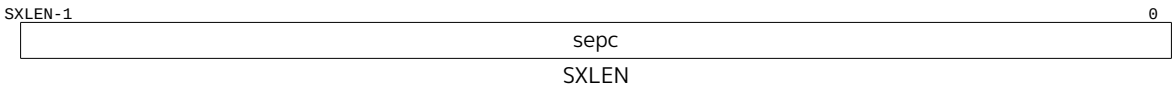


Figure 61. Supervisor exception program counter register

Supervisor Trap Value (`stval`) Register

See `stval` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

Figure 62. Supervisor trap value register

Supervisor Cause (`scause`) Register

.Supervisor cause (`scause`) register values after trap

See `scause` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

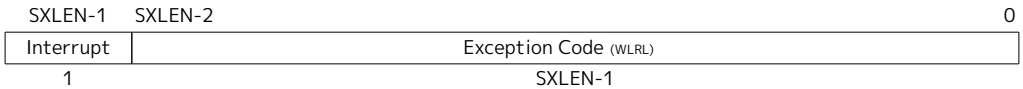


Figure 63. Supervisor Cause (`scause`) register.

Supervisor Environment Configuration (`senvcfg`) Register

See `senvcfg` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023). Zyhybrid adds a new Y bit.

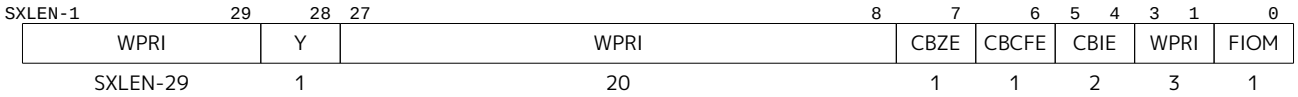


Figure 64. Supervisor environment configuration register (senvcfg)

Supervisor Status (`sstatus`) Register

See `sstatus` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

Page-Based 39-bit Virtual-Memory System (Sv39)

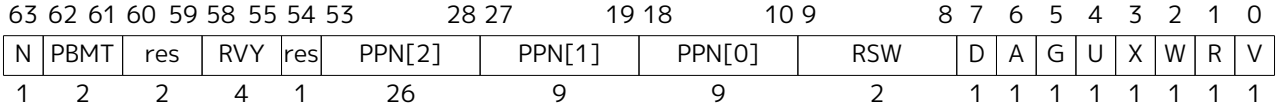


Figure 65. Sv39 page table entry.

Bibliography

Efficient Tagged Memory. (2017). www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201711-iccd2017-efficient-tags.pdf

Filardo, N. W., Gutstein, B. F., Woodruff, J., Clarke, J., Rugg, P., Davis, B., Johnston, M., Norton, R., Chisnall, D., Moore, S. W., Neumann, P. G., & Watson, R. N. M. (2024). *Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety*. doi.org/10.1145/3620665.3640416

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 Privileged Specification*. github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-056b6ff-2023-10-02/priv-isa-asciidoc.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