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

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>
- Michael Roe <mr101@cam.ac.uk>
- Peter Rugg <peter.rugg@cl.cam.ac.uk>
- Peter Sewell <peter.sewell@cl.cam.ac.uk>
- Carl Shaw <carl.shaw@codasip.com>
- Ricki Tura <ricki.tura@codasip.com>
- Robert N. M. Watson <robert.watson@cl.cam.ac.uk>
- Toby Wenman <toby.wenman@codasip.com>

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

Chapter 1. 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 adoptability from current ISAs and software stacks, low performance overhead for memory protection, significant performance improvements for software compartmentalization, formal grounding, and programmer-friendly underpinnings. It has been designed to provide strong, non-probabilistic protection rather than depending on short random numbers or truncated cryptographic hashes that can be leaked and reinjected, or that could be brute forced.

1.2. CHERI Extensions to RISC-V

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

Compatibility with RVA, RVB and RVM profiles will be listed in separate RVY based profiles (RVYA, RVYB, RVYM). These will list the compatible extensions.

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

ℹ️ *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.

RVY is defined as the base ISA that all CHERI RISC-V implementations must support. Zyhybrid and Svucrg are examples of optional extensions in addition to RVY.

In some cases adding an existing extension, such as H, causes additional instructions to be added.

In other cases adding an existing extension, such as C, causes additional instructions to be added and existing instructions to have modified behavior.

1.2.1. Stable Extensions and Specifications

Table 1. Unprivileged stable RVY extensions

Extension or Specification	Description
RVY	Base ISA

Extension or Specification	Description
Zydefaultcap	Default capability encoding
Zys	Ambient capability sealing instruction
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)	Zca instructions modified by RVY
Zicsr (RVY modified behavior)	Zicsr instructions modified by RVY
Zicbom (RVY modified behavior)	Zicbom instructions modified by RVY
Zicboz (RVY modified behavior)	Zicboz instructions modified by RVY
Zicbop (RVY modified behavior)	Zicbop instructions modified by RVY

Table 3. Unprivileged stable extension used by CHERI software

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

Table 4. Privileged stable extensions and specifications

Extension or Specification	Description
Machine-Level ISA (RVY)	Machine ISA
Supervisor-Level ISA (RVY)	Supervisor ISA
Zyhybrid for Privileged Architectures	Hybrid extension for RVI compatibility
Supervisor-Level ISA for Virtual Memory (RV64Y)	Virtual Memory
Sdtrig (RVY)	Debug triggers
Svucrg[1]	MMU-based acceleration of capability revocation for heap temporal safety

[1] Svucrglct is available for improved software revocation performance if Svucrg is implemented.

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
Vector "V" (RVY)	Vector
Zcmt (RV32Y)	Table Jump for RV32Y
Zcmp (RV32Y)	Push/pop and double move for RV32Y
Xycheriot	CHERIoT capability encoding
Zyseal	Capability-mediated capability (un)sealing instructions

Table 7. Privileged experimental extensions and specifications

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

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 describes the RV32Y and RV64Y base capability instruction sets, that extend the RV32I and RV64I (or RV32E/RV64E) integer instruction sets with CHERI. If the hart supports RV32E/RV64E instead of RV32I/RV64I, RVY the restrictions from RV32E and RV64E apply and 16 general purpose registers instead of 32 — but all other RVY changes are identical so the remainder of this chapter only refers to extending RV32I/RV64I behavior.

CHERI enhances the base ISA to add hardware memory access control. It has an additional memory access mechanism that protects *references to code and data* (pointers), rather than the *location of code and data* (integer addresses). This mechanism is implemented by providing a new primitive, called a **capability**, that software components can use to implement strongly protected pointers within an address space. Capabilities are unforgeable and delegatable tokens of authority that grant software the ability to perform a specific set of operations. In CHERI, integer-based pointers are replaced by capabilities to provide memory access control.

2.1. 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 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 A.1.2, "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.

2.2. 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 one-bit capability tag which is defined below.

RVY specify the fields which the capability format supports, and how those fields behave for XLEN=32 and XLEN=64.

The exact encoding format of the capability is described in Zydefaultcap.

> **i** *Future extensions may redefine the capability format providing they follow the rules defined by the RVY base ISA.*
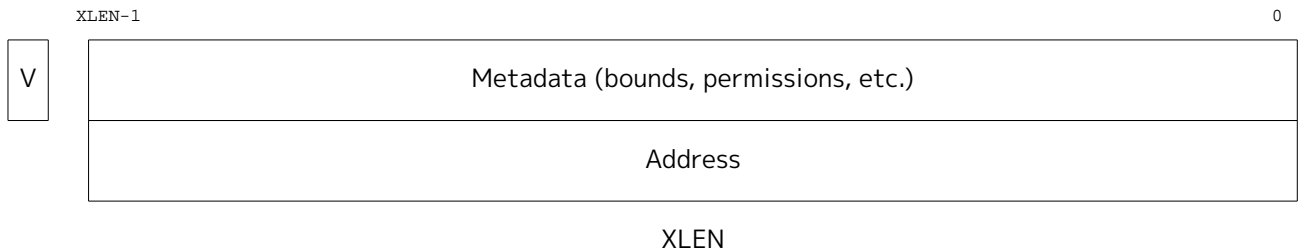
XLEN

Figure 1. CHERI Capability structure

2.2.1. 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.

> **i** *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.*

2.2.2. 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 to one 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, for capability manipulation in registers:

- Any instruction that wrote the capability to a register had at least one capability tag set in its input operands.
  - This is the *provenance* check.
- Any instruction that wrote the capability to a register requested a legal operation that does not increase bounds or permissions, and set the capability tag on the output.
  - This is the *monotonicity* check.
- Any instruction that wrote the capability to a register detected corrupted values.

- This is the *integrity* check.

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.

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.

> *When the capability tag is zero, the register or memory location may be used to store non-capability data.*

2.2.3. 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.

2.2.4. 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 which is permitted to set the capability tag to one (see C-permission), and that the stored capability data has its capability tag set.

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

2.2.5. 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.*
  - The extra bit is required to allow the bounds to include the top byte of memory.
- 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.

> *Inclusive top, with XLEN bits, was considered but rejected in favor of the exclusive top.*

> *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.

The RVY bounds encoding format is defined in Section A.1.1.4.

> *Future extensions may use alternative bounds encoding formats. Software can query the capability format by reading the* uycfg *CSR.*

Software can query the capability bounds:

- 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* can be calculated using a saturating addition of the YBASER and YLENR results.

> *A future extension may add an instruction to directly read the top if needed for performance.*

2.2.6. 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*).

Bounds can be programmed using the YBNDSW, YBNDSWI and YBNDSRW instructions, which set the current address to be the *base* bound and the *length* to be the operand (`rs2` or `imm`) value. The granularity constraints means 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 A.1.1.4.

2.2.7. Representability and Updating the Address

Because the CHERI Concentrate (Woodruff et al., 2019) encoding scheme for memory bounds shares the upper bits of the address with the bounds, not all out-of-bounds pointers can be represented.

> *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.*

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

> *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 come 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 Zydefaultcap (e.g., for accelerators or specific micro-controllers, see uycfg) 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.*

The bounds and representable region (or space) for Zydefaultcap are illustrated in Figure 2. E, MW and R in the figure are all introduced in Section A.1.1.4.2 along with the bounds decoding.
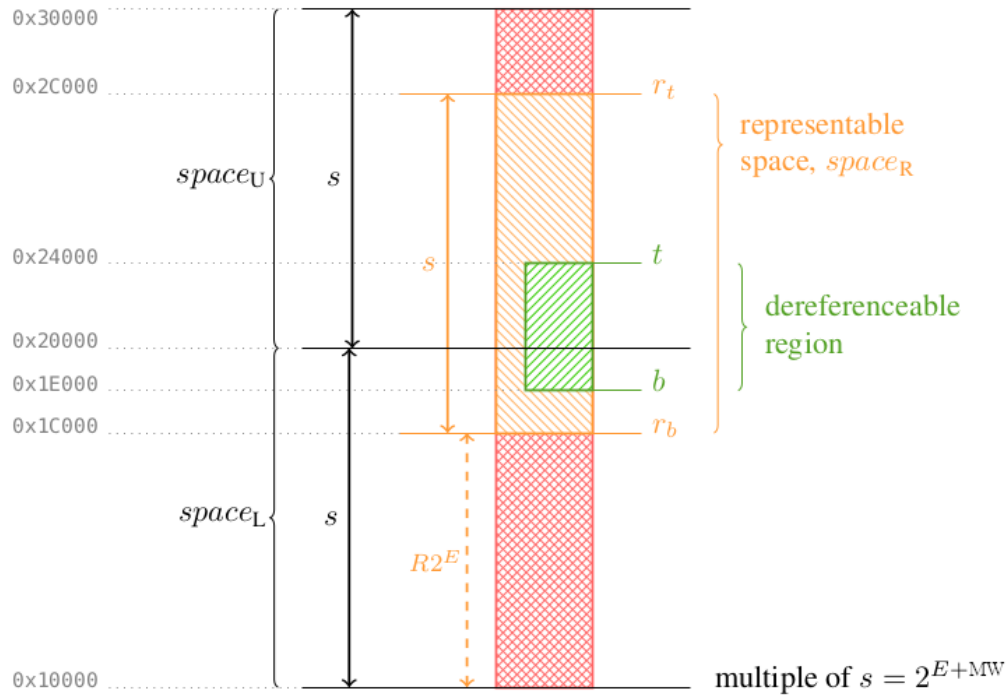
Figure 2. Memory address bounds encoded within a capability

2.2.8. Capability Type (CT)

This metadata value indicates the type of the capability. The sole type defined in the RVY base ISA is `0`. The type determines which operations the capability authorizes; extensions to RVY will define additional types and give additional semantics for capabilities with such types.

Which capability types a given CHERI platform supports is a function of the extensions and capability encoding format in use. The capability encoding additionally specifies a mapping between some bits within the capability format (usually described as "the CT field") and the space of 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.

Unsealed capabilities

When `CT=0`, the capability authorizes access to a region of memory as defined by the permissions and bounds.

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 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. (Zydefaultcap does not avail itself of this option, but, for example, the CHERIoT 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 Zys is available on a given platform, the type with which it seals capabilities is considered ambiently available. (With the capabilities of Zydefaultcap, 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. JALR (RVY) may unseal capabilities of particular, encoding-specified types before installing them to the program counter. Capabilities sealed with such a type are are dubbed "sentries" (a portmanteau of "sealed entries"). JALR (RVY) may also *seal* the return pointers it generates with encoding-specified types.

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

<mark>End new since last ARC review</mark>

> 🛈 *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.*

2.2.9. 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 currently defined in RVY are listed below.

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

> 🛈 *The encoding of permissions varies with MXLEN and is described in AP-field encoding.*

Table 8. AP-field summary

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

Permission	Type	Comment
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. In RVY, the only affected CSR is the unprivileged utidc CSR, which requires ASR-permission for writing, but not for reading. ASR permission is used for additional permission checks by some instructions from other extensions such as Zicbom.

ASR-permission permission checks always use the permission in the pc.

> 🛈 *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.*

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 (such as Zyhybrid and Zylevels1) may also introduce new dependency constraints. Capability *encodings* may impose additional constraints. to reduce the number of bits necessary to represent permissions.

Table 9. 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

2.2.10. 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.*

2.2.11. Special Capabilities

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^{MXLEN} 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 short-hand 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.

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.

2.3. Extended State

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

2.3.1. 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.

2*XLEN (YLEN)		XLEN	0
x0		x0	
x1		x1	
Same layout for x2-x30			
x31		x31	
pc		pc	
V	Metadata	Address	
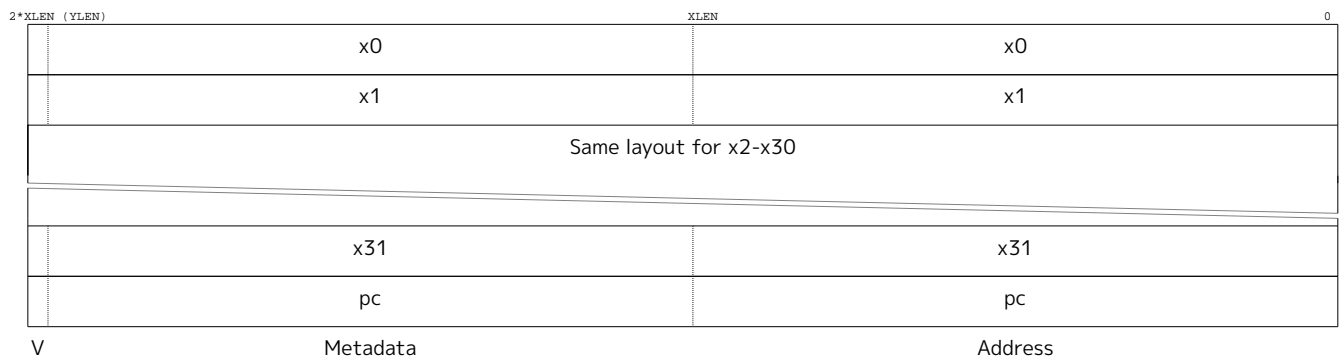
Figure 3. Extended registers in RVY

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

2.3.2. 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 execute permission
- All bytes of the instruction must be in bounds

- All integrity checks 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 storing in all copies of the pc in the implementation.*
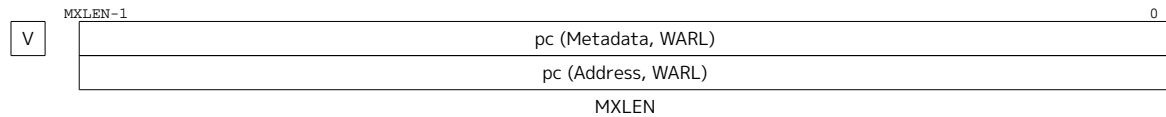
Figure 4. Program Counter Capability

2.3.3. Added CSRs

RV32Y and RV64Y add the YLEN-bit CSR shown in Table 10 and the XLEN-bit CSR shown in Table 11.

Table 10. Unprivileged capability CSRs added in RVY

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

Table 11. Unprivileged integer CSRs added in RVY

XLEN CSR	Permissions	Description
uycfg	RO; see section for details	Capability encoding version

User Thread Identifier Capability (utidc)

The utidc register is used to identify the current software thread in user mode. 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.*
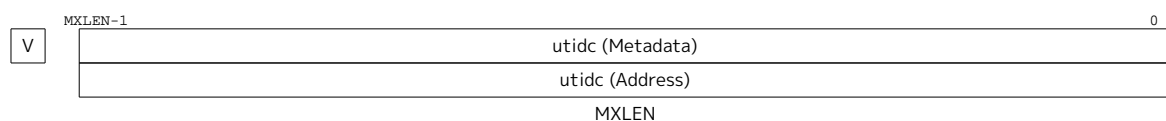
Figure 5. User thread identifier capability register

> ==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 piece of 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.

CHERI Capability Encoding (uycfg)

The uycfg register is a read-only XLEN CSR used to identify which CHERI capability encoding is used by the platform. The capability encoding both determines the in-memory representation of a CHERI capability and entails a set of extensions present atop RVY.

> **i** *Alternative capability encoding specifications, with, for example, new features and/or different bounds granularity, are possible and would not change how CHERI integrates with the RISC-V ISA, provided that said encodings still provide the primitives detailed in Section 2.2.*

> **i** *This CSR is read-only, but future extensions may provide a mechanism to modify the capability encoding currently in use.*

XLEN-1	9 8	0
Reserved	Features	Base Encoding
MXLEN-16	8	8

Figure 6. CHERI capability encoding CSR (uycfg) format

The following values for uycfg are defined:

Base	Features	Encoding
0x0	-	Reserved (unspecified capability encoding)
0x1	0x0	Zydefaultcap capabilities, without Zylevels1 features
	0x1	Zydefaultcap capabilities, with Zylevels1 features
0x2	0x0	Reserved for CHERIoT's (RV32-only) encoding

Future extensions that change capability encoding must define new values. Extensions that define encodings derived from Zydefaultcap and are generally compatible with this encoding (e.g. they only assign meaning to previously reserved bits) must preserve the Base Encoding field (the bottom 8 bits) as

`0x01`.

> We suggest that only low-level, system software query this value directly. Operating systems and/or runtime environments should provide a mechanism for software to determine if the platform's capability encoding is backwards-compatible with a given, presumed encoding. Among other things, such a facility should be used as part of the platform executable loader to reject executables that expect unknown or known-incompatible CHERI features or capability encodings.

2.3.4. Extended CSRs

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

RVY has three classes of CSR

- 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:
      i. 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

Instruction	Read Width	Write Width
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.

2.4. 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.

2.5. Added Instructions

RVY adds new instructions to operate on capabilities.

2.5.1. 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
ADDIY	Increment capability address by immediate, represented exactly check
ADDY	Increment capability address by register, represented exactly check
YADDRW	Replace capability address, represented exactly check

ADDIY

See ADDY.

ADDY

Synopsis

Capability pointer increment

Mnemonic

```
addy rd, rs1, rs2
addiy 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

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2!=x0	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
ADDY=0000110	increment	src	ADDY=000	dest	OP=0110011	

31	20 19	15 14	12 11	7 6	0
imm	rs1	funct3	rd	opcode	
12	5	3	5	7	
imm	src	ADDIY=010	dest	OP-IMM-32=0011011	
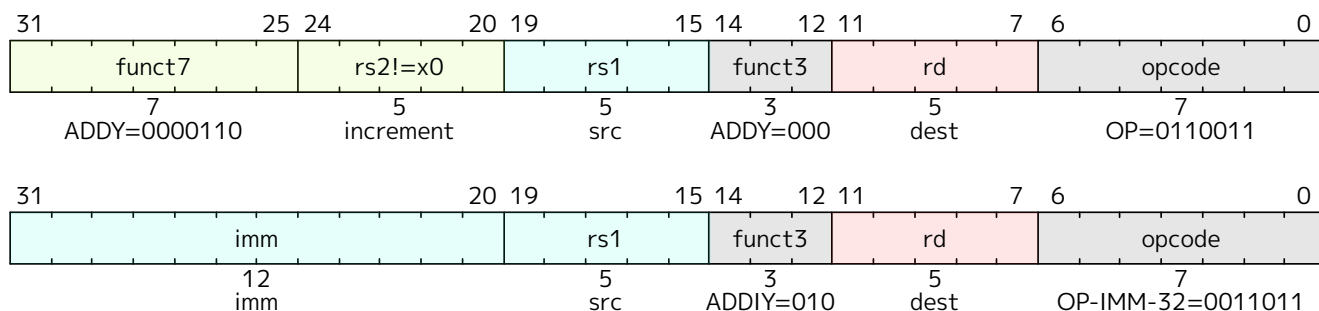
> 🛈 *ADDY with `rs2=x0` is decoded as YMV instead, the key difference being that capabilities cannot have their capability tag cleared by YMV.*

Description

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

For ADDY, increment `rd.address` by the value in `rs2[XLEN-1:0]` .
For ADDIY, 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 ADDY

```
let cs1_val = C(cs1);
let rs2_val = X(rs2);

let newCap = incCapAddrChecked(cs1_val, rs2_val);
```

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

Operation for ADDIY

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

    let newCap = incCapAddrChecked(cs1_val, immBits);

    C(cd) = newCap;
    RETIRE_SUCCESS
```

YADDRW

Synopsis

Write capability address

Mnemonic

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

Encoding

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
YADDRW=0000110	address	src	YADDRW=001	dest	OP=0110011	
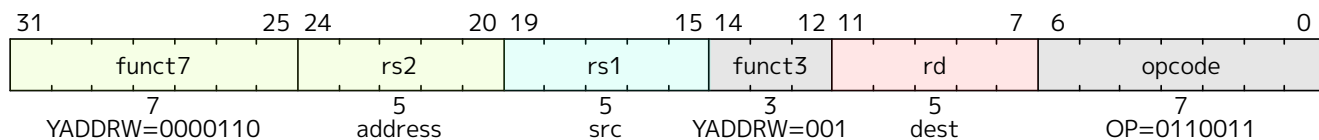
Description

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
```

2.5.2. 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	AND capability permissions (expand to 1-bit per permission before ANDing)
YMV	Move capability register
YHIW[1]	Set metadata and clear capability tag
YBNDSWI	Set immediate bounds on capability with rounding, clear capability tag if rounding is required
YBNDSW	Set register bounds on capability with rounding, clear capability tag if rounding is required
YBNDSRW	Set bounds on capability with rounding up as required
YBLD	Set rd to rs2 with its capability tag set after checking that rs2 is a subset of rs1
YSUNSEAL	Set rd to rs2 with a zero CT-field if rs2 is a sealed subset of rs1

[1] YHIW is a pseudoinstruction for PACKY

YPERMC

Synopsis

Clear capability permissions

Mnemonics

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

Encoding

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
YPERMC=0000110	mask	src	YPERMC=010	dest	OP=0110011	
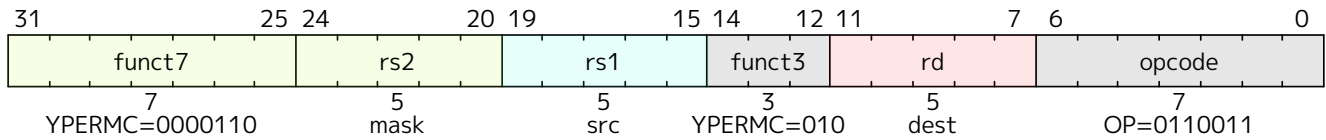
Description

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 7.

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.

    > **i** *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 iterating over the rules in Section 2.2.9.1, as well as any rules added by extensions or the capability encoding, to a fixed point (that is, a set of permissions not further reduced by any rule); encodings may, however, specify other encoding procedures.

    > **i** *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.

XLEN−1	24	23	19	18	17	16	15	SDPLEN+5	6	5	4	2	1	0
Reserved 0		Reserved 1		R	X	ASR	Reserved 1		SDP		C	Reserved 1		LM	W
XLEN-24		5		1	1	1	10-SDPLEN		SDPLEN		1	3		1	1
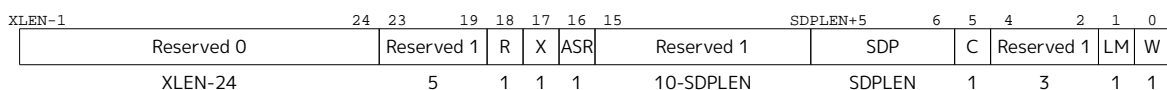
Figure 7. Capability permissions bit field

> **i** *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

```
let cs1_val = C(cs1);
let rs2_val = X(rs2);

let cond = capIsSealed(cs1_val) | not(capReservedValid(cs1_val));
let inCap = clearTagIf(cs1_val, cond);

let old_perms = packPerms(getArchPermsLegalized(inCap),
inCap.sd_perms).bits;

let new_perms = old_perms & rs2_val;

let (new_arch_perms, new_sd_perms) = unpackPerms(struct {bits =
new_perms});
let newCap = { setArchPerms(inCap, new_arch_perms) with sd_perms =
new_sd_perms };

C(cd) = newCap;
RETIRE_SUCCESS
```

YMV

Synopsis

Capability register copy

Mnemonic

`ymv rd, rs1`

Suggested assembly syntax

`ymv rd, rs1`

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		zero		rs1		funct3		rd		opcode	
7		5		5		3		5		7	
ADDY=0000110		rs2=x0		src		ADDY=000		dest		OP=0110011	
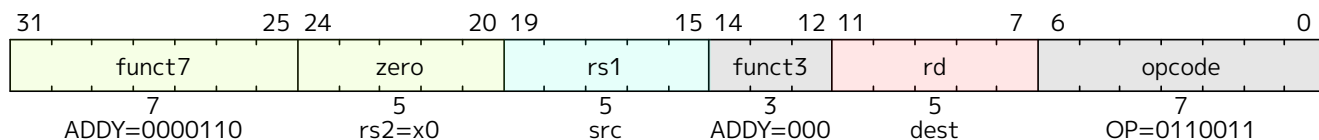
> **ℹ** *YMV is encoded as ADDY with* `rs2=x0`.

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
```

PACKY

Synopsis

Pack Y register

Mnemonic

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

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode	

7	5	5	3	5	7
PACKY=0000100			PACKY=011		OP=0110011
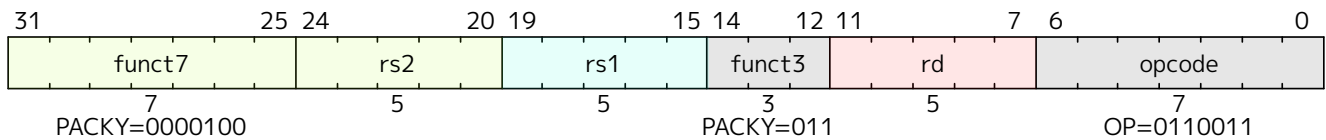
Description

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

Included in

RVY

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.

> **ℹ** *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
```

YBNDSWI

See YBNDSW.

YBNDSW

Synopsis

Write capability bounds

Mnemonics

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

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode	

7	5	5	3	5	7
YBNDSW=0000111	src2	src1	YBNDSW=000	dest	OP=0110011

31	30	29	20	19	15	14	12	11	7	6	0
funct2		imm		rs1=rd		funct3		rd=rs1		opcode	

3	10	5	3	5	7
00	imm	src	YBNDSWI=011	dest	OP-IMM-32=0011011
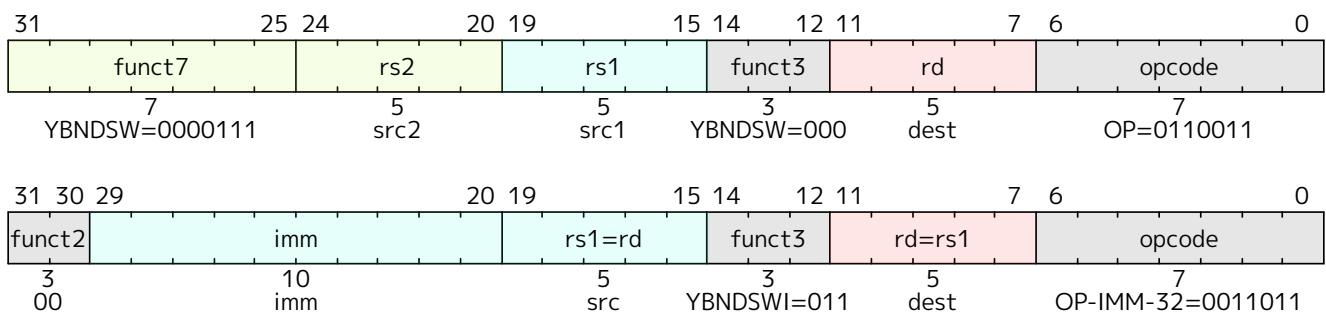
> ℹ️ *99.5% of uses of unrestricted allocation of rs1 and rd for YBNDSWI have rs1=rd, and so the cases where they do not match are reserved.*

==ARC note - YBNDSWI has been re-encoded with a longer immediate==

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`

> ℹ️ *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 <<<

YBNDSRW

Synopsis

Write capability bounds, rounding up if required

Mnemonic

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

Encoding
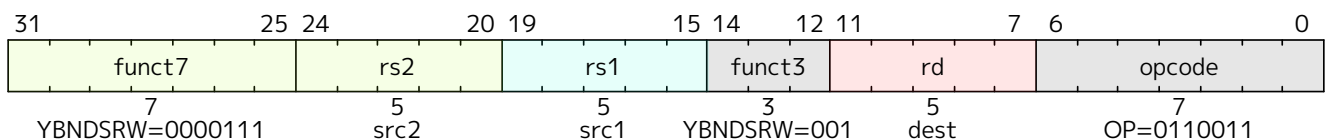
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
YBNDSRW=0000111	src2	src1	YBNDSRW=001	dest	OP=0110011

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 the requested bounds cannot be encoded exactly.

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
```

YBLD

Synopsis

Build capability

Mnemonic

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

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode	

7	5	5	3	5	7
YBLD=0000110	src2	src1	YBLD=101	dest	OP=0110011
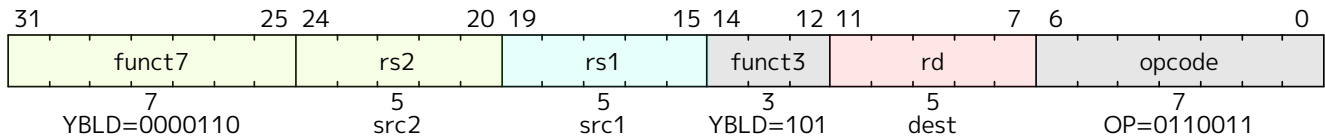
Description

Copy `rs2` to `rd`.

Set `rd.tag=1` if:

1. `rs1.tag` is set, 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`

==Begin new since last ARC review==

If `rd.ct` (that is, its CT-field) is neither 0 nor an ambient type, then set `rd.ct` to 0. That is, YBLD will construct a sealed capability only if its type is ambiently available.

==End new since last ARC review==

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

> ℹ️ *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

RVY

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
```

YSUNSEAL

Synopsis

Unseal by superset reconstruction

Mnemonic

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

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode	

7	5	5	3	5	7
YSUNSEAL=0000111	src2	src1	YSUNSEAL=010	dest	OP=0110011
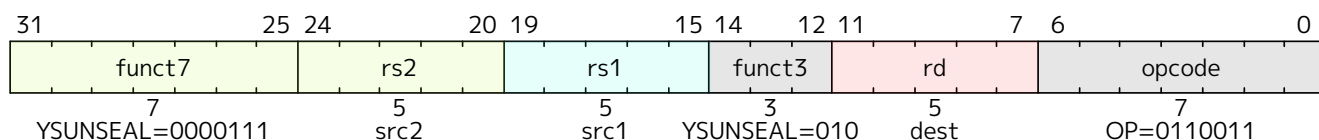
Description

Copy `rs2` to `rd`.

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

Set `rd.tag=1` if:

1. `rs1.tag` is set, 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`

> ℹ️ 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.

> ℹ️ 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

End new since last ARC review

2.5.3. 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	Get capability base
YLENR	Get capability length

YBASER

Synopsis

Read capability base address

Mnemonic

```
ybaser rd, rs1
```

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		funct5		rs1		funct3		rd		opcode	
7		5		5		3		5		7	
YBASER=0001000		YBASER=00101		src		YBASER=000		dest		OP=0110011	
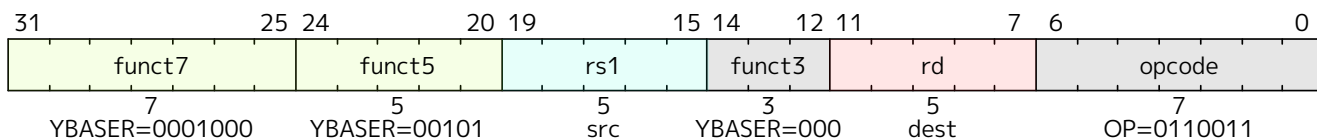
Description

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
```

YLENR

Synopsis

Read capability length

Mnemonic

```
ylenr rd, rs1
```

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		funct5		rs1		funct3		rd		opcode	

7	5	5	3	5	7
YLENR=0001000	YLENR=00110	src	YLENR=000	dest	OP=0110011
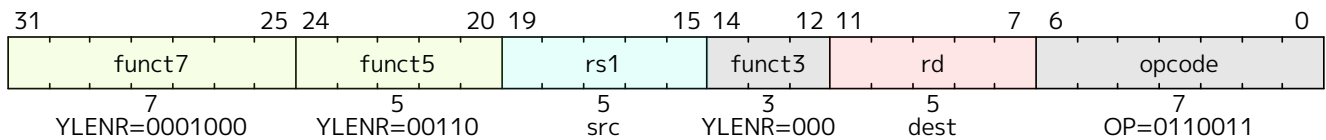
Description

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
```

2.5.4. 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	Get capability tag
YPERMR	Get capability architectural and software permissions
YTYPER	Get capability type
YHIR[1]	Get capability metadata

[1] YHIR is a pseudoinstruction for SRLIY

YTAGR

Synopsis

Read capability tag

Mnemonic

```
ytagr rd, rs1
```

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		funct5		rs1		funct3		rd		opcode	

7	5	5	3	5	7
YTAGR=0001000	YTAGR=00000	src	YTAGR=000	dest	OP=0110011
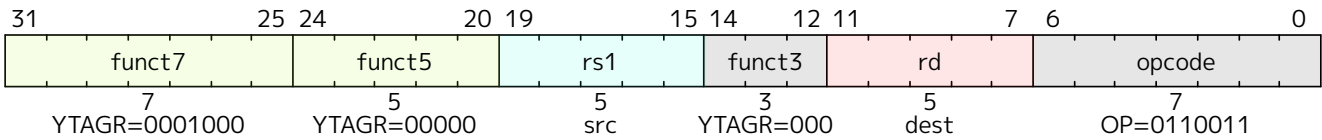
Description

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
```

YPERMR

Synopsis

Read capability permissions

Mnemonic

```
ypermr rd, rs1
```

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		funct5		rs1		funct3		rd		opcode	
7		5		5		3		5		7	
YPERMR=0001000		YPERMR=00001		src		YPERMR=000		dest		OP=0110011	
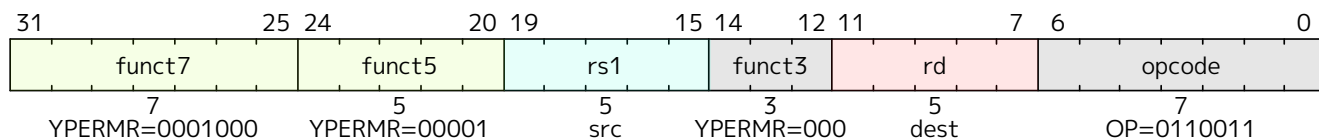
Description

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 7), 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.

XLEN-1	24	23	19	18	17	16	15	SDPLEN+5	6	5	4	2	1	0
Reserved 0		Reserved 1		R	X	ASR	Reserved 1	SDP		C	Reserved 1		LM	W
XLEN-24		5		1	1	1	10-SDPLEN	SDPLEN		1	3		1	1
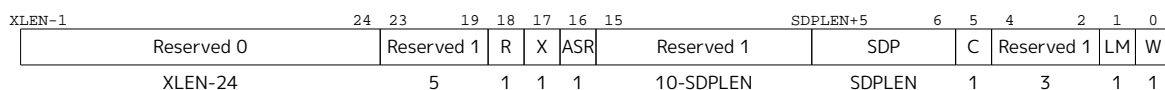
Figure 8. Capability permissions bit field

ℹ️ *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
```

YTYPER

Synopsis

Read capability type

Mnemonic

```
ytyper rd, rs1
```

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		funct5		rs1		funct3		rd		opcode	
7		5		5		3		5		7	
YTYPER=0001000		YTYPER=00010		src		YTYPER=000		dest		OP=0110011	
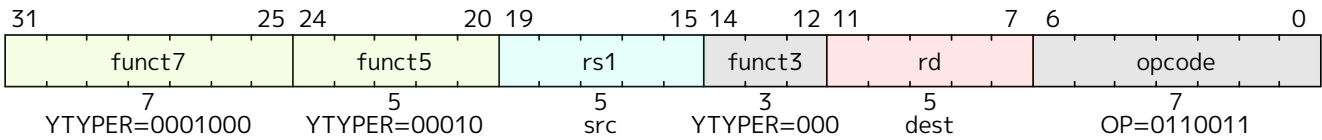
Description

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
```

SRLIY

Synopsis

Logical right shift of Y register

Mnemonic

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

Encoding

31	27 26	20 19	15 14	12 11	7 6	0
funct5	funct7	rs1	funct3	rd	opcode	
5	7	5	3	5	7	
000000	rv64: 1000000 rv32: 0100000	src	SRLIY=101	dest	OP-IMM=0010011	
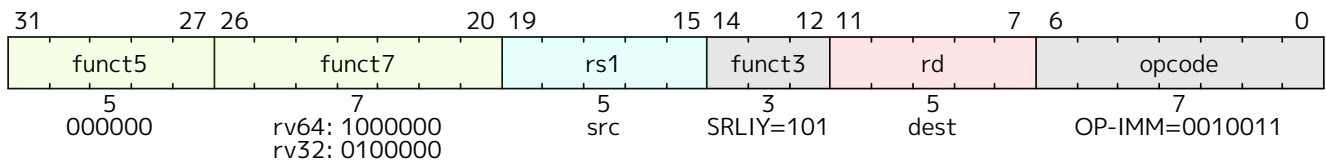
Description

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

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
```

2.5.5. Miscellaneous Instructions to Handle Capability Data

Table 17. Miscellaneous capability instruction summary in RVY

Mnemonic	Description
SYEQ	Full capability bitwise compare, set result true if all bits (including the capability tag) are equal
YLT	Set result true if rs1 and rs1 capability tags match and rs2 bounds and permissions are a subset of rs1
YAMASK	Representable Alignment Mask: Return mask to apply to address to get the requested bounds

SYEQ

Synopsis

Capability equality comparison including capability tag

Mnemonic

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

Encoding

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
SYEQ=0000110	src2	src1	SYEQ=100	dest	OP=0110011
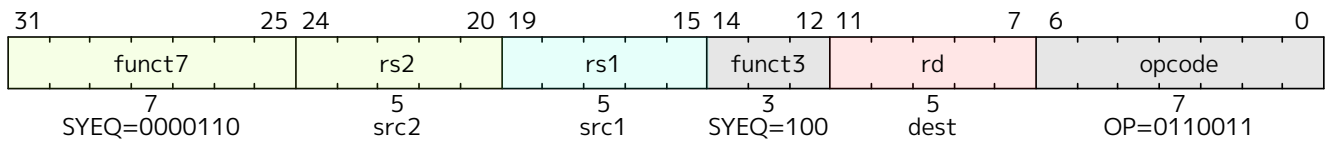
Description

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
```

YLT

Synopsis

Set Capability Subset

Mnemonic

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

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode	
7		5		5		3		5		7	
YLT=0000110		src2		src1		YLT=110		dest		OP=0110011	
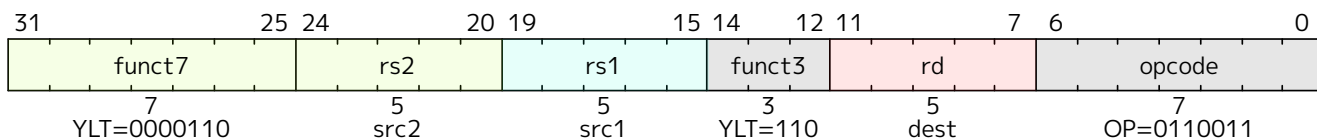
Description

`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
```

YAMASK

Synopsis

Capability alignment mask

Mnemonic

```
yamask rd, rs1
```

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		funct5		rs1		funct3		rd		opcode	

7	5	5	3	5	7
YAMASK=0001000	YAMASK=00111	src	YAMASK=000	dest	OP=0110011
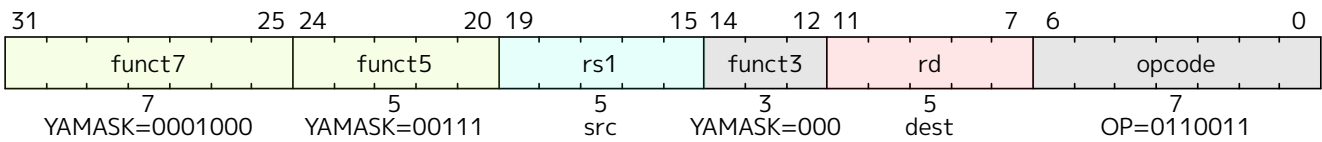
Description

`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 A.1.1.4 for the algorithm used to compute the next representable length.

Included in

RVY

Operation

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

2.5.6. 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

LY

Synopsis

Load capability

Mnemonic

`ly rd, offset(rs1)`

Encoding
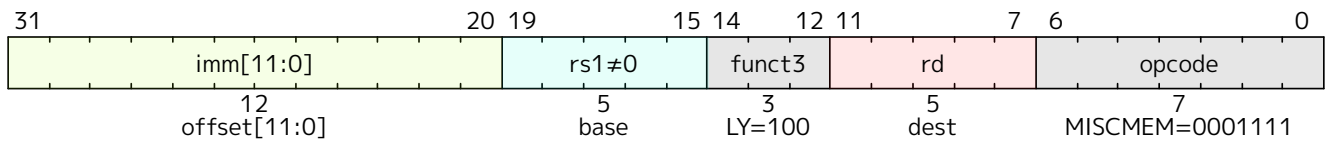
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1≠0	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	LY=100	dest	MISCMEM=0001111

> ℹ️ *Any instance of this instruction with a `rs1`=`x0` will raise an exception, as `x0` is defined to always hold a NULL capability. As such, the encodings with a `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.

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

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 `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.

> ℹ️ *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 }
      }
    }
  }
```

SY

Synopsis

Store capability

Mnemonic

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

Encoding
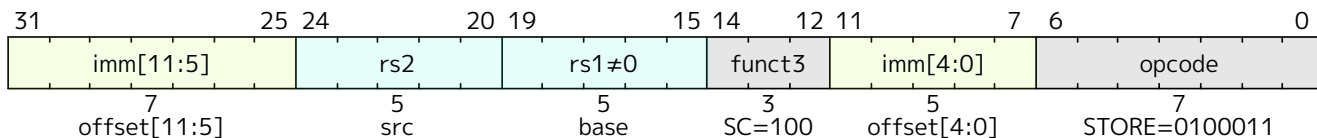
31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		rs2		rs1≠0		funct3		imm[4:0]		opcode	
7		5		5		3		5		7	
offset[11:5]		src		base		SC=100		offset[4:0]		STORE=0100011	

> **i** *Any instance of this instruction with a* `rs1=x0` *will raise an exception, as* `x0` *is defined to always hold a* NULL *capability. As such, the encodings with a* `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
}
      }
    }
  }
}
}
```

2.6. 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 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: ADDY and ADDIY. They must be used for all address incrementing.

> *Integer add (`ADD`) and capability add (ADDY) 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.

2.6.1. 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 is 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, read permission must be set in `rs1`
- For stores, write 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`)

2.6.2. 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 immediate to pc address, return capability.
JAL (RVY)	Jump to pc+offset, link and seal capability to rd.
JALR (RVY)	Indirect jump, link and seal capability to rd.

2.6.3. AUIPC (RVY)

Synopsis

Add upper immediate to pc

Mnemonic

```
auipc rd, imm
```

Encoding
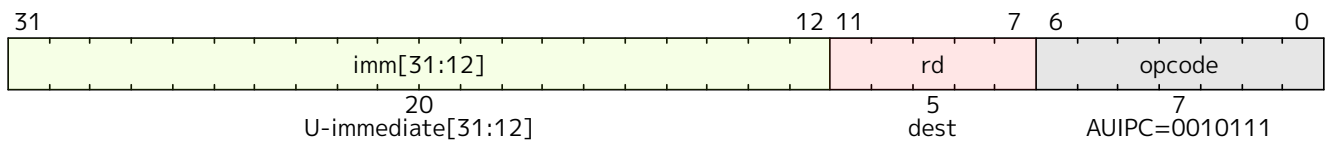
31	12 11	7 6	0
imm[31:12]	rd	opcode	
20	5	7	
U-immediate[31:12]	dest	AUIPC=0010111	

ℹ️ *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). 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
```

2.6.4. 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 Zydefaultcap, 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 11. (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 this same encoding-specified shift value or otherwise allow the capability encoding to set the shift amount.

End new since last ARC review

2.6.5. JAL (RVY)

Synopsis

Immediate offset jump, and link and seal to capability register

Mnemonic

```
jal rd, offset
```

Encoding
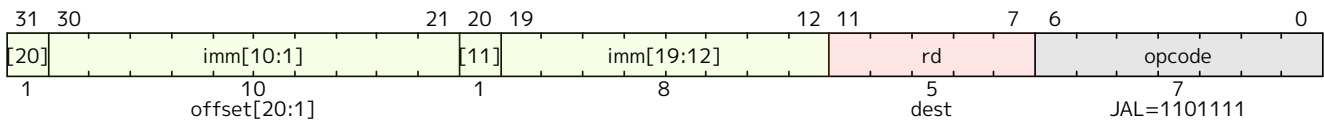
Description

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

2.6.6. JALR (RVY)

Synopsis

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

Mnemonic

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

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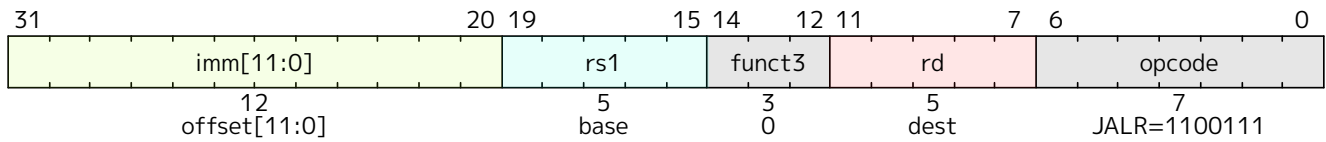
This instruction is extended from the version in the base ISA.

Begin changed since last ARC review

Description

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

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 the zero bit of the target pc address to zero.

2. Clear the capability tag of the target pc if the triple of its CT-field value, `rd`, and `rs1` is not acceptable, as defined by the capability encoding.

   All such triples are acceptable to Zydefaultcap.

3. If the target pc is a sentry capability (for Zydefaultcap: if it has CT-field **1**), then unseal it.

4. If the target pc is (still) sealed, clear its capability tag.

   This is not possible for Zydefaultcap, given its two CT-field values.

5. If `rd≠0`, compute the return capability and install it to `rd`:

   a. add the width of this instruction to the current pc, and

   b. if the capability encoding defines sentry capability CT-field values for return pointers, then seal the return capability as specified.

   *Zydefaultcap always seals return capabilities with CT-field **1**.*

6. Jump to the target pc.

   It is not possible for the target pc to be sealed by the time it is jumped to.

7. Apply any capability encoding specified architectural semantics for the type of `rs1`.

   There are no such additional semantics defined by the capabilities of Zydefaultcap.

The above procedure for JALR (RVY) defines three hooks for use by the capability encoding:

1. Architectural interpretation of zero or more particular CT-field values when a capability is passed as the input to JALR (RVY). These CT-field values are, by definition, sentry capability types. The "architectural interpretation" for these capabilities necessary includes unsealing as part of installation but may include other effects as well.

   > *For example, CHERIoT offers sentries that update a hart's `mstatus.MIE` field when invoked.*

2. A subset of those CT-field values for use by JALR (RVY) to seal the return capability installed in the link register. A sentry return capability grants the right to return exactly to the instruction after the caller's call without relative offset. Moreover, a sentry return capability does not grant the callee access to the memory pointed to by the caller's pc.

3. Conditional acceptability of the triple of the two *register selectors* (`rd`, `rs1`) used in an instance of the instruction together with the CT-field value of the capability in `rs1`.

   > *For Zydefaultcap, there are no conditions and all such triples are acceptable; again the motivation is CHERIoT, which uses register selectors to distinguish between forward and backward control transfers, and restricts which CT-field values are acceptable in each case.*

   > *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 instruction itself if the target pc will raise a CHERI Instruction Access Fault at the target.*

==End changed since last ARC review==

Included in

RVI (RVY modified behavior)

Operation

TODO

2.6.7. 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.*

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

The Zys extension adds the YSENTRY instruction to allow sealing arbitrary capabilities at a CT-field value specified by the capability encoding.

> ℹ️ *In principle, despite the names, the type used by YSENTRY might not be a sentry capability type, though that is not true of the capabilities of Zydefaultcap.*

> ℹ️ *Future extensions to RVY (such as CHERIoT, see uycfg) support additional CT-field values, so this instruction is part of an optional extension instead of the RVY base ISA.*

3.1. Added instructions

3.1.1. YSENTRY

Synopsis

Seal capability as a sentry

Mnemonic

```
ysentry rd, rs1
```

Encoding

31 25	24 20	19 15	14 12	11 7	6 0
funct7	funct5	rs1	funct3	rd	opcode
7	5	5	3	5	7
YSENTRY=0001000	YSENTRY=01000	src	YSENTRY=000	dest	OP=0110011
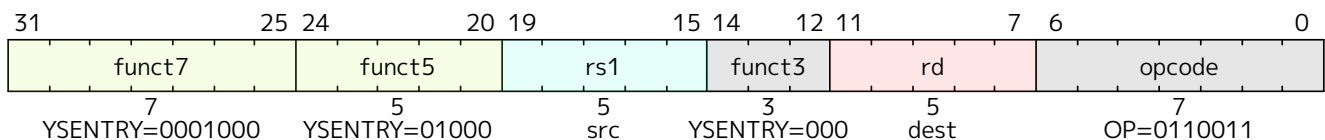
Description

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

Zys

Operation

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

```
RETIRE_SUCCESS
```

Chapter 4. "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.

ℹ️ *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 which execute unmodified. Providing that the privileged execution environment has set up ddc and pc appropriately, non-CHERI aware programs will execute unmodified providing 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.

4.1. CHERI Execution Modes

The two execution modes are:

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

ℹ️ *If RVC encodings are supported, load/store encodings will revert back to their non-CHERI encodings, such as C.LYSP reverting to C.FLWSP for RV32F. This behavior is summarized in Table 48, Table 49, Table 50 and Table 51.*

ℹ️ *Instructions which are modified on an RVY architecture (RVY) revert to their standard behavior.*

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

All memory accesses are implicitly authorized by ddc, including PREFETCH.W (RVY) and PREFETCH.R (RVY). PREFETCH.I (RVY) is authorized by pc instead.

ℹ️ *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 4.4).

(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.

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

> ℹ️ *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.*

4.1.1. 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. Although always present, it only takes effect in code capabilities, i.e., when the X-permission is set. The exact location of the M-bit in the capability format for XLEN=32 and XLEN=64 is described in Zydefaultcap.

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

> ℹ️ *While the M-bit only has an effect when installed in pc, it needs to be encoded in all capabilities. Since indirect jumps copy the full target capability into pc, it allows indirect jumps to change between modes (see Section 4.1.2).*

4.1.2. Changing CHERI Execution Mode

The M-bit of pc can be updated by the instructions listed in Table 21:

Table 21. 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*
YMODESWY	*(Non-CHERI) Address Mode*	Switch execution to *(CHERI) Capability Mode*

The M-bit of a X-permission-granting capability can be read and written by the instructions listed in Table 22:

Table 22. Zyhybrid instructions to observe and update the mode in a capability

Mnemonic	Description
YMODEW	Set capability execution mode
YMODER	Read capability mode

> ℹ️ *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).*

For capabilities that do not grant X-permission, M-bit must always be interpreted and reported as 0.

> ℹ️ *While this is not phrased as an additional rule for YPERMC to follow, beyond those of Section 2.2.9.1, capability encodings may nevertheless take advantage of this implication in their representation of architectural CHERI capabilities.*

4.1.3. 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 0 for *(CHERI) Capability Mode* and 1 for *(Non-CHERI) Address Mode* to x1:

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

> ℹ️ *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.*

4.2. Added instructions

4.2.1. YMODEW

Synopsis

Set capability execution mode

Mnemonic

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

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode	

7	5	5	7	5	7
YMODEW=0000110	src2	src1	YMODEW=111	dest	OP=0110011
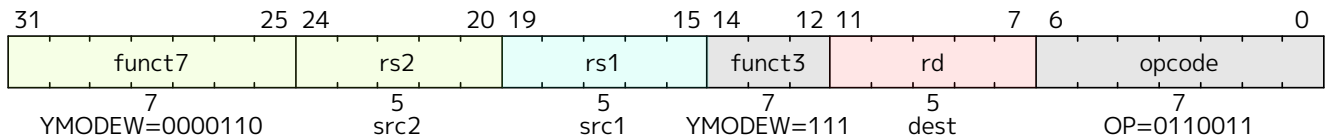
Description

Copy `rs1` to `rd`.

Clear `rd.tag` if `rs1` is sealed.

If `rs1` grants X-permission and `rs1`'s AP-field could have been produced by YPERMC, 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

Otherwise do not update the M-bit.

> ℹ️ *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
```

4.2.2. YMODER

Synopsis

Read capability mode

Mnemonic

```
ymoder rd, rs1
```

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		funct5		rs1		funct3		rd		opcode	

7	5	5	3	5	7
YMODER=0001000	YMODER=00011	src	YMODER=000	dest	OP=0110011
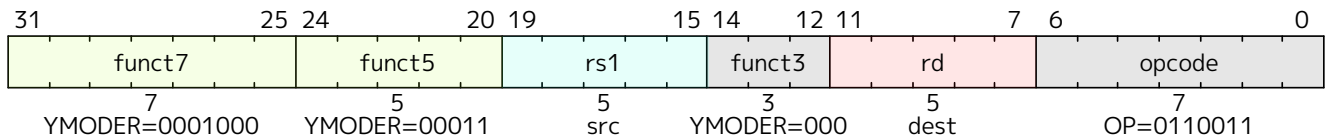
Description

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
```

4.2.3. YMODESWI

See YMODESWY.

4.2.4. YMODESWY

Synopsis

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

Mnemonic

```
ymodeswy
ymodeswi
```

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
funct7		funct5		funct5		funct3		funct5		opcode	

7	5	5	3	5	7
YMODESWY=0001001 YMODESWI=0001010	00000	00000	001	00000	OP=0110011
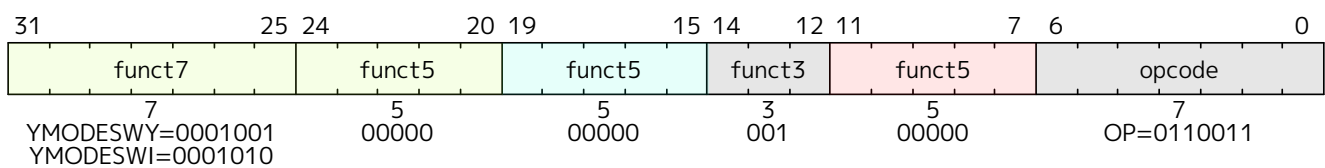
Description

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
```

4.3. Added State

Zyhybrid adds the YLEN-wide CSR shown in Table 23.

Table 23. Unprivileged YLEN-wide CSRs added in Zyhybrid

YLEN CSR	Permissions	Description
ddc	URW	User Default Data Capability

4.3.1. 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).

MXLEN-1		0
V	ddc (Metadata)	
	ddc (Address)	

MXLEN
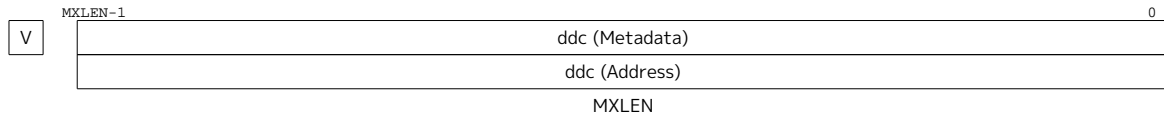
Figure 9. Unprivileged default data capability register

4.4. 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 only access 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 24. 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)

4.4.1. CSRRWI (RVY)

See CSRRCI (RVY).

4.4.2. CSRRS (RVY)

See CSRRCI (RVY).

4.4.3. CSRRSI (RVY)

See CSRRCI (RVY).

4.4.4. CSRRC (RVY)

See CSRRCI (RVY).

4.4.5. 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

31 20	19 15	14 12	11 7	6 0
csr	rs1/uimm	funct3	rd	opcode
12	5	3	5	7
source/dest CSR	source	CSRRS=010	dest	SYSTEM=1110011
	source	CSRRC=011		
	uimm[4:0]	CSRRWI=101		
	uimm[4:0]	CSRRSI=110		
	uimm[4:0]	CSRRCI=111		
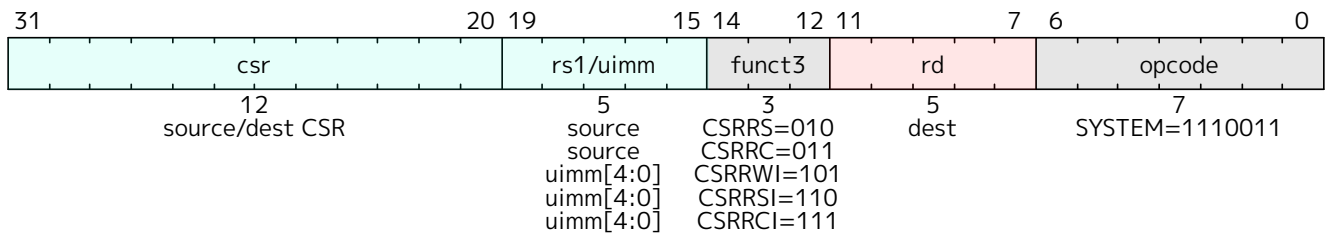
Description

> 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
```

4.4.6. CSRRW (RVY)

Synopsis

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

Mnemonic

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

Encoding

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
source/dest CSR	source	CSRRW=001	dest	SYSTEM=1110011	
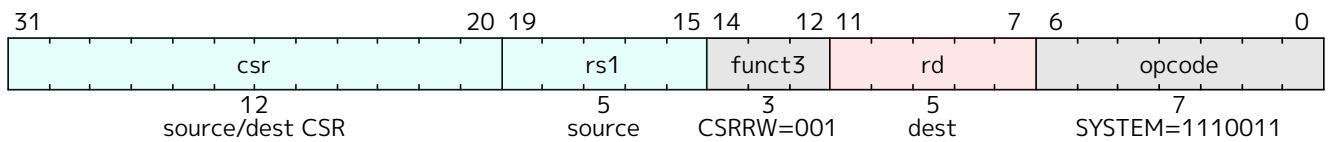
Description

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 5. "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.

> ℹ️ *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 check memory bounds and so it not possible to round a subword access up to a word or larger to gain the reservation set.*

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

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

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
op	aq	rl	LR.*=00000	base	.B=000	rdest[4:0]	AMO=0101111
LR.*=00010					.H=001		

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
op	aq	rl	src	base	width	rdest[4:0]	AMO=0101111
SY=00011					.B=000		
					.H=001		
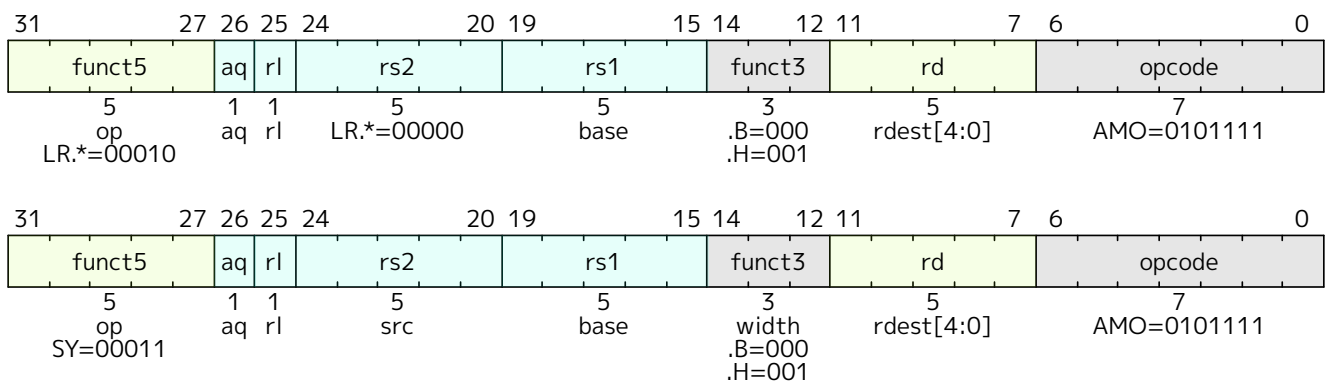
LR.[B|H] behave analogously to LR.[W|D].

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

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

Chapter 6. Vector "V" Extension (RVY)

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

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

i *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 load and stores where the base register is `x0` are reserved applies to all vector memory access instructions.

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

Chapter 7. "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.

7.1. Added Architectural Permissions (AP) Bits

Permission	Type	Comment
SL-permission	Data memory permission	Used to filter the permissions of loaded 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 through 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.

7.2. 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).

7.3. 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 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.

7.4. Interaction with YPERMC and YPERMR

XLEN-1	5	4	3	2
As is	GL	SL	LG	As is
XLEN-5	1	1	1	2
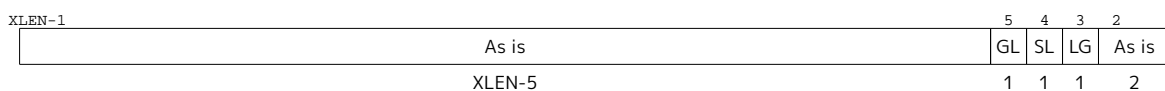
Figure 10. Extended capability permissions bit field (see Figure 7)

The GL(obal) Flag, SL-permission, and LG-permission fields are mapped into the capability permissions bitfield (Figure 7), used by YPERMC and YPERMR, as shown in Figure 10.

7.4.1. YPERMC and the Capability Global (GL) Flag

YPERMC can produce a new capability value with the *Capability Global* 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.

Implementations *are permitted but not mandated* to require that an explicit YPERMC instruction wishing to clear GL(obal) Flag has an input mask that is *entirely 0s* except for GL(obal) Flag (or else clear the capability tag of the resulting capability).

7.4.2. 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 2.2.9.1:

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

7.5. 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.*

7.6. 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.*

7.7. 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.

7.8. 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.

7.9. 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.

7.10. Summary Of System Behavior

Table 25. 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	0	Store data capability unmodified
			1	Store data capability with capability tag cleared

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

Table 26. 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

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

Chapter 8. "Zyseal" Extension for CHERI Capability (Un)Sealing

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

8.1. Explicit Sealing and Unsealing Operations

The RVY base architecture defines sealed capabilitys. The YBLD, JALR (RVY), and YSUNSEAL instruction and the Zys 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 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 ranges 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.

8.2. 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 will 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.

8.3. 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 Zydefaultcap 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 intrinsically.

8.4. Added Architectural Permissions (AP) Bits

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.

8.5. Interaction with YPERMC and YPERMR


```
XLEN-1                         26  25  24  23                                        0
┌──────────────────────────────┬───┬───┬────────────────────────────────────────────┐
│         Reserved 0           │US │SE │                   As is                     │
└──────────────────────────────┴───┴───┴────────────────────────────────────────────┘
          XLEN-26               1   1                       24
```

Figure 11. Extended capability permissions bit field (see Figure 7)

The SE-permission and US-permission fields are mapped into the capability permissions bitfield (Figure 7), used by YPERMC and YPERMR, as shown in Figure 11.

8.6. 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 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`.

8.6.1. 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.

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.*

Included in

Zyseal

Operation

TODO

8.6.2. 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

==End new since last ARC review==

Appendix A: CHERI (RV64Y) Unprivileged Appendix

A.1. Zydefaultcap - The Default Capability Encoding in RVY

This chapter describes the default capability encoding for both RV64Y and RV32Y. It covers the following configurations:

- uycfg.base=0, and
- uycfg.features=0 (no levels support) or 1 (Zylevels1 support)

> **ℹ** *No levels support is specified as an option to allow flexibility for future extensions to reuse the architectural permissions allocated to Zylevels1 in RV32Y. It is not recommended to exclude Zylevels1 until a better alternative exists.*

This minimal default encoding is indicated by uycfg.base=0, uycfg.features=1.

If Zyhybrid is implemented, the specification includes encoding the CHERI Execution Mode in the capability format.

> **⚠** *This chapter will appear as an appendix in the unpriv specification.*

A.1.1. Capability Encoding

The components of a capability, except the capability tag, are encoded as shown in Figure 12 for RV32Y and Figure 13 for RV64Y. Each memory location or register able to hold a capability must also store the capability tag as *out of band* or *hidden* information that software cannot directly set or clear. The capability metadata is held in the most significant bits and the address is held in the least significant bits.

31 30 29	25 24 23	21 20 19 18 17	12 11 10 9	2 1 0

SDP	AP, M	CL	Reserved	CT	EF	L8	T[7:2]	TE	B[9:2]	BE
Address										

32
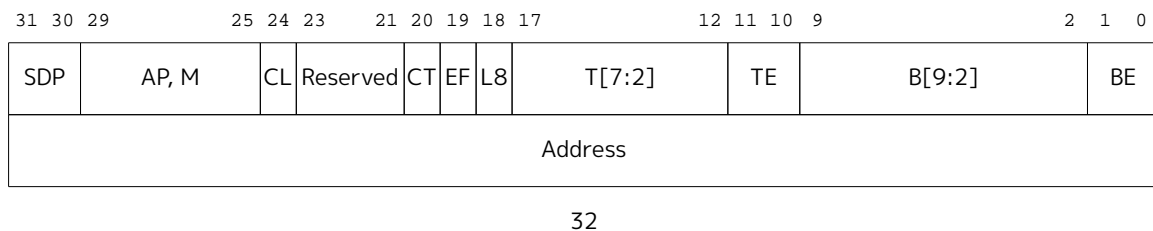
Figure 12. Capability encoding for RV32Y

63	57 56 53 52 51	44 43 42	28 27 26 25	17 16 14 13	3 2 0

Reserved	SDP	M	AP	CL	Reserved	CT	EF	T[11:3]	TE	B[13:3]	BE
Address											

64
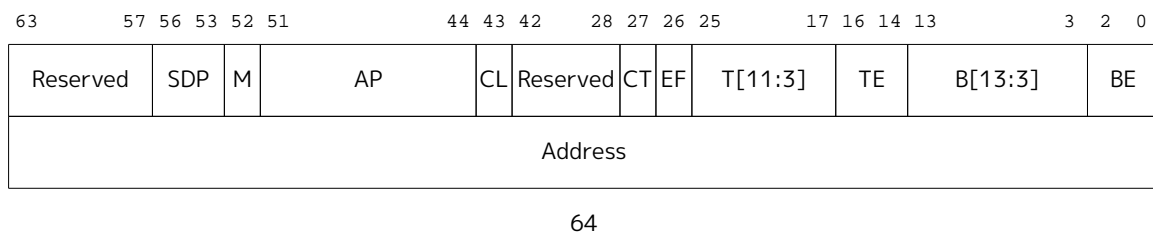
Figure 13. Capability encoding for RV64Y

Reserved bits are available for future extensions to RVY.

> **ℹ** *Reserved bits must be 0 in valid capabilities.*

The encoding of capabilities depends on the extensions supported by the current environment. The following extensions affect the capability encoding:

Zyhybrid

When Zyhybrid is supported, capabilities include an M-bit which is included in the specification below. If not supported the M-bit is reserved and reads as zero.

Zylevels1

The base definition of capabilities in RVY does not have capability-mediated Information Flow Control (IFC) mechanisms. Zylevels1 is available for this and other schemes are likely in the future to add more levels.

Architectural Permissions (AP) Encoding

The bit width of the permissions field depends on the value of MXLEN as shown in Table 27. A 5-bit vector encodes the permissions when RV32Y. For this case, the legal encodings of permissions are listed in Table 29. Certain combinations of permissions are impractical. For example, C-permission is superfluous when the capability does not grant either R-permission or W-permission. Therefore, it is only possible to encode a subset of all combinations.

Table 27. Permissions widths depending on MXLEN

MXLEN	AP-field width	Comment
32	5	Encodes some combinations of 8 permission bits, including the M-bit if Zyhybrid is supported.
64	8	Separate bits for each architectural permission.

For RV32Y, 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 29 for each quadrant.

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

Table 28. Encoding of architectural permissions for RV32Y without Zylevels1

Field[2:0]	R	W	C	LM	X	ASR	Mode[1]	Notes
Quadrant 0: Non-capability data read/write								
bit[2] - write, bit[1] - reserved (0), bit[0] - read								
Reserved bits for future extensions are 0 so new permissions are not implicitly granted								
0							N/A	No permissions
1	✔						N/A	Data RO
2-3	reserved							
4		✔					N/A	Data WO
5	✔	✔					N/A	Data RW
6-7	reserved							
Quadrant 1: Executable capabilities								
bit[0] - M-bit (0-(CHERI) Capability Mode, 1-(Non-CHERI) Address Mode)								
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								
0-2	reserved							
3	✔		✔				N/A	Data & Cap RO (no LM)
4-7	reserved							
Quadrant 3: Capability data read/write								
bit[2] - write, R and C implicitly granted.								
Reserved bits for future extensions must be 1 so they are implicitly granted								
0-2	reserved							
3	✔		✔	✔			N/A	Data & Cap RO
4-6	reserved							
7	✔	✔	✔	✔			N/A	Data & Cap RW

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

> When RV32Y there are many reserved permission encodings (see Table 29). 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 RV32Y, but is not a permission. It is orthogonal to permissions as it can vary arbitrarily using YMODEW.

Table 29. Encoding of architectural permissions for RV32Y with Zylevels1

Field[2:0]	R	W	C	LM	LG	SL	X	ASR	Mode[1]	Notes
Quadrant 0: Non-capability data read/write										
bit[2] - write, bit[1] - reserved (0), bit[0] - read										
Reserved bits for future extensions are 0 so new permissions are not implicitly granted										
0									N/A	No permissions
1	✔								N/A	Data RO
2-3	reserved									
4		✔							N/A	Data WO
5	✔	✔							N/A	Data RW
6-7	reserved									
Quadrant 1: Executable capabilities										
bit[0] - M-bit (0-(CHERI) Capability Mode, 1-(Non-CHERI) Address Mode)										
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										
bit[2] = write, bit[1] reserved, bit[0] = !SL. R and C implicitly granted, LM dependent on W permission.										
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										
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.*

When RV32Y, 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 *in order*:

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

If Zylevels1 is present, the following rules are run *in order*:

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)
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.

When RV64Y, there is a bit per permission as shown in Table 30. A permission is granted if its corresponding bit, and those of any dependent permissions, are set; otherwise, the capability does not grant that permission.

Table 30. Encoding of architectural permissions for RV64Y

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 otherwise.
7	SL-permission if Zylevels1 is implemented; reserved otherwise.

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

1. For RV64Y, the bit assigned to the M-bit must be zero if X-permission isn't set.

2. For RV32Y, the M-bit is only encoded in quadrant 1 and does *not* exist in the other quadrants.

> 🛈 *Future extensions may allow more combinations of permissions, especially for RV64Y.*

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

Software-Defined Permissions (SDP) Encoding

The width of the SDP-field depends on the underlying base architecture. The value of the SDP-field bits of the YPERMR result maps 1:1 to the SDP-field in the capability.

Table 31. SDP widths depending on MXLEN

MXLEN	SDPLEN
32	2
64	4

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

Capability Type (CT) Encoding

==Begin changed since last ARC review==

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 Zys 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 AP-field encoding 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.*

End changed since last ARC review

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

Concept

The bounds encode the base and top addresses that constrain memory accesses. The capability can be used to access any memory location A in the range base ≤ A < top. The bounds are encoded in 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 A.1.1.

The bounds field has the following components:

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

The bit width of T and B are defined in terms of the mantissa width (MW) which is set depending on the value of MXLEN as shown in Table 32.

Table 32. Mantissa width (MW) values depending on MXLEN

MXLEN	MW
32	10
64	14

The exponent E indicates the position of T and B within the capability's address as described in Section A.1.1. The bit width of the exponent (EW) is set depending on the value of MXLEN. The maximum value of the exponent is calculated as follows:

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

The possible values for EW and CAP_MAX_E are shown in Table 33.

Table 33. Exponent widths and CAP_MAX_E depending on MXLEN

MXLEN	EW	CAP_MAX_E
32	5	24
64	6	52

> *The address and bounds must be representable in valid capabilities i.e., when the capability tag is set (see Section A.1.1.4.4).*

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

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

Reconstituting the top two bits of T:

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

The bounds are decoded as shown in Figure 14 and Figure 15.

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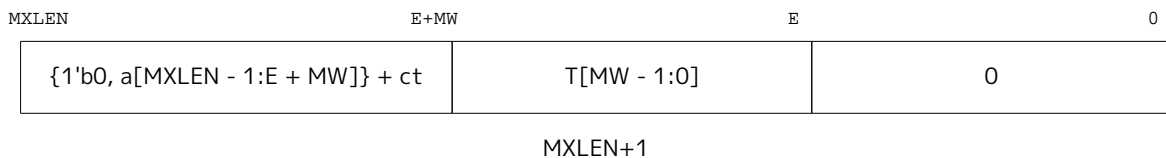
MXLEN+1

Figure 14. Decoding of the MXLEN+1 wide top (t) bound

MXLEN−1	E+MW	E	0
a[MXLEN - 1:E + MW] + cb	B[MW - 1:0]	0	
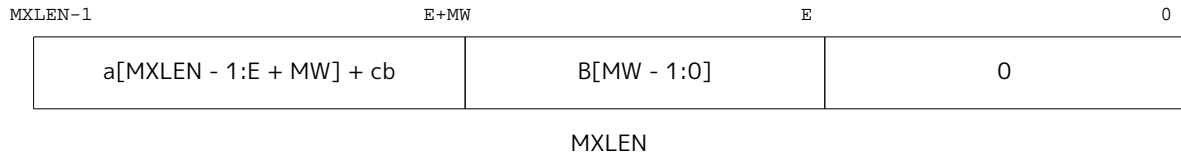
MXLEN

Figure 15. Decoding of the MXLEN wide base (b) bound

Figure 14 and Figure 15 include ranges which may not be present when the bounds are decoded:

- If E = 0 the lower section does not exist.

- If E+MW=MXLEN then the top section is only the least signification bit of c_t for the top bound, and top section doesn't exist for the bottom bound.

- If E+MW>MXLEN then neither top section exists, and so the bounds are calculated with no depending on the address field a.

The corrections c_t and c_b are calculated as as shown below using the definitions in Table 34 and Table 35.

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

ℹ️ *The comparisons in Table 34 and Table 35 are unsigned.*

Table 34. Calculation of top address correction

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

Table 35. 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 RV32Y, 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 RV32Y. 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 RV32Y). 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.

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 $t \geq 2^{MXLEN}$, i.e., t is an MXLEN + 1 bit value. However, b is an MXLEN-bit value and the size mismatch introduces additional complications when decoding, so the following condition is required to correct t for capabilities whose Representable Range wraps the edge of the address space:D

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

ℹ️   *The comparison is unsigned.*

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

ℹ️   *A capability has infinite bounds if E=CAP_MAX_E and it is not malformed (see Section A.1.1.4.4); this check is equivalent to b=0 and $t \geq 2^{MXLEN}$.*

Malformed Capability Bounds

A capability is *malformed* if its bounds cannot be correctly decoded. The following check indicates whether a capability is malformed. `enableL8` is true when RV32Y and false otherwise, indicating whether the `L8` bit is available for extra precision when `EF=1`.

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

Capabilities with malformed bounds:

1. Return both base and top bounds as zero, which affects instructions like YBASER.

2. Cause certain manipulation instructions like ADDIY to always set the capability tag of the result to zero.

AUIPC (RVY) Shift

Begin new since last ARC review

For the capabilities of this chapter, the AUIPC shift value is 12. The bounds encodings just described, for both 32- and 64-bit addresses, have sufficient representability that no AUIPC (RVY) instruction used to reach an in-bounds offset will clear the capability tag.

End new since last ARC review

A.1.2. Integrity of Capabilities

CHERI enforces the following rules for all valid capabilities:

1. The bounds are not malformed.
2. No reserved bit in the capability encoding is set.
3. The permissions can be legally produced by YPERMC.

In all cases the capability could not have been legally created, and so either:

- There has been a corruption of capability state due to memory or logic faults
- There is an 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.*

> *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.*

Implementing these checks is optional, as integrity failures are most likely due to IP compatibility issues.

A.1.3. Encoding of Special Capabilities

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 for RV64Y, 24 for RV32Y), so its bounds cover the entire address space such that the expanded base is 0 and top is 2^{MXLEN}.

Table 36. 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	zero	No meaning since non-executable (RV64Y and Zyhybrid only)
CT	zero	Unsealed
EF	zero	Internal exponent format
L_8	zero	Top address reconstruction bit (RV32Y only)
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

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

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.

> *This capability format has a single encoding for the Infinite capability. The CHERIoT encoding format has separate roots for code and for data, and will be included in a future version of this specification.*

Table 37. Field values of the Infinite capability

Field	Value	Comment
Capability Tag	one	Capability is valid
SDP	ones	Grants all permissions
AP (RV32Y)	0x8/0x9[1] (see Table 29)	Grants all permissions
AP (RV64Y)	0xFF (see Table 30)	Grants all permissions
CT	zero	Unsealed
EF	zero	Internal exponent format
L_8	zero	Top address reconstruction bit (RV32Y only)
T	zeros	Top address bits
T_E	zeros	Exponent bits
B	zeros	Base address bits
B_E	zeros	Exponent bits
Address	zeros[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:

- For RV32Y, the M-bit is set to 1 in the AP-field, giving the value 0x9

- For RV64Y, the M-bit is set to 1 in a separate M field which is *not shown* in the table above.

[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.

A.1.4. 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 A.1.1 is also circular, so address 0 is within the Representable Range of a capability where address 2^{MXLEN} - 1 is within the bounds. However, the decoded top address of a capability is MXLEN + 1 bits wide and does **not** wrap, so a capability with base 2^{MXLEN} - 1 and top 2^{MXLEN} + 1 is not a subset of the infinite capability and does not authorize access to the byte at address 0. Like malformed bounds (see Section A.1.1.4.4), it is impossible for a CHERI core to generate a valid capability with top > 2^{MXLEN}. If such a capability exists then it must have been caused by a logic or memory fault. Unlike malformed bounds, the top overflowing is not treated as a special case in the architecture: normal bounds check rules should be followed.

A.1.5. Representable Range Check

The concept of the *representability check* was introduced in Section 2.2.7.

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.

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

Table 38. Composition of the decoded top address bound

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

The *representable range* defines the range of addresses which do not corrupt the bounds encoding. The encoding was first introduced in Section A.1.1, and is repeated in a different form in Table 38 to aid this description.

For the address to be valid for the current bounds encoding, the value in the *Upper Section* of Table 38 *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 maybe inverted as specified Section A.1.1.4.3. Assuming `(E < (CAP_MAX_E - 1))`, the truth-table for this inversion is as follows:

Table 39. Top bound MSB inversion truth table

input_t[MXLEN:MXLEN-1]	b[MXLEN-1]	output_t[MXLEN:MXLEN-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 39 shows that `output_t[MXLEN]` does not depend on `input_t[MXLEN]` as:

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

This leads to the conclusions:

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

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

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

- when `E == CAP_MAX_E - 2`, `t[MXLEN-1:E] == T[MW-1:0]` and `b[MXLEN-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 $\texttt{s=2}^{E+MW}$, 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 $\texttt{R = B - 2}^{MW-2}$ in Section A.1.1. 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*.

A.2. "Xycheriot" Extension and Capabilities

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

CHERIoT is an existing open-source 32-bit RISC-V CHERI microcontroller ISA (and associated software ecosystem) with commercial silicon in development. This *draft* section explores how the modular structure of, and hooks within, the RV32Y base ISA and its extensions could be used to define a future edition of the CHERIoT ISA atop a ratified RISC-V CHERI. Much of the CHERIoT-specific *detail* is not yet imported hereto, but effort has been made to cite the existing CHERIoT ISA documentation. At the same time, this chapter ponders a few proposed incremental changes to the existing CHERIoT ISA; these are marked as "TODO, research required" and can be ignored by readers focused on RVY.

A.2.1. Required Extensions

Xycheriot is specific to RV32Y. It requires the presence of all of:

- Machine-Level ISA (RVY),
- Zylevels1, and
- Zyseal.

The present specification presumes the **absence** of all of:

- any execution mode other than M-mode (such as VS, HS, VU, S, or U),
- Zyhybrid,
- Zyhybrid for Privileged Architectures, and
- Zys.

> 🛈 *While Xycheriot is nominally compatible with Zyhybrid and 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.*

> 🛈 *While Xycheriot is nominally compatible with Zys, the operating system written for Xycheriot has a security model that presumes the absence of ambient sealing, and the present prose explicitly precludes Zys.*

A.2.2. Capability Load Filter and The Revocation Bitmap

As part of providing *heap temporal* safety, Xycheriot extends the semantics of `ly` to introduce a *capability load filter*. The memory block(s) in which software's dynamic allocation heap(s) are to reside are paired

with a *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 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.

Software dynamic memory allocators can set bits in a revocation bitmap to ensure that additional copies of capabilities to particular objects cannot be constructed (into register files and, so, into memory, too). This facilitates particularly straightforward global revocation of pointers to freed heap objects.

> *The location, size, and number of revokable memory blocks, as well as the mapping function from primal memory address to revocation bitmap address and bit index, are all platform-defined.*

> *See §7.8 / "Temporal safety" of The CHERIoT ISA for further details.*

A.2.3. Capability Encoding

`mycfg` Value

Harts implementing Xycheriot must report a constant `0x02` in `mycfg`. (See uycfg for the details of this CSR.)

> *TODO: That value is, of course, not finalized.*

Bounds Encoding

CHERIoT's bounds encodings use 9-bit top and bottom mantissa fields and a compressed 4-bit exponent value. The resulting encoding significantly differs from the bounds of Zydefaultcap; CHERIoT's encoding...

- has byte alignment only up to length 511.

- does not guarantee a representable region beyond the authorized base and top (except for "one past the end", as required by the C programming language, where the capability's address is equal to its upper bound).

  This requires using a non-default AUIPC shift value.

  > *TODO: As in §7.13.3 / "Capability bounds" in The CHERIoT ISA*

Permissions

Xycheriot defines *one* software defined permission, which we call `U0`. That is, its `SDPLEN` value is `1`.

There are 11 other architectural permission bits in Xycheriot, all of which are from its constituent components:

- R-permission, W-permission, C-permission, LM-permission, X-permission, and ASR-permission from RVY;

- LG-permission, SL-permission, and GL(obal) Flag from Zylevels1; and

- SE-permission and US-permission from Zyseal.

The aggregate permissions bit field (Figure 7) used by YPERMR and YPERMC for Xycheriot, which is completely determined by these components, is thus:

XLEN-1		26	25	24	23	19	18	17	16	15		7	6	5	4	3	2	1	0
Reserved 0			US	SE	Reserved 1		R	X	ASR		Reserved 1		U0	C	GL	SL	LG	LM	W
XLEN-26			1	1	5		1	1	1		9		1	1	1	1	1	1	1

Roots

Xycheriot defines *three* Root capability values:

- Its Root Executable capability value is defined as having GL(obal) Flag set and granting exactly all of X-permission, R-permission, C-permission, LM-permission, ASR-permission, and LG-permission.
- Its Root Data capability value is defined as having GL(obal) Flag set and granting exactly all of R-permission, W-permission, C-permission, LM-permission, LG-permission, and SL-permission.
- It defines a *root sealing* capability value that is unsealed, has full address space bounds, and grants exactly all of SE-permission, US-permission, and U0.

These root capabilities satisfy the following properties by construction. They will necessarily remain true through any series of YPERMC transitions:

w-nand-x

At most one of X-permission or W-permission may be set.

mem-nand-ct

The SE-permission, US-permission, and U0 collectively conflict with either of the R-permission or W-permission 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-permission and/or US-permission) or uninterpreted integers (those granting U0). Capabilities granting permissions from neither set are not distinguished.

Permission Encoding

Xycheriot uses a compressed 6-bit permission field. The compression thereof takes advantage of...

- the constraints on permission transitions defined across the base RVY ISA and the Zylevels1 extension,
- the permission partitioning in Xycheriot's root capability values, and
- two additional constraints on permission transitions:

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

The concrete encoding table is shown in Table 40.

> *This second rule adds R-permission as a dependency of SL-permission. Its sole effect is to exclude local and global capabilities that would otherwise grant exactly all of W-permission, C-permission, and SL-permission. These capabilities were judged to be the least useful of the*

combinations left after the by-construction properties of the roots and the myriad YPERMC rules had pruned the possible combinations. It is easy to verify that all other permission bits are determined by the presence of SL-permission and the absence of R-permission:

- *Granting SL-permission implies granting both W-permission and C-permission (by Zylevels1-2).*
- *Granting W-permission then implies that none of the following are granted:*
  - *X-permission (by w-nand-x),*
  - *ASR-permission (by base-3),*
  - *SE-permission, US-permission, or the SDP* U0 *(by mem-nand-ct).*
- *Not granting R-permission implies granting neither*
  - *LM-permission (by base-2) nor*
  - *LG-permission (by Zylevels1-1).*

The only permission bit left to consider is GL(obal) Flag. Thus, making SL-permission depend on R-permission has the effect of precluding exactly the two stated capability forms.

Table 40. Encoding of permissions for Xycheriot

Bit index						Implicit permissions	Description
5	4	3	2	1	0		
GL	1	1	SL	LM	LG	R, W, C	Memory cap-R/W
	1	0	1	LM	LG	R, C	Memory cap-R/O
	1	0	0	0	0	W, C	Memory cap-W/O
	1	0	0	R	W	∅	Memory data-only
	0	1	ASR	LM	LG	X, R, C	Memory executable
	0	0	SDP U0	SE	US	∅	Sealing

> **ℹ** TODO: See §7.13.1 / "Capability permissions" of The CHERIoT ISA.

CT-field Values and Encoding

Xycheriot uses a 3-bit field to encode CT-field values. Unsealed capabilities, those with type `0`, have an all-zeros type field, and an all-zeros type field always corresponds to an unsealed capability. Otherwise, the interpretation of this field *depends on the capability's X-permission value*: field values `0b001` through `0b111` correspond to types...

- 1 (inclusive) through 7 (inclusive) iff the capability grants X-permission, or
- 9 (inclusive) through 15 (inclusive) iff the capability does not grant X-permission.

That is, it is not possible to express a capability with type between 1 (inclusive) through 7 (inclusive) that does not grant X-permission or, dually, one with type between 9 (inclusive) through 15 (inclusive) that does grant X-permission. Attempts to YSEAL or otherwise construct such a capability, or one with type equal to 8 or above 15, will instead result in a cleared capability tag.

No non-zero CT-field value is considered ambiently available, even the sentry capability types (see below).

> **ℹ** *The lack of ambient types has implications:*

- *The Zys extension (and its YSENTRY instruction) requires an ambiently available type, and so Zys is incompatible with Xycheriot.*

- *If the result of a YBLD instruction has a set capability tag, then it will also be unsealed, even if the input bit pattern was one of a sealed capability.*

> *CHERIoT's RTOS's runtime environment bootstraps, from the few architectural capability sealing types available herein, a much larger space of virtual sealing types which apply to entire software objects rather than pointers to objects. This virtualization takes advantage of the XLEN-bits space of type identifiers offered by Zyseal, even though most of those types have no architectural encoding within capabilities. See cheriot.org/book/memory.html#token_apis for details.*

> *See §7.13.2 / "Sealed capabilities" of The CHERIoT ISA.*

AUIPC (RVY) Shift

AUIPC (RVY) and Xycheriot's own AUICGP define their AUIPC shift value to be **11**. That is, these instructions interpret their 20-bit immediate operand with an 11-bit left shift rather than the traditional 12. Thus, `auipc rd, 0x1` adds `1 << 11` to `pc`'s address and stores the result in `rd`.

> *Recall that the Xycheriot encoding does not guarantee sufficient "out-of-bounds representability" for the traditional 12-bit interpretation. (In fact, it guarantees only that one-past-the-end is representable, as required for capabilities to be suitable lowerings of pointers as defined by the C programming language.)*

A.2.4. Sentries

Xycheriot defines *five* sentry capability CT-field values (1 through 5, inclusive):

- Any of these values may be passed as inputs to JALR (RVY). When such a JALR (RVY) instruction retires, the hart's `mstatus.MIE` bit is updated as per Table 41.

- Two values, 4 and 5, are used by JALR (RVY) to seal the return capability. If the hart's `mstatus.MIE` bit is `0` after the prior instruction has retired, the return capability is sealed with 4; otherwise, 5 is used.

- JALR (RVY) is given conditional behavior based on the register selectors used, as detailed in Table 42. 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.

> *All of the sentry capability CT-field values are encodable if and only if the capability grants X-permission. The remaining two X-permission-associated CT-field values are not sentry capability types and remain available for software's use, as do all of the CT-field values associated with capabilities not granting X-permission.*

Table 41. Additional JALR (RVY) Architectural Semantics

CT-field	IRQs at Retirement (`mstatus.MIE`)
1	Unchanged (as is)
2, 4	Deferred (`0`)
3, 5	Enabled (`1`)

Table 42. JALR (RVY) Conditional Behavior

rs1	rd	Permitted rs1 CT-field-s	Comments
`ra`	`null`	4, 5	Function return
≠`ra`	`null`	0, 1	Tail call
any	∉{ `null`, `ra` }	0, 1	Code outlining
any	`ra`	0, 1, 2, 3	Function call

> ℹ️ See §7.13.2 / "Sealed capabilities" of *The CHERIoT ISA*.

> ℹ️ *TODO, research required: Can we refine that table to introduce "must be unsealed" and "must be sealed" forward arc rows, or do we need a second instruction instead?*
>
> *How hard is it to explain such a thing to LLVM? Would we need to push type refinements up onto C / C++ function pointers?*

A.2.5. Stack High Watermark CSRs

Xycheriot introduces two new XLEN CSRs: `mshwm` and `mshwmb`. 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 `mshwm` and `mshwmb` cause `mshwm` to be updated to the lowest targeted address (rounded down to capability alignment).

> ℹ️ *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 `mshwmb` to the lower bound of the thread's stack capability and `mshwm` to its upper bound. Privileged stack zeroing code can be used to lower `mshwm`, so that all bytes between the addresses held in `mshwm` and `mshwmb` are known to be zero.*

> ℹ️ See §7.15 / "Stack high water mark" of *The CHERIoT ISA*.

A.2.6. New Instructions

AUICGP

Xycheriot defines a new U-type instruction, AUICGP, which shifts (by the AUIPC shift) and adds its 20-bit immediate to the address of mtidc (using the semantics of the ADDY instruction) and stores the resulting capability in a target register, analogously to AUIPC (RVY) and pc.

> ℹ️ *See `AUICGP` in The CHERIoT ISA. Therein, a compartment's globals pointer is not held in a CSR, but rather the `x2` GPR; the use of a CSR is presently experimental. The impact of lowering register pressure is yet to be measured.*

YBNDSRDW

> ℹ️ *TODO: setbounds register that fixes base and rounds length down for representability. (Contrast the rounding behavior of YBNDSW.) See `CSetBoundsRoundDown` in The CHERIoT ISA.*
>
> *Maybe this should be some non-Xycheriot extension.*

YTOPR

> ℹ️ *TODO: Two-operand instruction that fuses* YBASER, YLENR, *and ADD. Easy for microarchitecture given how we unpack capabilities in registers. See* `CGetTop` *in The CHERIoT ISA.*
>
> *Maybe this should be some non-Xycheriot extension.*

Loads and Stores With `x0` Authority

Load and store instructions using `x0` as the base register, such as `ly rd, n(x0)`, would, in the RVY base ISA, necessarily fault due to the clear capability tag of the all-zeros `x0`. Xycheriot (re)defines these reserved instruction values as loads and stores using mtidc as the authority, instead.

> ℹ️ *TODO, research required: we have yet to have a research implementation with this change in place. Performance impacts have yet to be evaluated.*
>
> *Perhaps this should be some non-Xycheriot extension.*

Conditional Branches

RVY reserves uncompressed `beq` and `bne` instruction encodings in which `rs1≥rs2`. Xycheriot defines such instructions with `rs2=0` to be conditional branches on the capability tag of `rs1` being set (`beq`) or clear (`bne`).

> ℹ️ *TODO, research required: we have yet to have a research implementation with this change in place. The impacts on code size and dynamic instruction count are yet to be measured.*
>
> *Maybe this should be some non-Xycheriot extension.*

A.2.7. Miscellaneous Changes

CSR Reset Values

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 Xycheriot's multiple root capability values, we redefine two CSRs' reset values:

mtidc
: The mtidc register's reset value is changed to be Xycheriot's Root Data capability value (with set capability tag).

mscratch
: The mscratch register's reset value is changed to be Xycheriot's *sealing* capability value (with set capability tag).

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.

> ℹ️ *All three of these CSRs reset to Root Executable capabilities (with unspecified addresses), which satisfy this legalization requirement. Neither architecture nor software expects to be able to place capabilities thereby prohibited in these CSRs.*

> *TODO, research required: we can readily additionally require not-X-permission of one of mtidc and mscratch, whichever is not involved in the* `csrrw` *at the start of trap entry. If we can revise trap entry suitably, we could further require not-X-permission of both and not-sealing-root of mtidc.*

JALR (RVY) Exceptions

We permit, but do not require, JALR (RVY) instructions which compute target pcs with a clear capability tag, to raise a trap, rather than retiring. The security of the system is not altered by such behavior, but raising faults at the JALR (RVY) instruction improves the diagnostic and debugging experience.

A.2.8. Porting from CHERIoTv1

Assembler Register Names

CHERIoTv1 differentiated between register selector operands intended to access an integer value (`xN`) from those intended to access a capability value (`cN`). ABI names for registers were similarly differeniated, using a prefix of `c` when the full capability was intended (`csp`, `ca0`, &c). RVY does away with the distinction: all register selectors are `xN` and all ABI names are un-prefixed.

`CGetAddr` and `CSub` Replacement

CHERIoTv1 inherited from CHERI v9, and subsequently offered, a `CGetAddr` instruction which copied a capability while explicitly zeroing its capability tag and metadata, preserving only the address. `CGetAddr` largely dates back to early CHERI research machines, when capability registers were, like floating registers (without Zfinx), distinct from integer registers, and was largely obsolesced by CHERIoTv1 using a merged register file. Since RVY likewise uses a merged regiser file, instructions consuming XLEN-width values use the XLEN-width address field of a capability while ignoring its capability tag and metadata, and instructions producing XLEN-width results write capability values with zero capability tag and metadata. Thus, `addi x1, x2, 0` exactly replaces CHERIoTv1's `CGetAddr x1, c2`; the latter form, and/or a `YADDRR` mnemonic consistent with YADDRW, may still be usefully provided by an assembler for documeting programmer intent.

Similarly, CHERIoTv1's `CSub rd, cs1, cs2` instruction for computing the (integer) differences between two capabilities' address fields, may be replaced with integer subtraction, `sub rd, rs1, rs2`.

Permission Bitfield Semantics

The permissions bit field used by YPERMR (replacing `CGetPerm`) is permuted with respect to CHERIoTv1, and permissions are renamed. Nevertheless, permission **semantics** are in direct correspondence, thus:

CHERIoTv1	Xycheriot
EX	X-permission
SR	ASR-permission
SE	SE-permission
US	US-permission
UO	SDP bit 0
GL	GL(obal) Flag
SL	SL-permission

CHERIoTv1	Xycheriot
LM	LM-permission
LG	LG-permission
MC	C-permission
SD	W-permission
LD	R-permission

The bitwise AND behavior of CHERIoTv1's `CAndPerm` has been replaced with the bitwise AND NOT behavior of YPERMC. Permission masks should thus be permuted and bitwise negated.

Elimination of Special Capability Registers

CHERIoTv1 inherited from University CHERI a notion of "special capability registers" (SCRs; its §7.10) in addition to CSRs. The base RVY ISA has instead extended some CSRs to YLEN. CHERIoTv1's `MTCC`, `MScratchC`, and `MEPCC` SCRs directly correspond with RVY's mtvec, mscratch, and mepc YLEN-wide CSRs. CHERIoTv1 additionally has a `MTDC` SCR, which may yet make an appearance in Xycheriot.

> **ℹ️** *TODO, research required: should we add/keep a direct `MTDC` replacement CSR in addition to mscratch?*

A.3. 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.3.1. RVY added instructions

Table 43. RVY added instructions

Mnemonic	RV32Y	RV64Y	Function
LY	✔	✔	Load capability
SY	✔	✔	Store capability
ADDY	✔	✔	Capability pointer increment
ADDIY	✔	✔	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

Mnemonic	RV32Y	RV64Y	Function
YBLD	✔	✔	Build capability
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.3.2. 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 44. 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.3.3. 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 45. 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.3.4. Zys

Zys adds the sentry capability type.

Table 46. Zys instruction extension

Mnemonic	RV32Y	RV64Y	Function
YSENTRY	✔	✔	Seal capability as a sentry

A.3.5. C (RVY added instructions)

An RVY core which supports C also supports C (RVY added instructions).

C (RVY added instructions) is incompatible with Zcf (RV32) and Zcd (RV64).

Table 47. 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.3.6. RV32 / RV32Y RVC load/store mapping summary

Table 48. 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 49. 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.3.7. RV64 / RV64Y RVC load/store mapping summary

Table 50. 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 51. 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.*

C.LY

see C.LYSP.

C.LYSP

Synopsis

Capability loads (C.LY, C.LYSP), 16-bit encodings

> ℹ️ *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

15 13	12	11 7	6 2	1 0
funct3	imm	rd≠0	imm	op
3	1	5	5	2
RV32Y: C.LYSP=011 RV64Y: C.LYSP=001	[5]	dest	offset[4:3|8:6] offset[4|9:6]	C2=10

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1'	imm	rd'	op
3	3	3	2	3	2
RV32Y: C.LY=011 RV64Y: C.LY=001	offset[5:3] offset[5:4|8]	base	offset[7:6]	dest	C0=00
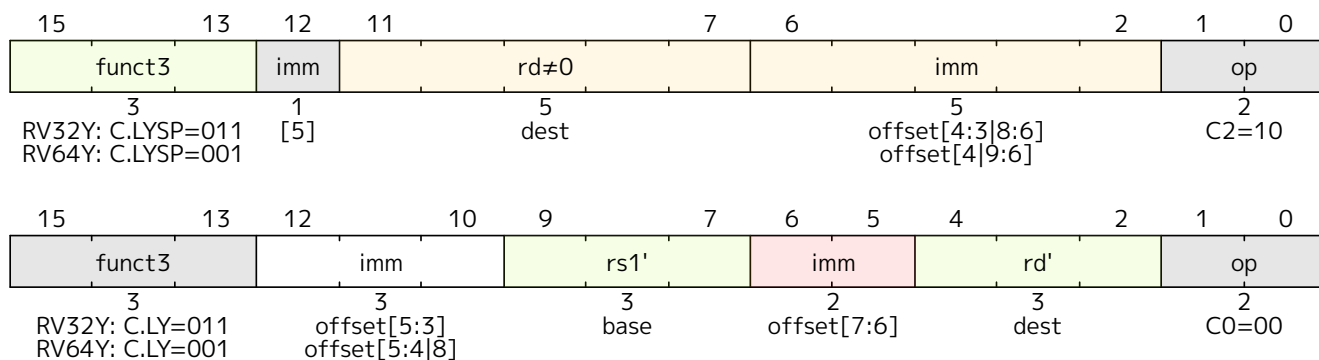
(CHERI) Capability Mode Description

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

C.SY

see C.SYSP.

C.SYSP

Synopsis

Capability stores (C.SY, C.SYSP), 16-bit encodings

ℹ️   *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

15 13	12 7	6 2	1 0
funct3	imm	rs2	op
3	6	5	2
RV32Y: C.SYSP=111 RV64Y: C.SYSP=101	offset[5:3|8:6] offset[5:4|9:6]	src	C2=10

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1'	imm	rs2'	op
3	3	3	2	3	2
RV32Y: C.SY=111 RV64Y: C.SY=101	offset[5:3] offset[5:4|8]	base	offset[7:6] offset[7:6]	src	C0=00
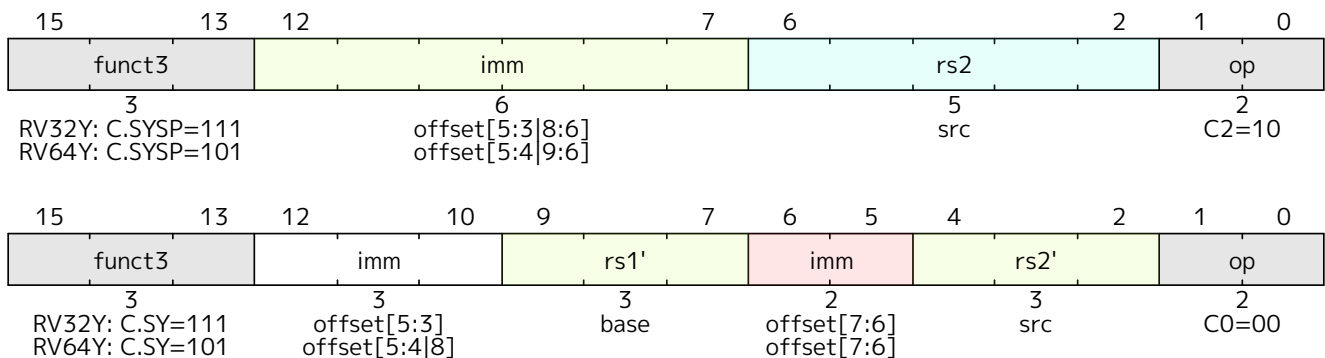
(CHERI) Capability Mode Description

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.3.8. C (RVY modified behavior)

An RVY core which supports C also supports C (RVY modified behavior) which *modifies* the behavior of some instructions.

> ℹ *c.ymv is renamed from* `c.mv` *to avoid ambiguity in disassembly.*

Table 52. 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.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

C.ADDI16SP (RVY)

Synopsis

Stack pointer increment in blocks of 16, 16-bit encoding

Mnemonic

```
c.addi16sp nzimm
```

Expansion

```
addiy sp, sp, nzimm
```

Encoding

15 13	12	11 7	6 2	1 0
funct3	imm[9]	rd/rs1=2	imm[4|6|8:7|5]	op
3	1	5	5	2
C.ADDI16SP=011	nzimm[9]	2	nzimm[4|6|8:7|5]	C1=01
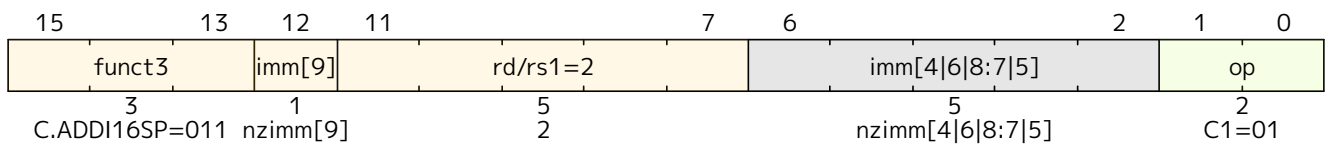
Description

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)))
```

C.ADDI4SPN (RVY)

Synopsis

Stack pointer increment in blocks of 4, 16-bit encoding

Mnemonic

```
c.addi4spn rd', nzuimm
```

Expansion

```
addiy rd', sp, nzuimm
```

Encoding

15 13	12 5	4 2	1 0
funct3	imm	rd'	op
3	8	3	2
C.ADDI4SPN=000	nzuimm[5:4|9:6|2|3]	dest	C0=00
C.ADDI4SPN=000	nzuimm[5:4|9:6|2|3]	dest	C0=00
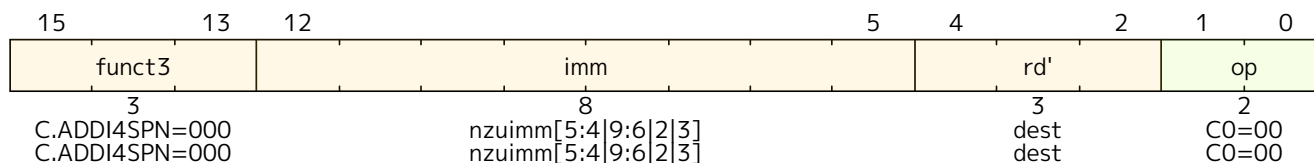
Description

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)))
```

C.YMV

Synopsis

Capability register copy, 16-bit encoding

Mnemonic

```
c.ymv rd, rs2
```

Expansion

```
ymv rd, rs2
```

Suggested assembly syntax

```
ymv rd, rs2
```

> **i** `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

15 12	11 7	6 2	1 0
funct4	rd	rs2	op
4	5	5	2
C.YMV=1000	dest≠0	src≠0	C2=10
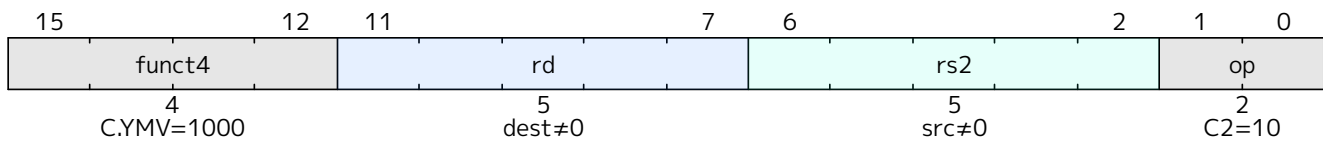
Description

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

C.JR (RVY)

Synopsis

Jump to capability register, 16-bit encoding

Mnemonic

```
c.jr rs1
```

Expansion

```
jalr x0, 0(rs1)
```

Encoding

15	12	11	7	6	2	1	0
funct4		rs1		funct5		op	
4		5		5		2	
C.JR=1000		src≠0		C.JR=00000		C2=10	
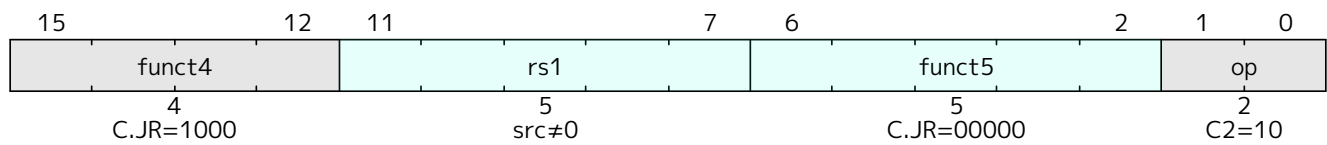
(CHERI) Capability Mode Description

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)

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)

15	13	12		2	1	0
funct3		imm			op	
3		11			2	
int: C.JAL=001		offset[11|4|9:8|10|6|7|3:1|5]			C1=01	
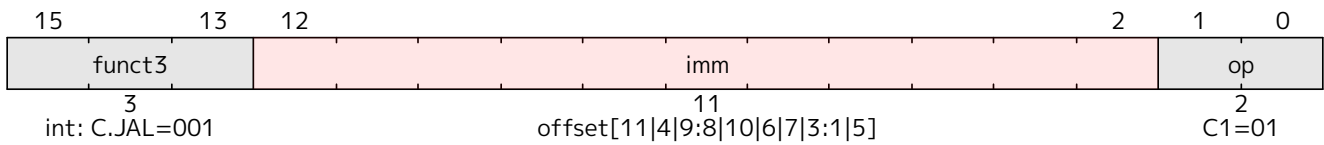
Description

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)

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
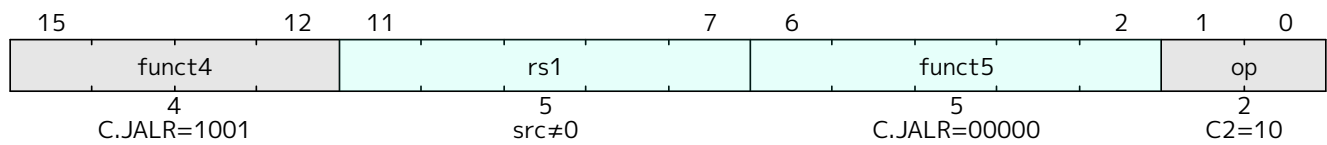
15 12	11 7	6 2	1 0
funct4	rs1	funct5	op
4	5	5	2
C.JALR=1001	src≠0	C.JALR=00000	C2=10

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.3.9. Zalrsc (RVY added instructions)

Specifying RVY and Zalrsc adds atomic capability load and store instructions.

Table 53. Zalrsc (RVY added instructions) instruction extension

Mnemonic	RV32Y	RV64Y	Function
LR.Y	✔	✔	Load Reserved capability
SC.Y	✔	✔	Store Conditional capability

LR.Y

Synopsis

Load Reserved capability

Mnemonic

```
lr.y rd, 0(rs1)
```

Encoding
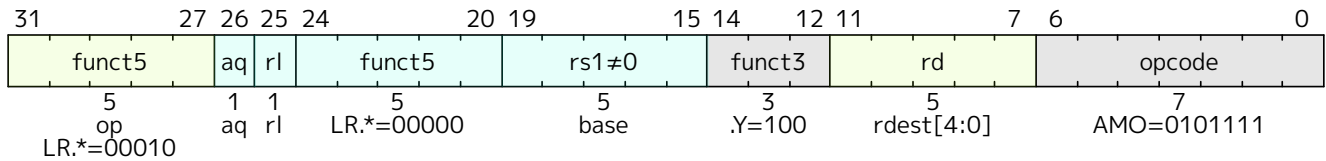
31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct5		aq	rl	funct5		rs1≠0		funct3		rd		opcode	

5	1	1	5	5	3	5	7
op	aq	rl	LR.*=00000	base	.Y=100	rdest[4:0]	AMO=0101111
LR.*=00010							

> ℹ️ *Any instance of this instruction with a* `rs1=x0` *will raise an exception, as* `x0` *is defined to always hold a NULL capability. As such, the encodings with a* `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

SC.Y

Synopsis

Store Conditional capability

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

Mnemonic

```
sc.y rd, rs2, 0(rs1)
```

Encoding
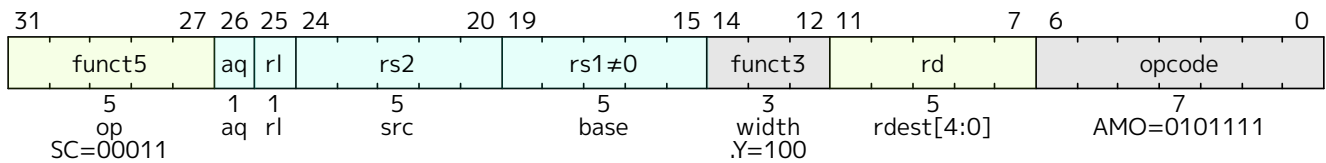
31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1≠0	funct3	rd	opcode
5	1	1	5	5	3	5	7
op	aq	rl	src	base	width	rdest[4:0]	AMO=0101111
SC=00011					.Y=100		

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 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.3.10. Zaamo (RVY added instructions)

Specifying RVY and Zaamo gives Zaamo (RVY added instructions) functionality, which adds atomic capability swap.

Table 54. Zaamo (RVY added instructions) instruction extension

Mnemonic	RV32Y	RV64Y	Function
AMOSWAP.Y	✔	✔	Atomic swap of capabilities

AMOSWAP.Y

Synopsis

Atomic swap of capabilities

Mnemonic

```
amoswap.y rd, rs2, 0(rs1)
```

Encoding
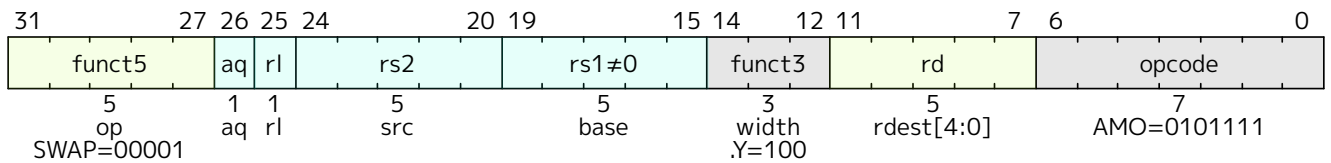
31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct5		aq	rl	rs2		rs1≠0		funct3		rd		opcode	
5		1	1	5		5		3		5		7	
op		aq	rl	src		base		width		rdest[4:0]		AMO=0101111	
SWAP=00001								.Y=100					

> **ℹ** *Any instance of this instruction with a* `rs1=x0` *will raise an exception, as* `x0` *is defined to always hold a NULL capability. As such, the encodings with a* `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.3.11. Zba (RVY added instructions)

Specifying RVY and Zba gives Zba (RVY added instructions) functionality, which adds more instructions.

Table 55. Zba (RVY added instructions) instruction extension

Mnemonic	RV32Y	RV64Y	Function
SH1ADDY	✔	✔	shift and add, representability check
SH2ADDY	✔	✔	shift and add, representability check
SH3ADDY	✔	✔	shift and add, representability check
SH4ADDY (RV64Y)		✔	shift and add, representability check
SH1ADDY.UW (RV64Y)		✔	shift and add unsigned word, representability check
SH2ADDY.UW (RV64Y)		✔	shift and add unsigned word, representability check
SH3ADDY.UW (RV64Y)		✔	shift and add unsigned word, representability check
SH4ADDY.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.*

SH1ADDY

See SH4ADDY (RV64Y).

SH2ADDY

See SH4ADDY (RV64Y).

SH3ADDY

See SH4ADDY (RV64Y).

SH4ADDY (RV64Y)

Synopsis

Shift by *n* and add for address generation (SH1ADDY, SH2ADDY, SH3ADDY, SH4ADDY)

Mnemonics (RVY)

```
sh1addy rd, rs1, rs2
sh2addy rd, rs1, rs2
sh3addy rd, rs1, rs2
```

Mnemonics (RV64Y)

```
sh4addy rd, rs1, rs2
```

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
0 0 1 0 0 0 0		rs2		rs1		func3		rd		0 1 1 0 0 1 1	

SH1ADDY=011
SH2ADDY=101
SH3ADDY=000
rv64: SH4ADDY=111

OP
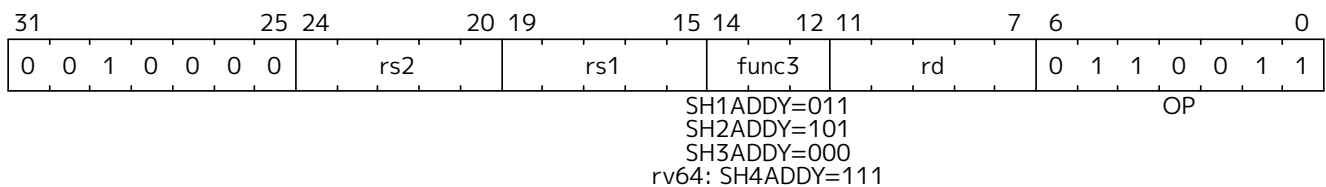
Description

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
```

SH1ADDY.UW (RV64Y)

See SH4ADDY.UW (RV64Y).

SH2ADDY.UW (RV64Y)

See SH4ADDY.UW (RV64Y).

SH3ADDY.UW (RV64Y)

See SH4ADDY.UW (RV64Y).

SH4ADDY.UW (RV64Y)

Synopsis

Shift by n and add unsigned words for address generation (SH1ADDY.UW, SH2ADDY.UW, SH3ADDY.UW, SH4ADDY.UW)

Mnemonics (RV64Y)

```
sh1addy.uw rd, rs1, rs2
sh2addy.uw rd, rs1, rs2
sh3addy.uw rd, rs1, rs2
sh4addy.uw rd, rs1, rs2
```

Encoding

31	25	24	20	19	15	14	12	11	7	6	0
0 0 1 0 0 0 0		rs2		rs1		func3		rd		0 1 1 1 0 1 1	

rv64: SH1ADDY.UW=011
rv64: SH2ADDY.UW=101
rv64: SH3ADDY.UW=000
rv64: SH4ADDY.UW=111

OP-32

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.3.12. Zicbom (RVY modified behavior)

Specifying RVY and Zicbom gives Zicbom (RVY modified behavior) functionality, which extends the checking.

Table 56. 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

CBO.CLEAN (RVY)

Synopsis

Perform a clean operation on a cache block

Mnemonic

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

Encoding

31	20 19	15 14	12 11	7 6	0
funct12	rs1≠0	funct3	funct5	opcode	
12 CBO.CLEAN=00.001	5 base	3 CBO=010	5 CBO=00000	7 MISC-MEM=0001111	
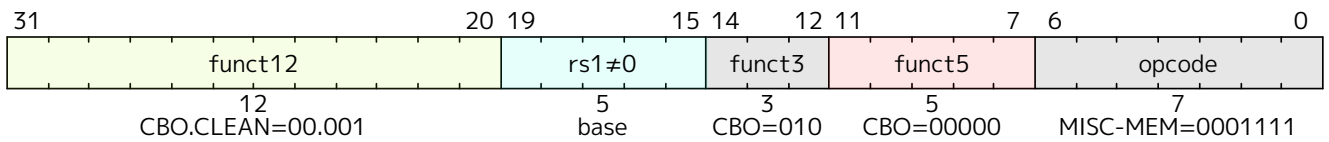
Description

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
```

CBO.FLUSH (RVY)

Synopsis

Perform a flush operation on a cache block

Mnemonic

```
cbo.flush 0(rs1)
```

Encoding

31	20	19	15	14	12	11	7	6	0
funct12		rs1≠0		funct3		funct5		opcode	

12	5	3	5	7
cap: CBO.FLUSH=00.0010	base	CBO=010	CBO=00000	MISC-MEM=0001111

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
```

CBO.INVAL (RVY)

Synopsis

Perform an invalidate operation on a cache block

Mnemonic

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

Encoding

31	20	19	15	14	12	11	7	6	0
funct12		rs1		funct3		funct5		opcode	
12		5		3		5		7	
CBO.INVAL=00.0000		base		CBO=010		CBO=00000		MISC-MEM=0001111	
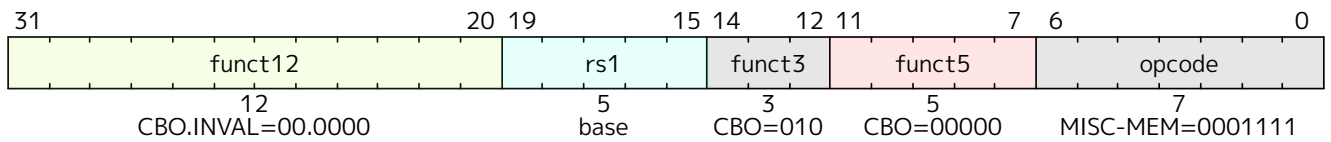
Description

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.3.13. Zicboz (RVY modified behavior)

Specifying RVY and Zicboz gives Zicboz (RVY modified behavior) functionality, which extends the checking.

Table 57. Zicboz (RVY modified behavior) instruction extension

Mnemonic	RV32Y	RV64Y	Function
CBO.ZERO (RVY)	✔	✔	Cache block zero

CBO.ZERO (RVY)

Synopsis

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

Mnemonic

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

Encoding

31 20	19 15	14 12	11 7	6 0
funct12	rs1≠0	funct3	funct5	opcode
12 CBO.ZERO=00.0100	5 base	3 CBO=010	5 CBO=00000	7 MISC-MEM=0001111
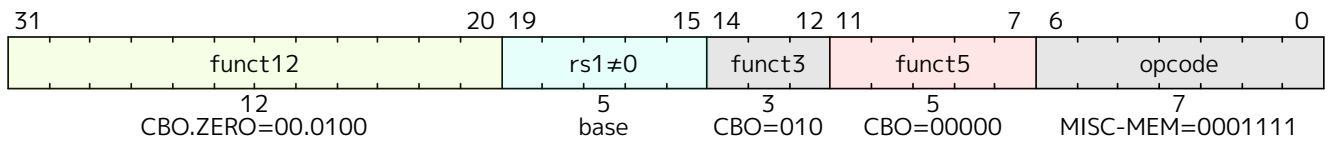
Description

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

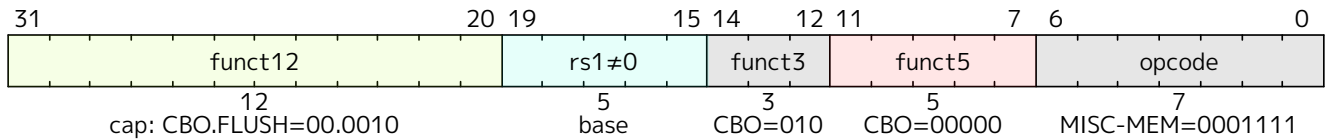Zicboz, RVY

Included in

Zicboz (RVY modified behavior)

Operation

```
TBD
```

A.3.14. Zicbop (RVY modified behavior)

Specifying RVY and Zicbop gives Zicbop (RVY modified behavior) functionality, which extends the checking.

Table 58. Zicbop (RVY modified behavior) instruction extension

Mnemonic	RV32Y	RV64Y	Function
PREFETCH.R (RVY)	✔	✔	Prefetch instruction cache line, always valid
PREFETCH.W (RVY)	✔	✔	Prefetch read-only data cache line
PREFETCH.I (RVY)	✔	✔	Prefetch writable data cache line

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

31 imm[11:5] 25	24 funct5 20	19 rs1≠0 15	14 funct3 12	11 imm[4:0] 7	6 opcode 0
7 offset[11:5]	5 PREFETCH.I=00000	5 base	3 ORI=110	5 zero	7 OP-IMM=0010011

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

Prerequisites

Zicbop, RVY

Included in

Zicbop (RVY modified behavior)

Operation

```
TODO
```

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

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		funct5		rs1≠0		funct3		imm[4:0]		opcode	
7		5		5		3		5		7	
offset[11:5]		PREFETCH.R=00001		base		ORI=110		zero		OP-IMM=0010011	

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

Prerequisites

Zicbop, RVY

Included in

Zicbop (RVY modified behavior)
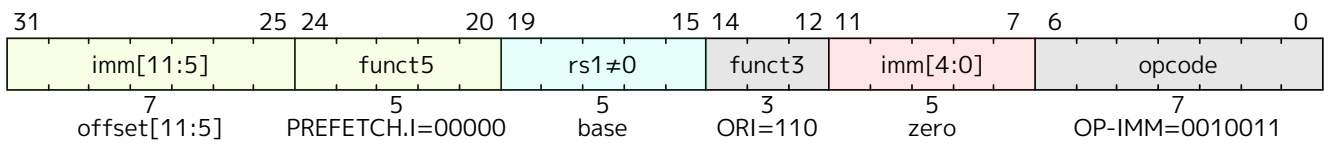
Operation

```
TODO
```

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

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		funct5		rs1≠0		funct3		imm[4:0]		opcode	
7		5		5		3		5		7	
offset[11:5]		PREFETCH.W=00011		base		ORI=110		zero		OP-IMM=0010011	
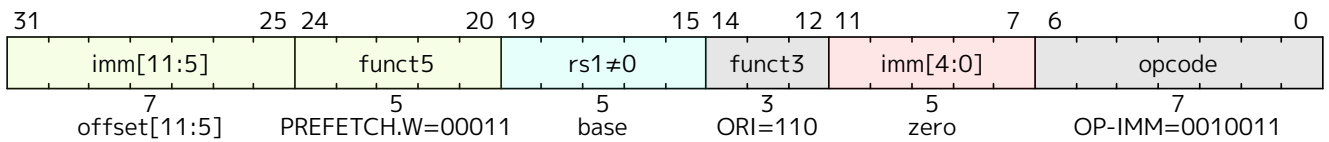
Description

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

Prerequisites

Zicbop, RVY

Included in
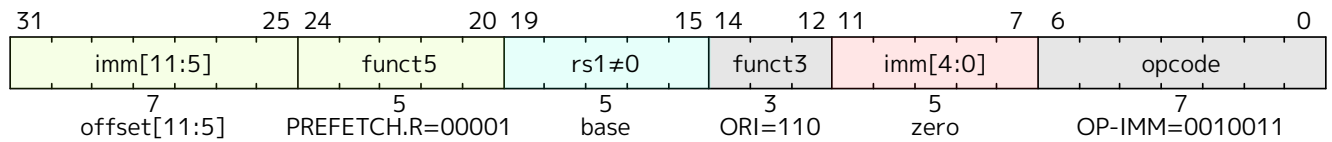
Zicbop (RVY modified behavior)

Operation

```
TODO
```

A.3.15. Zyhybrid

An RVY core which supports Zyhybrid adds the instructions in Table 59.

Table 59. 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*
C.JAL (RV32Y)	✔		Immediate offset jump, and link and seal to capability register, 16-bit encoding

A.3.16. "Zcmp", "Zcmt" (RVY)

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

A.3.17. "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 60.

Table 60. 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

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

15		13	12				8	7			4	3	2	1	0
1	0	1	1	1	0	0	0		rlist			spimm[5:4]		1	0
FUNCT3														C2	

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];
}
```

ℹ️   *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
```

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

15		13	12				8	7			4	3	2	1	0
1	0	1	1	1	0	1	0		rlist			spimm[5:4]		1	0
	FUNCT3														C2

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];
}
```

ℹ️   *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
```

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

15		13	12				8	7			4	3	2	1	0
1	0	1	1	1	1	1	0		rlist			spimm[5:4]		1	0

FUNCT3   C2

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
```

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

15		13	12			8	7		4	3	2	1	0
1	0	1	1	1	1	0	0	rlist		spimm[5:4]		1	0

FUNCT3   C2
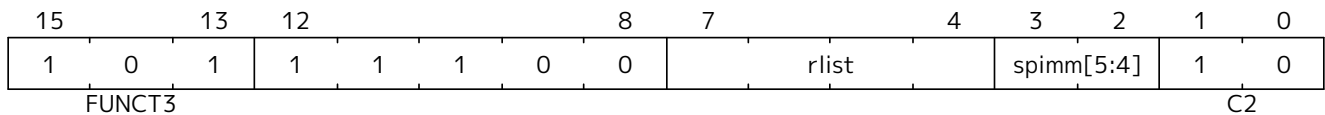
Assembly Syntax:

```
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];
}
```

ℹ️   *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
```

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
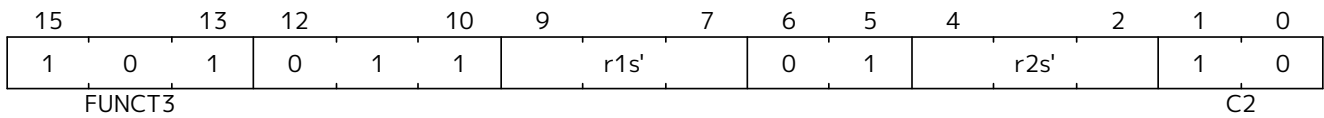
15		13	12		10	9		7	6	5	4		2	1	0
1	0	1	0	1	1		r1s'		0	1		r2s'		1	0
	FUNCT3														C2

> **ℹ** *The encoding uses sreg number specifiers instead of xreg number specifiers to save encoding space. The saved register encoding is shown in Table 60.*

Description

Atomically move two saved capability registers `s0-s7` into `a0` and `a1`.

Prerequisites

C or Zca, RVY, Zcmp

Operation

```
TBD
```

CM.MVA01S (RV32Y)

Synopsis

Move two saved registers into argument registers 0 and 1. 16-bit encoding.

Mnemonic

```
cm.mva01s c1s', c2s'
```

Encoding
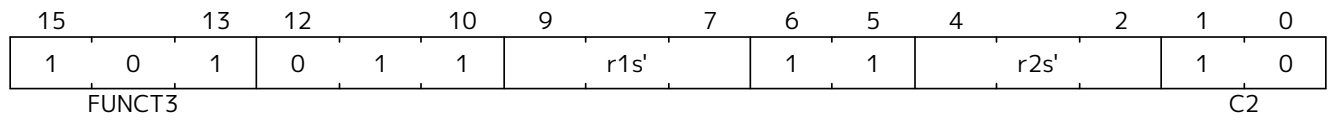
15		13	12		10	9		7	6	5	4		2	1	0
1	0	1	0	1	1		r1s'		1	1		r2s'		1	0
	FUNCT3														C2

> ℹ️ *The encoding uses sreg number specifiers instead of xreg number specifiers to save encoding space. The saved register encoding is shown in Table 60.*

Description

Atomically move two capability registers `a0` and `a1` into `s0-s7`.

Prerequisites

C or Zca, RVY, Zcmp

Operation

```
TBD
```

A.3.18. "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.*

Jump Vector Table CSR (jvt)

The Zcmt jvt CSR is extended to be a full capability.
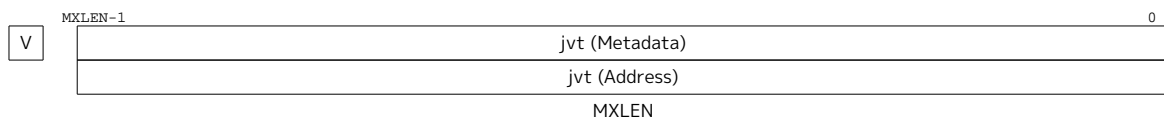
Figure 16. Jump Vector Table Capability register

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.

CM.JALT (RV32Y)

Synopsis

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

Mnemonic (RV32)

```
cm.jalt index
```

Encoding

15		13	12		10	9		2	1	0
1	0	1	0	0	0		index		1	0
	FUNCT3								C2	

> **ℹ** *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
```

CM.JT (RV32Y)

Synopsis

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

Mnemonic (RV32)

```
cm.jt index
```

Encoding

15		13	12		10	9		2	1	0
1	0	1	0	0	0		index		1	0
	FUNCT3								C2	

> **ℹ** *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.4. 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 9. "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.

9.1. 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.

9.1.1. 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.

ℹ | *mtvec (RVY) exists in all CHERI implementations, and so may be used as a source of a Root Executable capability after reset.*

V	Metadata (WARL)		
	BASE [MXLEN-1:2] (WARL)		MODE (WARL)
	MXLEN-2		2
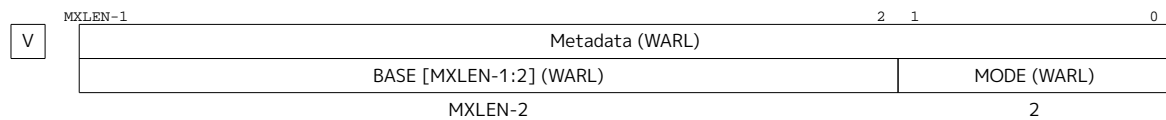
Figure 17. Machine-mode trap-vector base-capability register

The fields in the metadata are WARL as many fields can be implemented as constants.

ℹ | *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.*

ℹ | *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.

> ℹ️ *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.

> ℹ️ *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).

9.1.2. 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.

MXLEN-1	0
V	mscratch (Metadata)
	mscratch (Address)
	MXLEN
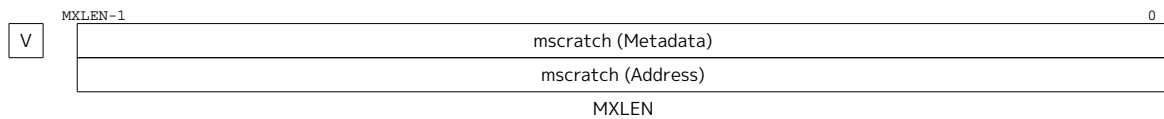
Figure 18. Machine-mode scratch capability register

9.1.3. Machine Exception Program Counter Capability (mepc)

The mepc is extended to hold a capability. Its reset value is nominally a Root Executable capability.

MXLEN-1	0
V	mepc (Metadata, WARL)
	mepc (Address, WARL)
	MXLEN
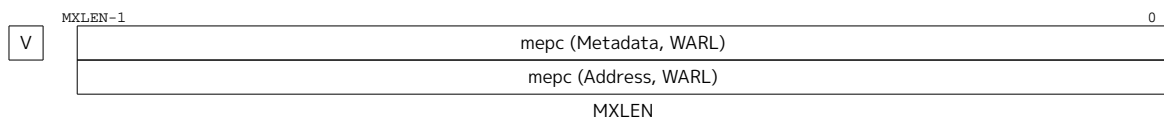
Figure 19. Machine exception program counter capability register

`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).

9.1.4. 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.
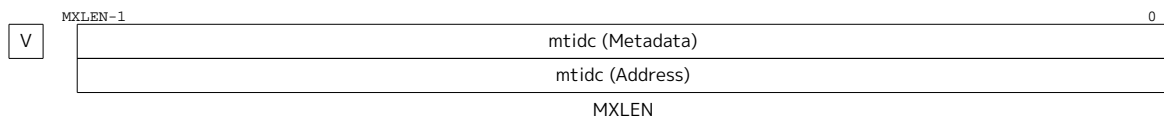
Figure 20. Machine thread identifier capability register

9.1.5. Machine CHERI Capability Encoding (mycfg)

The mycfg register is used to identify which CHERI capability encoding is used by the platform. The capability encoding both determines the in-memory representation of a CHERI capability and entails a set of extensions present atop RVY. The CSR exists in the *writable* namespace and all bits are defined to be WARL, but, absent any future extensions, it is expected that each implementation has exactly one legal value. For the meaning of the individual fields, refer to the uycfg register in the unprivileged specification.
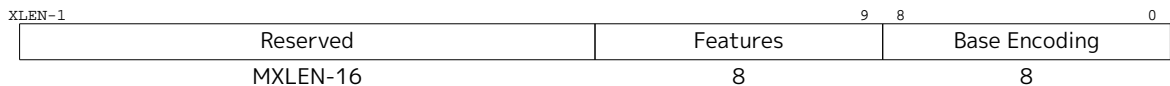
Figure 21. Machine CHERI capability encoding CSR (`mycfg`*) format*

9.2. Machine-Level CSRs modified by RVY

9.2.1. Machine Status Registers (mstatus and mstatush)

The **mstatus** and **mstatush** registers operate as described in mstatus (RVY) with two 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.

> **i** *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.

> **i** *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.*

9.2.2. 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 61 respectively.

Table 61. Synchronous exception priority in decreasing priority order. Entries added in RVY are in **bold**

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, 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	Illegal instruction
	32	**CHERI Instruction Access Fault due to pc ASR-permission clear**
	0	Instruction address misaligned
	8,9,11	Environment call
	3	Environment break
	3	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** Optionally: Load/store/AMO address misaligned
	35, 36, 13, 15, 5, 7	During address translation for an explicit memory access: First encountered **CHERI Load Page Fault[2]**, **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

Priority	Exc.Code	Description
Lowest	35	If not higher priority: CHERI Load Page Fault[3]

[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] The higher priority CHERI Load Page Fault covers capability loads or atomics where the loaded capability tag *is not* checked (Svucrg is implemented) .

[3] The lower priority CHERI Load Page Fault covers capability loads or atomics where the loaded capability tag *is* checked (Svucrglct is implemented).

> ℹ️ *The full details of the CHERI exceptions are in Table 62.*

9.2.3. Machine Trap Delegation Register (medeleg)

Bits 32,33,34,35,36 of medeleg refer to a valid CHERI exception and so can be used to delegate CHERI exceptions to supervisor mode.

9.2.4. Machine Trap Value Register (mtval)

For all CHERI faults, mtval is written with the MXLEN-bit effective address which caused the fault.

```
MXLEN-1                                                    0
┌──────────────────────────────────────────────────────────┐
│                         mtval                              │
└──────────────────────────────────────────────────────────┘
                          MXLEN
```
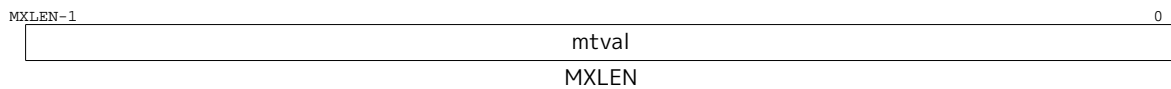
Figure 22. Machine trap value register

9.2.5. "Smstateen/Ssstateen" Integration

The TID bit in `mstateen0` controls access to the stidc CSR.

```
 63     62      61     60     59     58      57     56    55                                    48
┌──────┬──────┬──────┬──────┬──────┬──────┬────────┬──────┬───────────────────────────────────────┐
│ SE0  │ENVCFG│ WPRI │CSRIND│ AIA  │IMSIC │CONTEXT │P1P13 │                 WPRI                   │
└──────┴──────┴──────┴──────┴──────┴──────┴────────┴──────┴───────────────────────────────────────┘
 47                                                                                              32
┌────────────────────────────────────────────────────────────────────────────────────────────────┐
│                                          WPRI                                                     │
└────────────────────────────────────────────────────────────────────────────────────────────────┘
 31                                                                                              16
┌────────────────────────────────────────────────────────────────────────────────────────────────┐
│                                          WPRI                                                     │
└────────────────────────────────────────────────────────────────────────────────────────────────┘
 15                                                                     4      3      2     1     0
┌──────────────────────────────────────────────────────────────────────┬──────┬──────┬─────┬─────┐
│                              WPRI                                     │ TID  │ JVT  │FCSR │  C  │
└──────────────────────────────────────────────────────────────────────┴──────┴──────┴─────┴─────┘
```

Figure 23. Machine State Enable 0 Register (`mstateen0`*)*

9.3. 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.

9.3.1. 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.
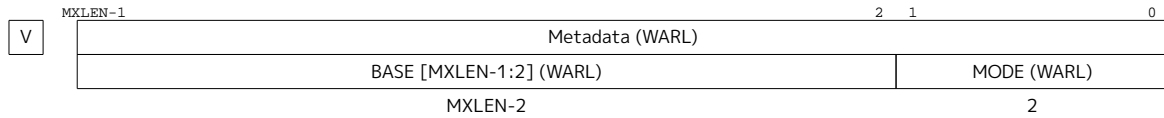
Figure 24. Supervisor trap-vector base-capability register

The handling of stvec (RVY) is otherwise identical to mtvec (RVY), but in supervisor mode.

9.3.2. 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.
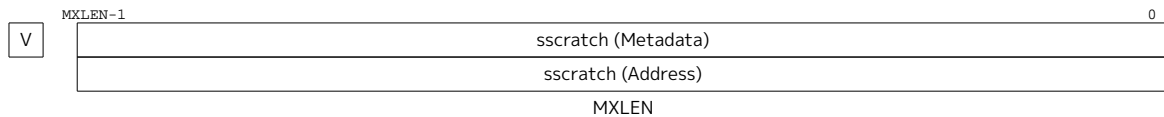
Figure 25. Supervisor scratch capability register

9.3.3. 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 76, 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.
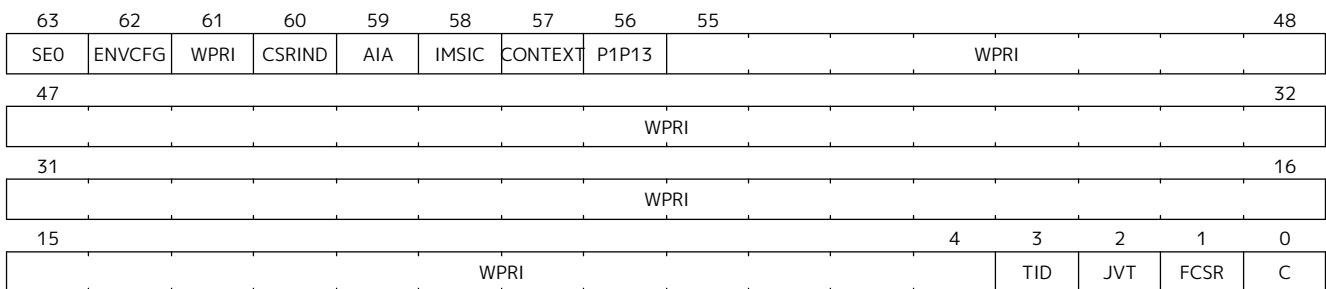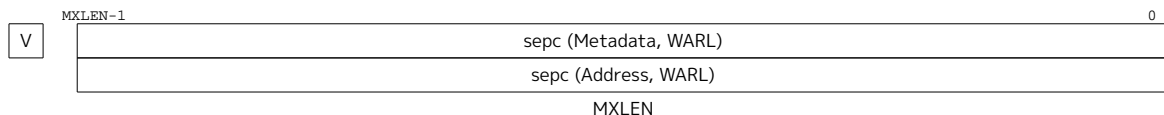

Figure 26. Supervisor exception program counter capability register

9.3.4. 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.
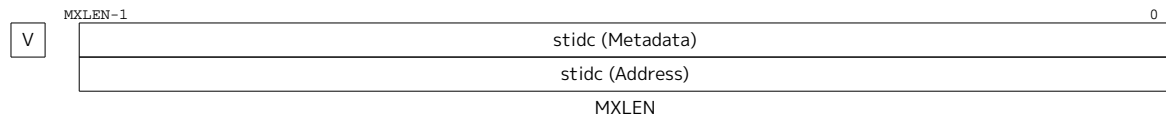
Figure 27. Supervisor thread identifier capability register

9.3.5. Supervisor CHERI Capability Encoding (sycfg)

The sycfg register is used to identify which CHERI capability encoding is used by the platform. The capability encoding both determines the in-memory representation of a CHERI capability and entails a set of extensions present atop RVY. The CSR exists in the *writable* namespace and all bits are defined to be WARL, but, absent any future extensions, it is expected that each implementation has exactly one legal value. For the meaning of the individual fields, refer to the uycfg register in the unprivileged specification.
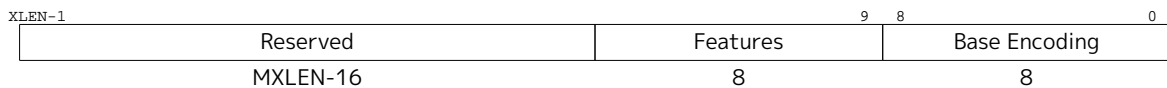
Figure 28. Supervisor CHERI capability encoding CSR (`sycfg`*) format*

9.4. Supervisor-Level CSRs modified by RVY

9.4.1. 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.

9.4.2. Supervisor Trap Value Register (stval)

stval is updated following the same rules as Section 9.2.4 for CHERI exceptions and CHERI page faults which are delegated to HS-mode or S-mode.

9.4.3. "Smstateen/Ssstateen" Integration

The TID (thread ID) bit in `sstateen0` controls access to the utidc CSR. See utidc for a description of the usage.
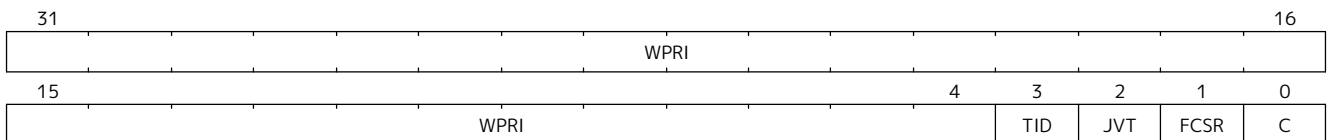
Figure 29. Supervisor State Enable 0 Register (`sstateen0`*)*

9.5. 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.

> ℹ️ auth_cap *is* rs1*, unless in (Non-CHERI) Address Mode when it is* ddc *(if Zyhybrid is*

> *implemented).*

Table 62. 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 have 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)	Illegal instruction	CHERI Instruction Access Fault	pc does not have 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 have 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 have W-permission
All atomics, CBO.CLEAN, CBO.FLUSH, CBO.INVAL	34	CHERI Store/AMO Access Fault	`auth_cap` does not have 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.
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.*

9.6. CHERI Exceptions and speculative execution

should be non-normative - and needs more details - move to appendix?

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.

9.7. 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 63. 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 capability tag is set to one.

Memory regions that do not have the *CHERI Capability Tag* PMA do not require storage for capability tags.

9.8. Virtual Memory

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.*

9.9. 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.

9.9.1. SRET (RVY)

See MRET (RVY).

9.9.2. MRET (RVY)

Synopsis

Trap Return (MRET, SRET)

Mnemonics

```
mret
sret
```

Encoding

31	20 19	15 14	12 11	7 6	0
funct12	rs1	funct3	rd	opcode	

```
        12                      5        3         5           7
MRET=001100000010               0      PRIV=0      0      SYSTEM=111011
SRET=000100000010
```
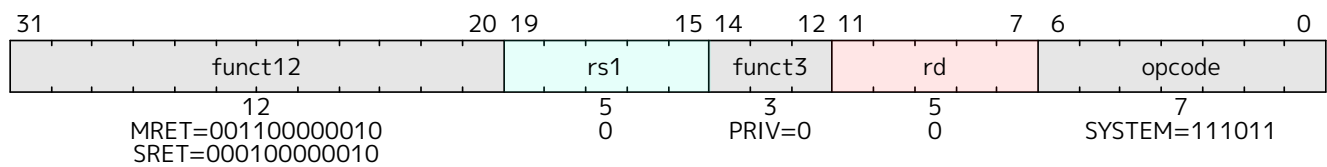
Description

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 10. "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 *disabling CHERI*).

> ℹ️ *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

> ℹ️ *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.

> ℹ️ *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 do 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 64 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 64. 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 (see Zyhybrid).

Chapter 11. "Supervisor-Level ISA for Virtual Memory (RV64Y)" Extension, Version 1.0 for RV64Y

Virtual memory support for RV64Y requires at least one additional bit to be allocated in the page table entries, to control access to capabilities in virtual memory pages.

11.1. Capability Read-Write (CRW) Bit

Supervisor-Level ISA for Virtual Memory (RV64Y) defines the Capability Read-Write (CRW) bit in Page Table Entries (PTEs) for Sv39, Sv48, and Sv57 virtual memory systems on RV64Y harts. The CRW bit controls whether capabilities with their capability tag set can be written to and loaded from a virtual page.

> ℹ️ *Sv32 does not have any spare PTE bits, and so this bit does not exist for RV32.*

> ℹ️ *Any RV64Y hart that supports Sv39 must also implement the PTE.CRW bit.*

11.1.1. 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*

11.2. CHERI page faults

CHERI adds the concept of *CHERI page faults*. They are split into :

- CHERI Load Page Fault (cause value 35), and
- CHERI Store/AMO Page Fault (cause value 36)

They are prioritized against other fault types as shown in Table 61.

The PTE.CRW bit allows CHERI Store/AMO Page Faults to be raised.

CHERI harts which implement Sv39 must also implement a revocation scheme to prevent use-after-free attacks.

The current revocation scheme (Svucrg) also allows CHERI Load Page Faults to be raised.

> ℹ️ *A future extension may define an improved scheme.*

11.2.1. Extending the Page Table Entry Format

The page table entry format remains unchanged for Sv32. However, a new bit, Capability Read-Write (CRW), is added to leaf PTEs in Sv39, Sv48 and Sv57 as shown in Figure 30, Figure 31 and Figure 32 respectively. For non-leaf PTEs this bit remains reserved and must be cleared by software for forward compatibility, or else a page-fault exception is raised. Additionally, if the hypervisor extension is enabled this bit remains reserved for leaf and non-leaf PTEs used in guest address translation.

63	62 61	60 59 58	57	54 53	28 27	19 18	10 9	8 7	6	5	4	3	2	1	0
N	PBMT	Reserved	CRW	Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
1	2	2	1	4	26	9	9	2	1	1	1	1	1	1	1	1
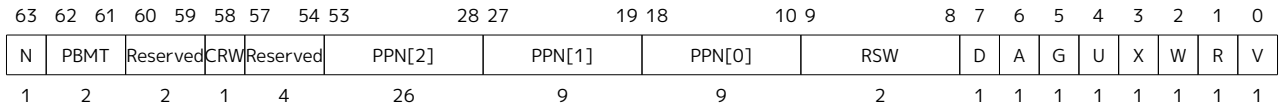
Figure 30. Sv39 page table entry

63	62 61	60 59 58	57	54 53	10 9	8 7	6	5	4	3	2	1	0
N	PBMT	Reserved	CRW	Reserved	PPN	RSW	D	A	G	U	X	W	R	V
1	2	2	1	4	44	2	1	1	1	1	1	1	1	1

53	37 36	28 27	19 18	10
PPN[3]	PPN[2]	PPN[1]	PPN[0]
17	9	9	9
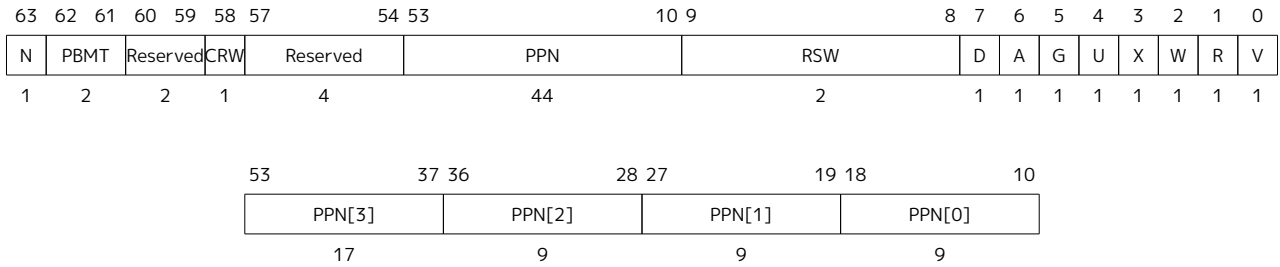
Figure 31. Sv48 page table entry

63	62 61	60 59 58	57	54 53	10 9	8 7	6	5	4	3	2	1	0
N	PBMT	Reserved	CRW	Reserved	PPN	RSW	D	A	G	U	X	W	R	V
1	2	2	1	4	44	2	1	1	1	1	1	1	1	1

53	46 45	37 36	28 27	19 18	10
PPN[4]	PPN[3]	PPN[2]	PPN[1]	PPN[0]
8	9	9	9	9
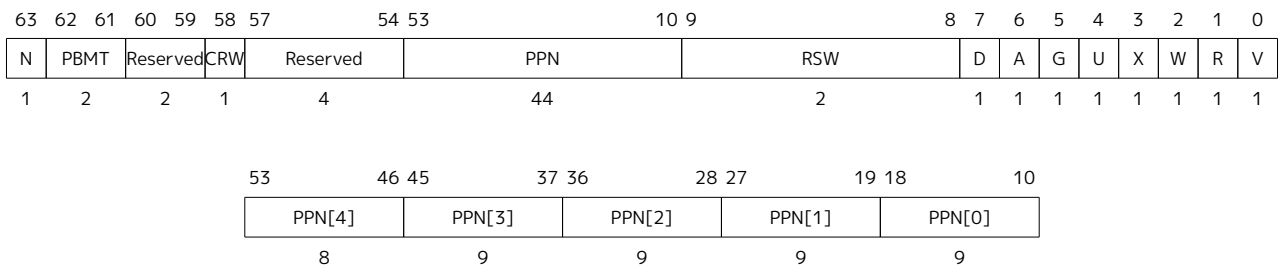
Figure 32. Sv57 page table entry

The CRW bit indicates whether reading or writing capabilities with the capability tag set to the virtual page is permitted. When the CRW bit is set, capabilities are written as usual.

If the CRW bit is clear then:

- When a capability load or AMO instruction is executed, the capability tag bit of the loaded capability is cleared before it is written to the destination register.
- A *CHERI Store/AMO Page Fault* exception is raised when a capability store or AMO instruction is executed and the capability tag bit of the to-be-stored capability is set.

Table 65. Summary of memory access behavior depending on CRW in the PTEs

PTE.CRW	Instruction	Behavior
0	Capability load	Set loaded capability tag to zero
0	Capability store/AMO	Raise a *CHERI Store/AMO Page Fault* if the capability tag of the to-be-stored capability is set
1	Any	Normal operation.

> ℹ️ *The capability tag bit of the stored capability is checked after it is potentially cleared due to missing permissions. Therefore, the behavior in this section isn't relevant if:*

- *The authorizing capability doesn't have C-permission.*
- *Any extension-specific mediation has already cleared the capability tag of the to-be-stored capability.*

11.3. Invalid 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 76, 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 to the implementation to decode the bounds for invalid addresses reduces the size of bounds comparators from 64-bits to the supported virtual address width.*

11.3.1. 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.*

11.3.2. 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.

11.3.3. 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.

11.4. Integrating RVY with Debug

11.4.1. 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.

> ℹ️ *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.*

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.

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 77.

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.

DXLEN−1   0
| dpc |
DXLEN

Figure 33. 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.

DXLEN−1   0
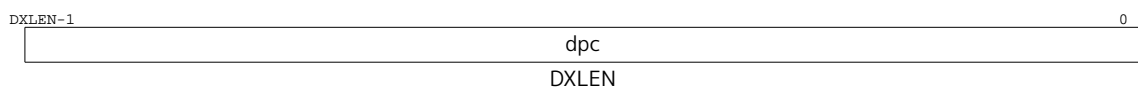| dscratch0 |
DXLEN

Figure 34. 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.

```
DXLEN-1                                                                    0
┌──────────────────────────────────────────────────────────────────────────┐
│                             dscratch1                                       │
└──────────────────────────────────────────────────────────────────────────┘
                                DXLEN
```

Figure 35. Debug scratch 1 register

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.

```
    MXLEN-1                                                                  0
┌───┐┌──────────────────────────────────────────────────────────────────────┐
│ V ││                         dpc (Metadata)                                 │
└───┘├──────────────────────────────────────────────────────────────────────┤
     │                         dpc (Address)                                  │
     └──────────────────────────────────────────────────────────────────────┘
                                MXLEN
```

Figure 36. 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, a 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).

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.
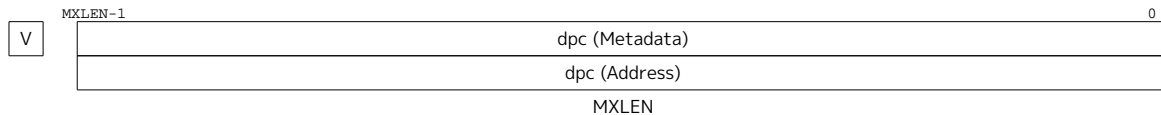
```
    MXLEN-1                                                                  0
┌───┐┌──────────────────────────────────────────────────────────────────────┐
│ V ││                      dscratch0 (Metadata)                              │
└───┘├──────────────────────────────────────────────────────────────────────┤
     │                      dscratch0 (Address)                               │
     └──────────────────────────────────────────────────────────────────────┘
                                MXLEN
```

Figure 37. Debug scratch 0 capability register

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
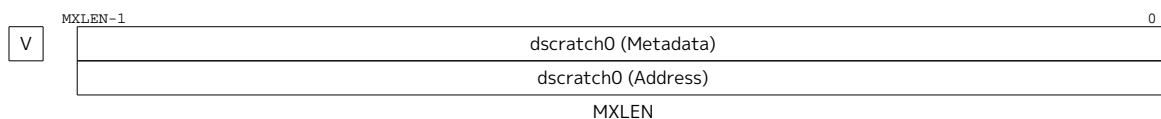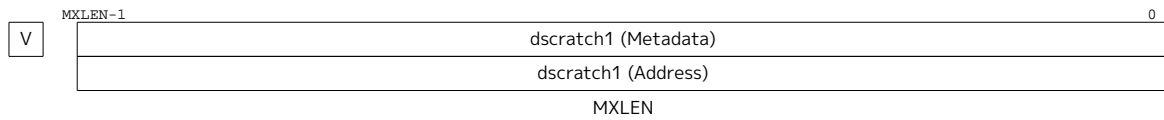
are UNSPECIFIED.

Figure 38. Debug scratch 1 capability register

Debug Root Capability Selector (drootcsel)

drootcsel is a debug mode accessible integer CSR. The address and access details are shown in Table 23.

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.
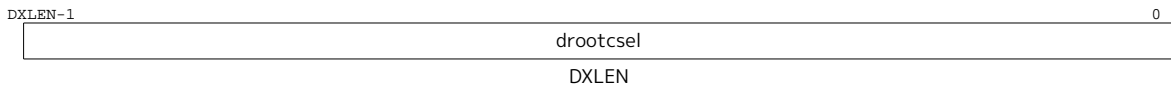
Figure 39. Debug root capability register

Debug Root Capability Register (drootc)

drootc is a debug mode accessible capability CSR. The address and access details are shown in Table 23. It exposes the capability selected by drootcsel.

If Zyhybrid is implemented, the Root Executable exposed when drootcsel is `0` is further specified thus:

- 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 subsequent instructions execution 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.
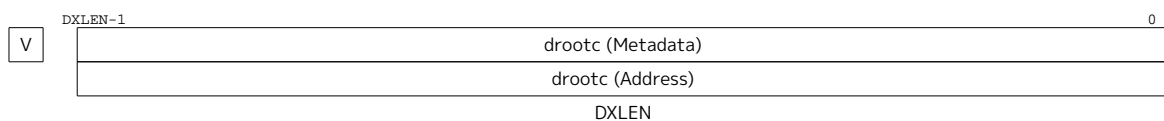
Figure 40. Debug root capability register

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.

DRET (RVY)

Synopsis

Debug Return (DRET)

Mnemonic

```
dret
```

Encoding

31	20	19	15	14	12	11	7	6	0
funct12		rs1		funct3		rd		opcode	

12	5	3	5	7
DRET=011110110010	0	PRIV=0	0	SYSTEM=111011
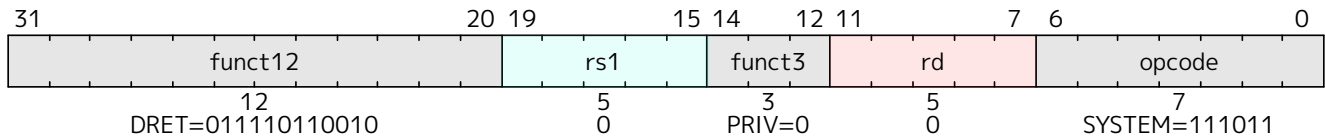
Description

DRET (RVY) returns from debug mode. It unseals dpc (RVY) and writes the result into pc.

> ℹ️ *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
```

11.4.2. Integrating Zyhybrid with Sdext

A new debug default data capability (dddc) CSR is added at the CSR number shown in Table 23.

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
```

> ℹ️ *There is no CHERI enable/disable bit for debug mode, so CHERI register and instruction access is always permitted in debug mode.*

Debug Default Data Capability CSR (dddc)

dddc is a debug mode accessible capability CSR. The address is shown in Table 23.

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.
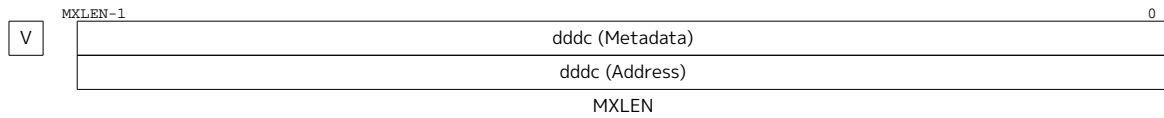
Figure 41. Debug default data capability

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 76, dddc is a data pointer, so it does not need to be able to hold all possible invalid addresses (see Invalid address conversion).

11.4.3. "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 66.

Debug triggers are higher priority than CHERI exceptions to allow debug.

Table 66. Synchronous exception priority (including triggers) in decreasing priority order. Entries added in RVY are in bold

Priority	Exc. Code	Description	Trigger
Highest	3 3 3 3		etrigger icount itrigger mcontrol/mcontrol6 after (on previous instruction)
	3	Instruction address breakpoint	mcontrol/mcontrol6 execute address before
	32	**Prior to instruction address translation:** **CHERI Instruction Access Fault due to pc checks (tag, execute permission, and bounds)**	
	12, 1	During instruction address translation: First encountered page fault or access fault	
	1	With physical address for instruction: Instruction access fault	
	3		mcontrol/mcontrol6 execute data before

Priority	Exc. Code	Description	Trigger
	2 0 8,9,11 3	Illegal instruction Instruction address misaligned Environment call Environment break	
	3	Load/store/AMO address breakpoint	mcontrol/mcontrol6 load/store address before
	3		mcontrol/mcontrol6 store data before
	33,34	**Prior to address translation for an explicit memory access:** **Load/store/AMO capability address misaligned** **CHERI Load Access Fault, CHERI Store/AMO Access Fault due to capability checks (tag, sealed, permissions and bounds)**	
	4,6	Optionally: Load/store/AMO address misaligned	
	13, 15, 5, 7	During address translation for an explicit memory access: First encountered **CHERI Load Page Fault, 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	
	13	If not higher priority: **CHERI Load Page Fault** [3]	
Lowest	3		mcontrol/mcontrol6 load data before

> **i** *See the notes beneath Synchronous exception priority in decreasing priority order for details about CHERI load page fault priority.*

Chapter 12. Pointer Masking (Ssnpm, Smnpm, Smmpm, Sspm, Supm) (RV64Y)

> **i**   *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.

> **i**   *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 decode, which are 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 be actually be addressed with pointer masking enabled (such as one that covers 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.

> **i**   *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 ADDIY/ADDY/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)) are not affected by pointer masking.

Chapter 13. "Svucrg" Extension, Version 1.0 for RV64Y

⚠️   *This chapter will appear in the priv spec. Exact location TBD.*

ℹ️   *Sv32 (for RV32) does not have any spare PTE bits, and so no features from this chapter can be implemented.*

The Svucrg extension adds the ability to perform capability revocation of user mode pages (see Section 13.1) by adding the PTE.CRG, PTE.CD and sstatus.UCRG bits as described below.

ℹ️   *Svucrg depends on Supervisor-Level ISA and RV64Y also being supported.*

ℹ️   *Svucrg is strongly recommended but not mandatory as a future version of this specification may specify an improved method.*

ℹ️   *There is no explicit mechanism for enabling or disabling Svucrg.*

ℹ️   *If software ignores the new PTE bits then there is no change in functionality unless capabilities are accessed.*

ℹ️   *A future version of this specification may include kernel revocation which may require an sstatus.SCRG bit.*

The minimum level of PTE support is to set CRW and CD to 1 in all PTEs intended for storing capabilities (i.e., private anonymous mappings) and leave sstatus.UCRG and CRG in all PTEs set to 0, which will allow capabilities with their tags set to be loaded and stored successfully.

ℹ️   *Hardware initiated memory accesses from the page-table walker are not checked by a capability.*

13.1. Capability Revocation

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

13.2. Extending the Page Table Entry Format

Two new bits, Capability Read Generation (CRG) and Capability Dirty (CD), are added to leaf PTEs in Sv39, Sv48 and Sv57 as shown in Figure 42, Figure 43 and Figure 44 respectively. For non-leaf PTEs these bits remain reserved and must be cleared by software for forward compatibility, or else a page-fault exception is raised. Additionally, if the hypervisor extension is enabled these bits remain reserved for leaf and non-leaf PTEs used in guest address translation.

63	62 61	60 59	58	57	56	55 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
N	PBMT	Reserved	CRW	CRG	CD	Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
1	2	2	1	1	1	2	26	9	9	2	1	1	1	1	1	1	1	1

Figure 42. Sv39 page table entry

63	62 61	60 59	58	57	56 55	54 53 10 9	8 7	6	5	4	3	2	1	0
N	PBMT	Reserved	CRW	CRG	CD	Reserved PPN	RSW	D	A	G	U	X	W	R	V
1	2	2	1	1	1	2 44	2	1	1	1	1	1	1	1	1

53 37	36 28	27 19	18 10
PPN[3]	PPN[2]	PPN[1]	PPN[0]
17	9	9	9

Figure 43. Sv48 page table entry

63	62 61	60 59	58	57	56 55	54 53 10 9	8 7	6	5	4	3	2	1	0
N	PBMT	Reserved	CRW	CRG	CD	Reserved PPN	RSW	D	A	G	U	X	W	R	V
1	2	2	1	1	1	2 44	2	1	1	1	1	1	1	1	1

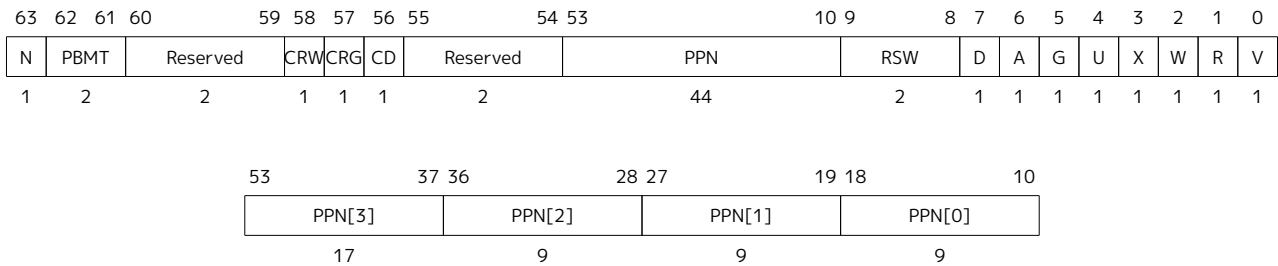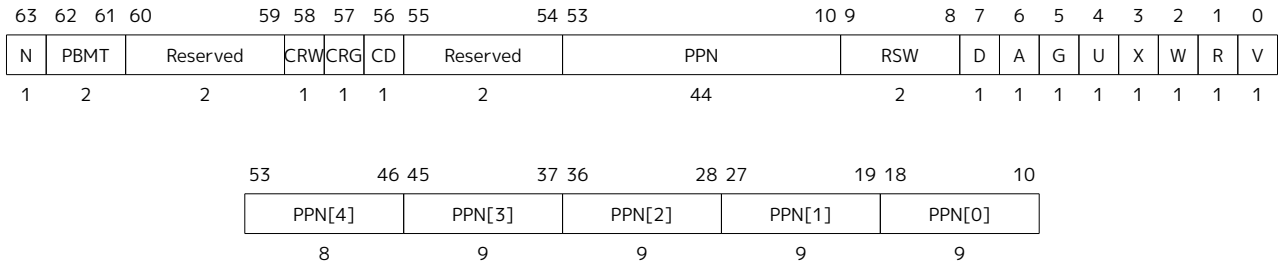| 53 46	45 37	36 28	27 19	18 10
PPN[4]	PPN[3]	PPN[2]	PPN[1]	PPN[0]
8	9	9	9	9

Figure 44. Sv57 page table entry

> ℹ The behavior in this section isn't relevant if:

1. The authorizing capability doesn't have C-permission, for loads, stores and AMO.

2. Any extension-specific mediation has already cleared the stored tag, for stores and AMOs.

The CRW bit (defined by Supervisor-Level ISA for Virtual Memory (RV64Y)) indicates whether reading or writing capabilities with the capability tag set to the virtual page is permitted. When the CRW bit is set, capabilities are written as usual, and capability reads are controlled by the CRG bit.

> ℹ *The capability tag bit of the stored capability is checked after it is potentially cleared due to lack of C-permission.*

If the CRW bit is clear, the "no capability state", then the existing rules from Chapter 11 are followed:

- When a capability load or AMO instruction is executed, the capability tag bit of the loaded capability is cleared before it is written to the destination register.

- When a capability store or AMO instruction is executed and the capability tag bit of the capability being written is set, the implementation raises a CHERI Store/AMO Page Fault.

When the CRW bit is set, the "capability state", then the behavior of capability loads is controlled by CRG and the behavior of capability stores is controlled by CD.

When CRW is set, the CRG bit indicates the current generation of the virtual memory page with regards to the ongoing capability revocation cycle. The implementation raises CHERI Load Page Fault when a capability load or AMO instruction is executed and:

- the authorizing capability grants C-permission,
- the virtual page's CRG bit does not equal sstatus.UCRG, and
- the virtual page's PTE.U is set.

When the *Svucrglct* extension is implemented, the CHERI Load Page Fault is raised when capability load or AMO instruction satisfies the conditions above and the capability read from memory has its capability tag set.

Svucrglct implies Svucrg.

When CRW is set, the CD bit indicates that a capability was stored to the virtual page since the last time the CD bit was cleared. When a capability store or AMO instruction is executed, the capability tag bit of the capability being written is set and the CD bit is clear, two schemes are permitted (also see Section 13.3):

- The same behavior as when CRW is clear, allowing software interpretation of this state.
- The implementation sets the CD bit. The PTE update behaves in the same way as the D bit update described by the *Svadu* extension.

When CRW, CD and CRG are all clear, the implementation is required to clear loaded tags and raise CHERI Store/AMO Page Fault when the stored capability tag is set.

> **i** *Other CD and CRG combinations when CRW=0 are reserved for future extensions. The reserved PTE states behave as the CRW=0, CD=0, CRG=0, unless a future extension defines an alternative function.*

Table 67. Summary of Load CRW and CRG behavior in the PTEs

PTE.CRW	PTE.CD	PTE.CRG	PTE.U	Load/AMO
0	0	0	X	Clear loaded tag
0	0	1	X	Reserved
0	1	X	X	Reserved
1	X	≠ sstatus.UCRG	1	CHERI Load Page Fault, or CHERI Load Page Fault if tag is set for *Svucrglct*[1]
1	X	= sstatus.UCRG	1	Normal operation
1	X	X	0	Normal operation[2]

[1] The choice here is whether to take data dependent exceptions on load data for loads or atomic operations. The default is to take the trap without checking the value of the loaded capability tag. Taking a trap when the capability tag is not set will introduce additional traps during revocation sweeps. If *Svucrglct* is implemented then the trap is only taken if the loaded capability tag is set, to reduce software overhead from revocation sweeps. Checking the loaded tag affects the exception priority, see Synchronous exception priority in decreasing priority order.

> **i** *Svucrglct is an optimization for software, and as such implementations are allowed to conservatively fault under certain conditions even if the capability tag is not set.*

> **i** *Implementations which already take synchronous traps on loaded data, such as ECC faults, should implement Svucrglct instead of Svucrg.*

Svucrglct implies *Svucrg*.

[2] A future version of this specification may check an SCRG bit in sstatus for kernel revocation.

Table 68. Summary of Store CRW and CD behavior in the PTEs

PTE.CRW	PTE.CD	PTE.CRG	Store/AMO
0	0	0	CHERI Store/AMO Page Fault if stored tag is set
0	1	X	Reserved
1	0	X	CHERI Store/AMO Page Fault if stored tag is set, or hardware CD update
1	1	X	Normal operation

13.3. Enabling Software or Hardware PTE updates

The decision about whether to take exceptions on capability stores with the capability tag set to a page with PTE.CRW=1 and PTE.CD=0 is determined by whether the *Svade* and *Svadu* extensions are enabled. These cause PTE Accessed and Dirty updates to be done in software, via the exception handler, or by a hardware mechanism respectively.

- If only *Svade* is implemented, or enabled through henvcfg.ADUE or menvcfg.ADUE, then take a CHERI Store/AMO Page Fault.
- If only *Svadu* is implemented, or enabled through henvcfg.ADUE or menvcfg.ADUE, then do the hardware update of setting PTE.CD=1 as described in Section 13.2.

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

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

When V=1 vsstatus.UCRG is in effect.

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

63	62	61	60 — 48
SD	WPRI	UCRG	WPRI

47 — 43	42	41	40	39	38	37	36	35 — 34	33 — 32
WPRI	MDT	MPELP	WPRI	MPV	GVA	MBE	SBE	SXL[1:0]	UXL[1:0]

31 — 25	24	23	22	21	20	19	18	17	16
WPRI	SDT	SPELP	TSR	TW	TVM	MXR	SUM	MPRV	XS[1:0]

15	14 — 13	12 — 11	10 — 9	8	7	6	5	4	3	2	1	0
XS[1:0]	FS[1:0]	MPP[1:0]	VS[1:0]	SPP	MPIE	UBE	SPIE	WPRI	MIE	WPRI	SIE	WPRI

*Figure 45. Machine-mode status (**mstatus**) register for RV64Y*

63	62	61	60 — 48
SD	WPRI	UCRG	WPRI

47 — 34	33 — 32
WPRI	UXL[1:0]

31 — 25	24	23	22 — 20	19	18	17	16
WPRI	SDT	SPELP	WPRI	MXR	SUM	WPRI	XS[1:0]

15	14 — 13	12 — 11	10 — 9	8	7	6	5	4 — 2	1	0
XS[1:0]	FS[1:0]	WPRI	VS[1:0]	SPP	WPRI	UBE	SPIE	WPRI	SIE	WPRI

*Figure 46. Supervisor-mode status (**sstatus**) register when SXLEN=64*

63	62	61	60 — 48
SD	WPRI	UCRG	WPRI

47 — 34	33 — 32
WPRI	UXL[1:0]

31 — 20	19	18	17	16
WPRI	SUM	MXR	WPRI	XS[1:0]

15	14 — 13	12 — 11	10 — 9	8	7	6	5	4 — 2	1	0
XS[1:0]	FS[1:0]	WPRI	VS[1:0]	SPP	WPRI	UBE	SPIE	WPRI	SIE	WPRI

*Figure 47. Virtual Supervisor-mode status (**vsstatus**) register when VSXLEN=64*

Chapter 14. Hypervisor "H" Extension (RVY)

ℹ️   *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, is that address CSRs added for hypervisors are extended to YLEN size. The remainder of this chapter describes these changes in detail.

14.1. 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 determines controls the endianness of explicit memory accesses from VS-mode and implicit memory accesses to VS-level memory management data structures. VSBE=0 indicates little endian and VSBE=1 is big endian. VSBE must be read-only and equal to MBE when the implementation only supports RVY.

14.2. 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 48.
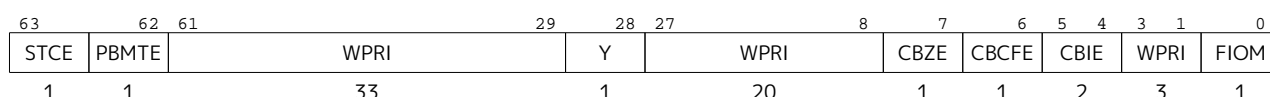
63	62	61	29	28	27	8	7	6	5	4	3	1	0
STCE	PBMTE		WPRI		Y	WPRI		CBZE	CBCFE		CBIE	WPRI	FIOM
1	1		33		1	20		1	1		2	3	1

*Figure 48. 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 10.

The reset value is 0.

14.3. Hypervisor Exception Delegation Register (hedeleg)

Bits 32,33,34,35,36 of hedeleg refer to a valid CHERI exceptions and so can be used to delegate CHERI exceptions to supervisor mode.

14.4. 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.

14.5. Virtual Supervisor Trap Vector Base Address Capability Register (vstvec)

The `vstvec` register is extended to hold a capability. Its reset value is a Root Executable capability.
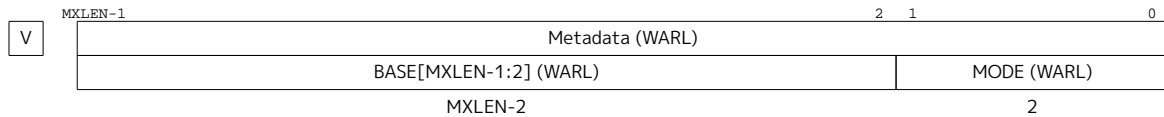
Figure 49. Virtual supervisor trap vector base address capability register

The handling of vstvec (RVY) is otherwise identical to mtvec (RVY), but in virtual supervisor mode.

14.6. Virtual Supervisor Scratch Register (vsscratch)

The `vsscratch` 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.

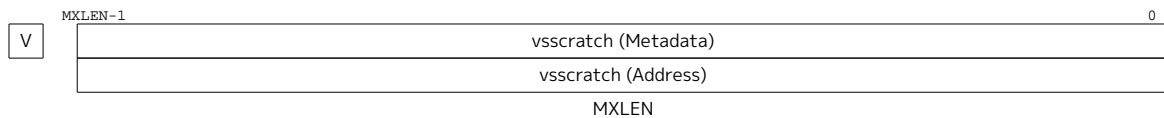It is not WARL, all capability fields must be implemented.

Figure 50. Virtual supervisor scratch capability register

14.7. Virtual Supervisor Exception Program Counter Capability (vsepc)

The `vsepc` register is extended to hold a capability. Its reset value is a Root Executable capability.

As shown in Table 76, 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.
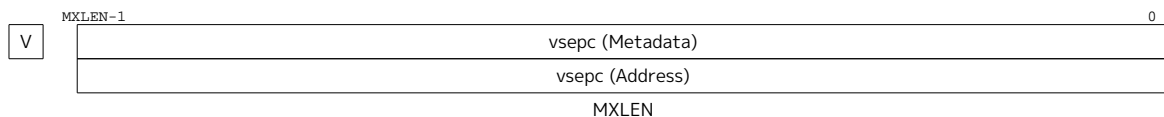

Figure 51. Virtual supervisor exception program counter capability

14.8. 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 which are delegated to VS-mode.

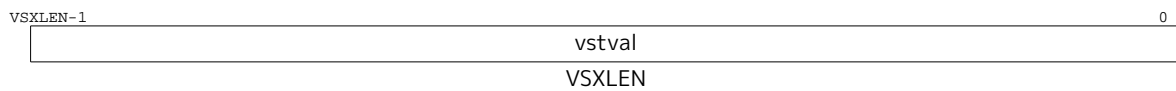| VSXLEN−1	0
vstval	
VSXLEN	

Figure 52. Virtual supervisor trap value register

14.9. Virtual Supervisor Thread Identifier Capability (vstidc)

The vstidc register is used to identify the current software thread in virtual supervisor mode. It is used to identify the current software thread in As 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. On reset the capability tag of vstidc will be set to 0 and the remainder of the data is UNSPECIFIED.
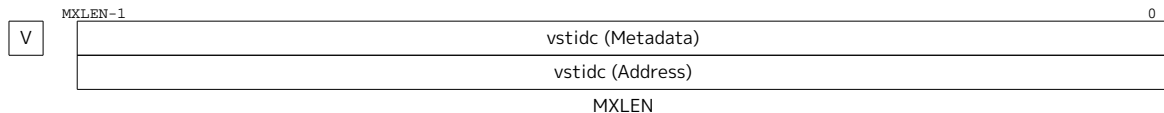
V	MXLEN−1	0
	vstidc (Metadata)	
	vstidc (Address)	
	MXLEN	

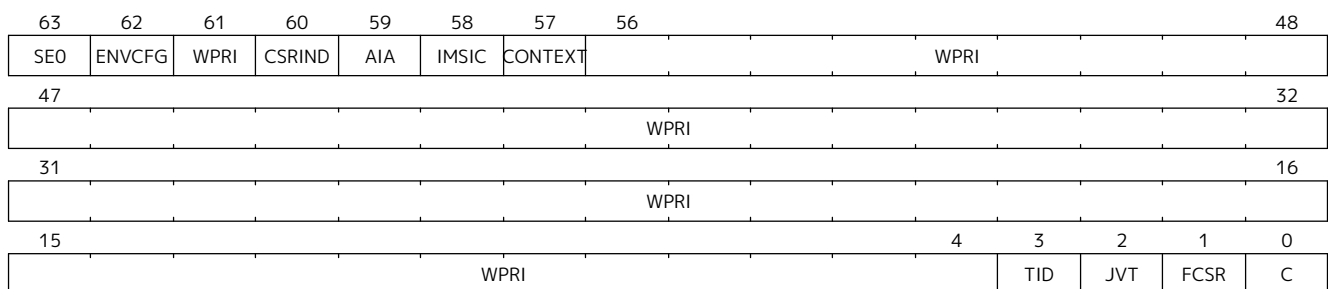*Figure 53. Virtual supervisor thread identifier capability register*

14.10. Virtual Supervisor CHERI Capability Encoding (vsycfg)

The vsycfg register is used to identify which CHERI capability encoding is used by the platform. The capability encoding both determines the in-memory representation of a CHERI capability and entails a set of extensions present atop RVY. The CSR exists in the *writable* namespace and all bits are defined to be WARL, but, absent any future extensions, it is expected that each implementation has exactly one legal value. For the meaning of the individual fields, refer to the uycfg register in the unprivileged specification.

XLEN−1	9 8	0
Reserved	Features	Base Encoding
MXLEN-16	8	8
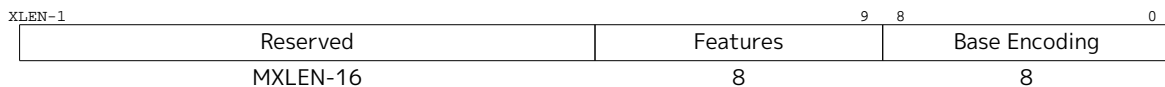
Figure 54. Virtual Supervisor CHERI capability encoding CSR (`vsycfg`) format

14.11. "Smstateen/Ssstateen" Integration

The new TID bit controls access to the vstidc CSR.

63	62	61	60	59	58	57	56		48
SE0	ENVCFG	WPRI	CSRIND	AIA	IMSIC	CONTEXT		WPRI	

47		32
WPRI		

31		16
WPRI		

15		4	3	2	1	0
WPRI			TID	JVT	FCSR	C

Figure 55. Hypervisor State Enable 0 Register (`hstateen0`)

14.12. 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 2.6. 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.
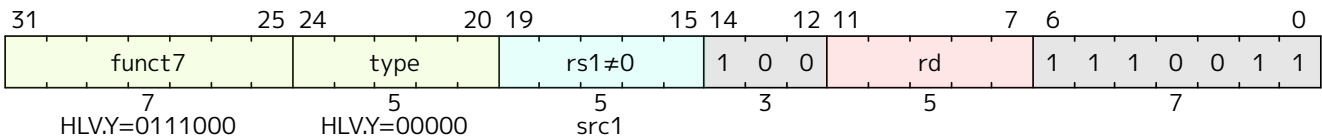
14.12.1. HLV.Y

Synopsis

Hypervisor virtual-machine load capability

Mnemonic

```
hlv.y rd, rs1
```

Encoding

> **ℹ** *Any instance of this instruction with a* `rs1`=`x0` *will raise an exception, as* `x0` *is defined to always hold a* NULL *capability. As such, the encodings with a* `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.

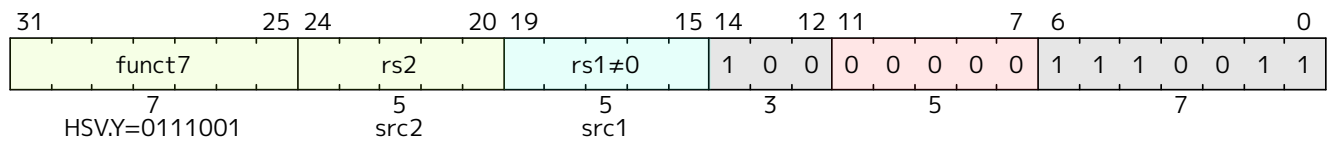**Prerequisites**

RVY, H

Operation

TBD

14.12.2. HSV.Y

Synopsis

Hypervisor virtual-machine store capability

Mnemonic

```
hsv.y rs2, rs1
```

Encoding

31 funct7 25	24 rs2 20	19 rs1≠0 15	14 12	11 7	6 0
funct7	rs2	rs1≠0	1 0 0	0 0 0 0 0	1 1 1 0 0 1 1
7	5	5	3	5	7
HSV.Y=0111001	src2	src1			

> ℹ️ *Any instance of this instruction with a* `rs1`=`x0` *will raise an exception, as* `x0` *is defined to always hold a* NULL *capability. As such, the encodings with a* `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

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 69. 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 70. 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 71. 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 72. 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 73. 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 74. 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 75. 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 76. 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 76. 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.

Machine-Level ISA (RVY) and Supervisor-Level ISA (RVY) extend the CSRs listed in Table 77, Table 78, Table 79, Table 80 and Table 81 from the base RISC-V ISA and its extensions.

> **ℹ** *If Zyhybrid is supported then the CHERI Execution Mode determines whether YLEN or XLEN bits are returned (see CSRRW (RVY)).*

Table 77. 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 78. 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

> **ℹ** *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 79. 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 80. 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

RVY CSR	Address	Prerequisites	Permissions	Description
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 81. 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.

ℹ️ *Clearing capability tags in memory does not necessarily require updating the associated data.*

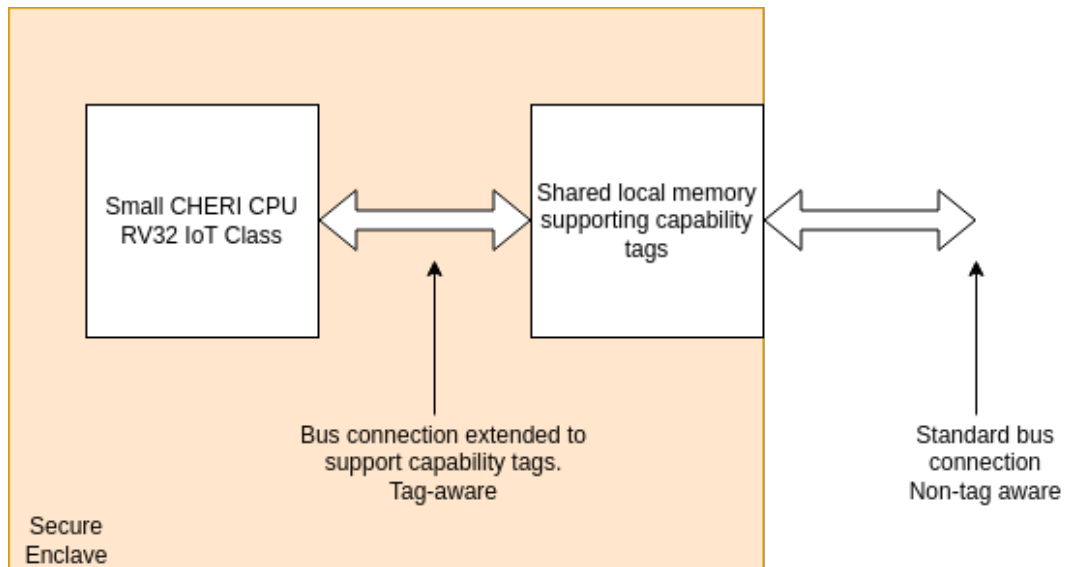## B.3.1. Small CHERI system example

Figure 56. 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.
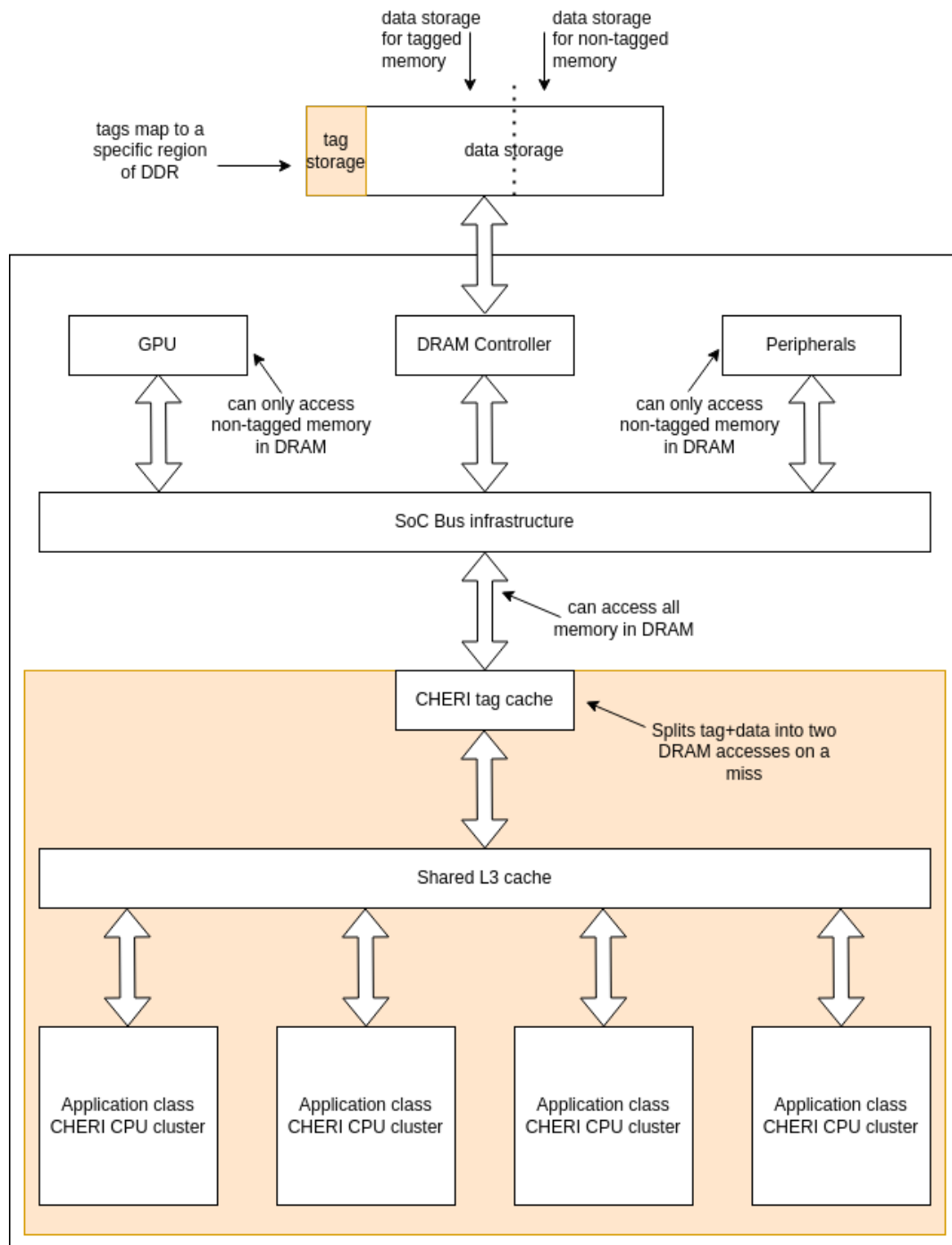
B.3.2. Large CHERI system example

Figure 57. 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 only accessible 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 tagged memory data storage region. These constraints will be part of the design of the network-on-chip. It *is* possible for the GPU and peripherals to read the capability tagged 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).

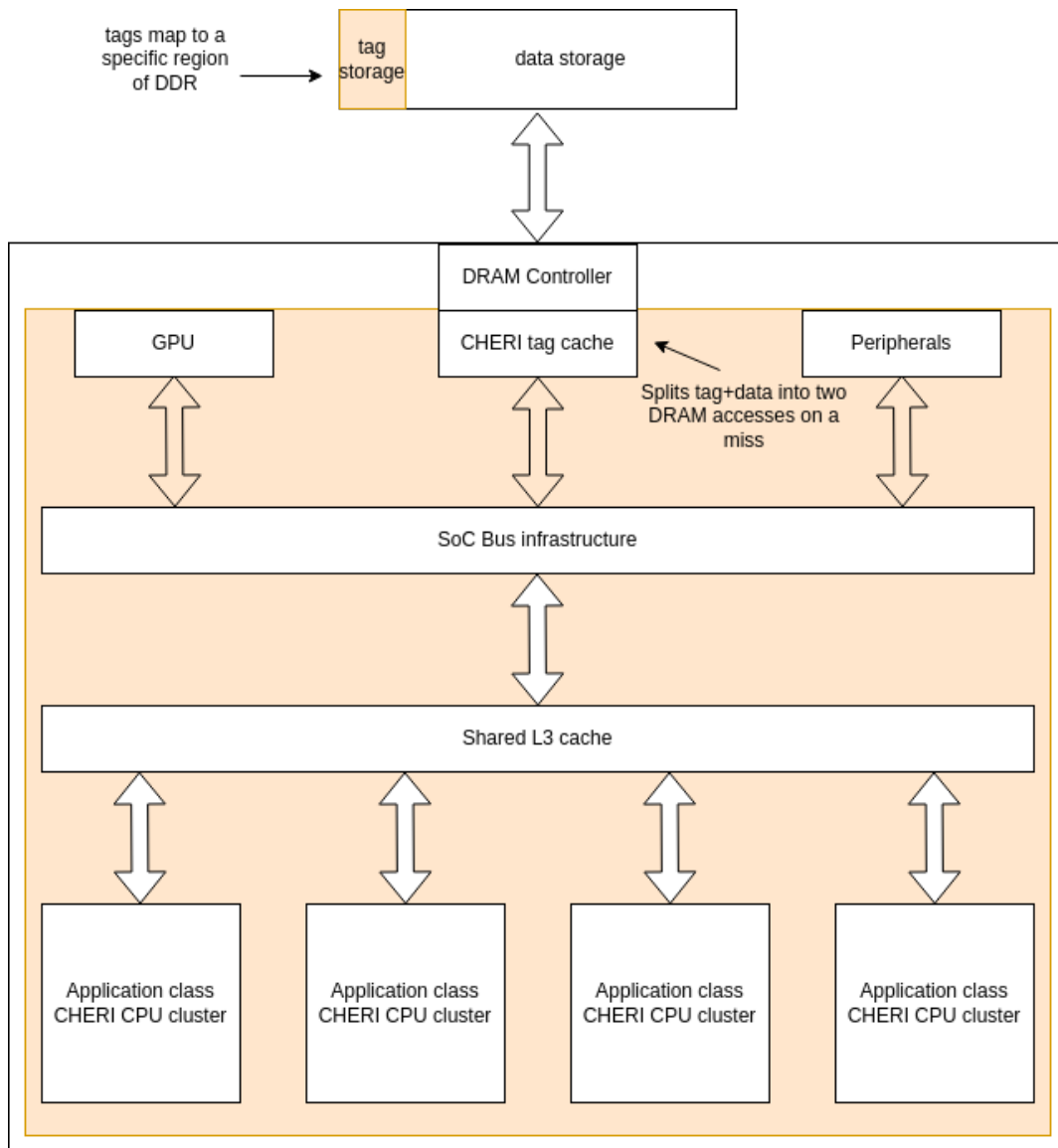### B.3.3. Large CHERI pure-capability system example

Figure 58. 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 57.

There is likely to be a performance difference between the two systems. The main motivation for Figure 57 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).

MXLEN-1	0
mscratch

MXLEN

Figure 59. Machine-mode scratch register

Machine Cause (**mcause**) Register

Synchronous exception priority in decreasing priority order

Machine cause (**mcause**) register values after trap

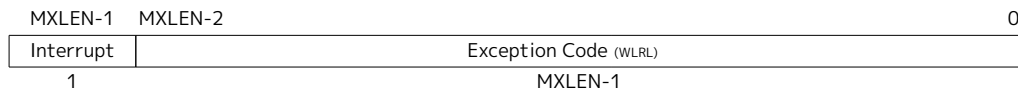> See mcause in Chapter *Machine-Level ISA, Version 1.13* in (RISC-V, 2023).

MXLEN-1	MXLEN-2	0
Interrupt	Exception Code (WLRL)	
1	MXLEN-1	

Figure 60. 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).

MXLEN-1	2 1	0
BASE [MXLEN-1:2] (WARL)	MODE (WARL)	
MXLEN-2	2	

Figure 61. 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).
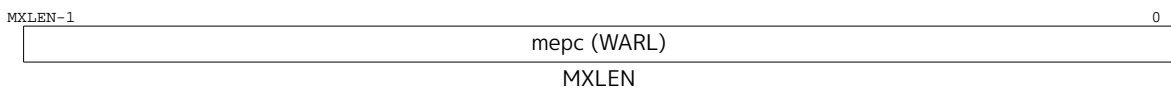
MXLEN-1	0
mepc (WARL)

MXLEN

Figure 62. 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).
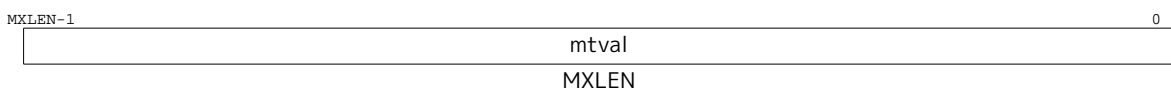
MXLEN-1	0
mtval

MXLEN

Figure 63. 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 also 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.

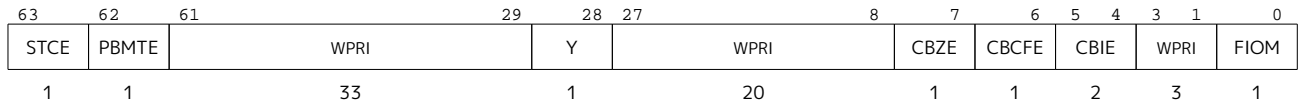| 63	62	61 29	28	27 8	7	6	5 4	3 1	0
STCE	PBMTE	WPRI	Y	WPRI	CBZE	CBCFE	CBIE	WPRI	FIOM
1	1	33	1	20	1	1	2	3	1

Figure 64. 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

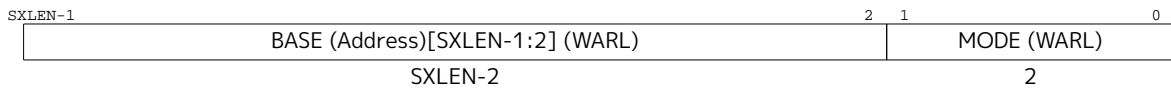See `stvec` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

SXLEN-1 2	1 0
BASE (Address)[SXLEN-1:2] (WARL)	MODE (WARL)
SXLEN-2	2

Figure 65. 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).

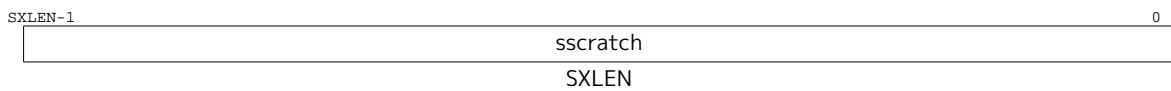| SXLEN-1   0 |
|--|
| sscratch |
| SXLEN |

Figure 66. Supervisor-mode scratch register

Supervisor Exception Program Counter (`sepc`) Register

See `sepc` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

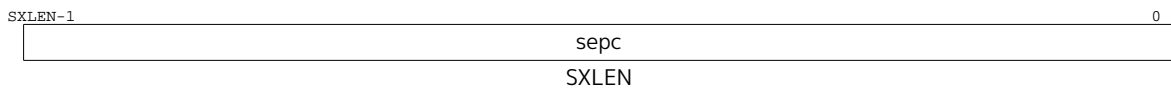| SXLEN-1   0 |
|--|
| sepc |
| SXLEN |

Figure 67. Supervisor exception program counter register

Supervisor Trap Value (`stval`) Register

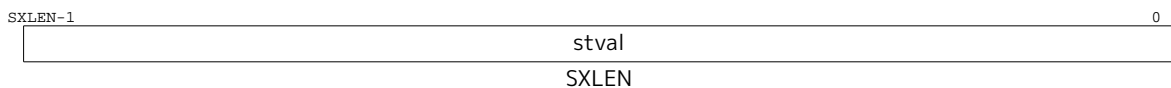See `stval` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

SXLEN-1   0
stval
SXLEN

Figure 68. Supervisor trap value register

Supervisor Cause (`scause`) Register

.Supervisor cause (`scause`) register values after trap

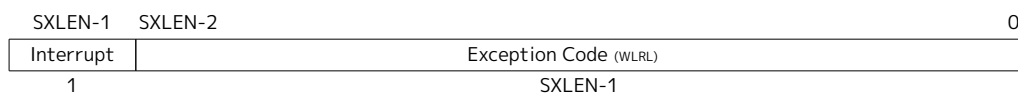See `scause` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

SXLEN-1	SXLEN-2 0
Interrupt	Exception Code (WLRL)
1	SXLEN-1

Figure 69. Supervisor Cause (`scause`*) register.*

Supervisor Environment Configuration (`senvcfg`) Register

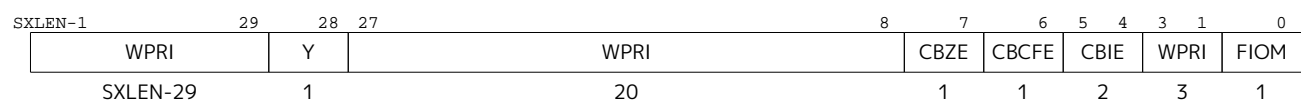See `senvcfg` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023). Zyhybrid adds a new Y bit.

SXLEN−1 29	28	27 8	7	6	5 4	3 1	0
WPRI	Y	WPRI	CBZE	CBCFE	CBIE	WPRI	FIOM
SXLEN-29	1	20	1	1	2	3	1

Figure 70. Supervisor environment configuration register (senvcfg)

Supervisor Status (`sstatus`) Register

See `sstatus` in Chapter *Supervisor-Level ISA, Version 1.13* in (RISC-V, 2023).

Bibliography

Efficient Tagged Memory. (2017). www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201711-iccd2017-efficient-tags.pdf

RISC-V. (2023). *RISC-V Unprivileged Specification*. github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-056b6ff-2023-10-02/unpriv-isa-asciidoc.pdf

RISC-V. (2023). *RISC-V 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