The Capstone-RISC-V Instruction Set Reference

Table of Contents

Version Information: Version 1.0 +Caplifive

1. Introduction

Capstone is a CPU instruction set architecture (ISA) that creates a single unified architectural abstraction for achieving multiple security goals, thus liberating software developers from the burden of working with the distinct fundamental primitives exposed by numerous security extensions that often do not interoperate easily.

1.1. Supported Properties

The ultimate goal of Capstone is to provide a unified architectural abstraction for multiple security goals. This goal requires Capstone to support the following properties.

> **Exclusive access**
>
> Software should be guaranteed exclusive access to certain memory regions if needed. This is in spite of the existence of software traditionally entitled to higher privileges such as the OS kernel and the hypervisor.
>
> **Revocable delegation**
>
> Software components should be able to delegate authority to other components in a revocable manner. For example, after an untrusted library function has been granted access to a memory region, the caller should be able to revoke this access.
>
> **Dynamically extensible hierarchy**
>
> The hierarchy of authority should be dynamically extensible, rather than predefined by the architecture such as hypervisor-kernel-user found in traditional platforms. This makes it possible to use the same set of abstractions for memory isolation and memory sharing regardless of where a software component lies in the hierarchy.
>
> **Safe context switching**
>
> A mechanism that protects the confidentiality and integrity of the execution context of software during control flow transfers across security domain boundaries, including asynchronous ones such as those for interrupt and exception handling, should be provided.

1.2. Major Design Elements

The Capstone architecture design is based on the idea of capabilities, which are unforgeable tokens that represent authority to perform memory accesses and control flow transfers, among other operations. Capstone extends the basic capability model with new capability types including the

following.

> **Linear capabilities**
>
> Linear capabilities are guaranteed not to alias with other capabilities that both grant memory access and are in architecturally visible locations (i.e., their actual contents might affect the execution of the whole system). Operations on linear capabilities maintain this property. For example, instructions can only move, but not copy, linear capabilities between general-purpose registers. They can hence enable safe exclusive access to memory regions. Capabilities that do not have this property are called *non-linear* capabilities.
>
> **Revocation capabilities**
>
> Revocation capabilities cannot be used to perform memory accesses or control flow transfers. Instead, they convey the authority to revoke other capabilities. Each revocation capability is derived from a linear capability and can later be used to revoke (i.e., invalidate) capabilities derived from it. This mechanism enables revocable and arbitrarily extensible chains of delegation of authority.
>
> **Uninitialised capabilities**
>
> Uninitialised capabilities convey write-only authority to memory. They can be turned into linear capabilities after the memory region has been "initialised", i.e., when the whole memory region has been overwritten with fresh data. Uninitialised capabilities enable safe initialisation of memory regions and prevent secret leakage without incurring extra performance overhead.

1.3. Capstone-RISC-V ISA Overview

While Capstone does not assume any specific modern ISA, we choose to propose a Capstone variant to RISC-V due to its open nature and the availability of toolchains and simulators.

The Capstone-RISC-V ISA is an RV64IZicsr variant that makes the following types of changes to the base architecture:

- Each general-purpose register is extended to 129 bits to accommodate 128-bit capabilities.
- Part of the machine state is extended and new instructions are added to support it.
- New instructions for manipulating capabilities are added.
- New instructions for memory accesses using capabilities are added.
- New instructions for control flow transfers using capabilities are added.

- Semantics of some existing instructions are adjusted to support capabilities.
- Semantics of interrupts and exceptions are adjusted to support capabilities.

1.4. Caplifive Extension

This document describes a version of Capstone-RISC-V with the *Caplifive extension*, which enables Capstone to support guest machines with traditional access control abstractions. This is achieved through the idea of *caplification*: it allows exposing capability-backed physical memory regions to lower privilege levels through a modified PMP (Physical Memory Protection) structure which requires valid capabilities in its entries instead of arbitrary ranges and permissions.

Compared to vanilla Capstone-RISC-V, Capstone-RISC-V with Caplifive defines domain *internal structures* which can consist of privilege levels found in traditional architectures. The highest privilege level operates using capabilities in the same way as in vanilla Capstone-RISC-V, whereas lower privilege levels are exposed to an abstraction compatible with that on traditional architectures. In particular, Caplifive allows software in lower privilege levels to access memory using raw addresses and potentially through virtual memory, as long as the access is allowed according to the capabilities configured in the modified PMP structure. Note that such details are internal to each domain itself. Across domains, software interacts using capabilities in the same way as in vanilla Capstone-RISC-V.

The Caplifive extension additionally includes the following changes:

- Introduction of C-mode and a mechanism to enter it from M-mode.
- Mechanism to post V-interrupts to a domain.
- Mechanism to expose capabilities in C-mode to lower privilege levels.

1.5. Assembly Mnemonics

Each Capstone-RISC-V instruction is given a mnemonic prefixed with `CS.`. In contexts where it is clear we are discussing Capstone-RISC-V instructions, we will omit the `CS.` prefix for brevity.

In assembly code, the list of operands to an instruction is supplied following the instruction mnemonic, with the operands separated by commas, in the order of `rd`, `rs1`, `rs2`, `imm` for any operand the instruction expects.

1.6. Notations

When specifying the semantics of instructions, we use the following notations to represent the type of each operand:

I

Integer register.

C

Capability register.

S

Sign-extended immediate.

Z

Zero-extended immediate.

1.7. Bibliography

The initial motivation, design, evaluation, and analysis of Capstone have been discussed in the following paper:

- Capstone: A Capability-based Foundation for Trustless Secure Memory Access by Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, Prateek Saxena. In *Proceedings of the 32nd USENIX Security Symposium*. Anaheim, CA, USA. August 2023.

2. Programming Model

The Capstone-RISC-V ISA has extended part of the machine state, including both some registers and the memory, to enable the storage and handling of capabilities.

2.1. Modes

At any point in time, a hart operates in one of the following modes (mode ID in parentheses):

- U-mode (0)
- S-mode (1)
- M-mode (3)
- C-mode (3)

U-, S-, and M-modes already exist in the base RV64IZicsr ISA. When operating in those modes, the hart follows the same behaviours as defined in RV64IZicsr, e.g., using physical addresses for memory accesses in M-mode and virtual addresses for memory accesses if the virtual memory is enabled in S-mode and U-mode.

C-mode (capability mode) is an extra privilege level introduced in the Caplifive extension. It replaces M-mode when `CAPSTONE_EN = 1` (`CAPSTONE_EN` is a single-bit hart-local state). When operating in C-mode, the hart is required to use capabilities for accessing the memory (including fetching instructions). C-mode exposes the same set of interfaces to S-mode and U-mode as M-mode does. For example, C-mode can handle undelegated interrupts and exceptions from S-mode and U-mode. Whether `CAPSTONE_EN = 1` is therefore transparent to S-mode and U-mode, which can interact with C-mode in the same way as with M-mode.

2.2. Domains

When `CAPSTONE_EN = 1`, the run-time state of the system is organised in individual compartments called *domains*. Each domain includes the execution context of a logical thread. A hart at any point in time executes in exactly one domain.

When a domain is running on a hart, the following events are notable:

- Execution of CALL and RETURN instructions: Those instructions receive arguments that specify another domain and trigger a synchronous domain switch.
- Exceptions: An exception is triggered by the execution of a specific instruction in the current domain. Since the event is local to the domain itself, it does not trigger a domain switch. Instead, exceptions are handled within the same domain that triggers them.
- Interrupts: Unlike exceptions, interrupts are triggered by external factors rather than by specific instructions executed in the domain. We distinguish two types of interrupts which are handled differently:
    - H-interrupts (hardware interrupts): Those are interrupts the hart receives from interrupt controllers (either hart-local or platform-wide). Since such interrupts are not intended for

specific domains (interrupt controllers and the I/O devices behind them are not aware of domains), a switch to a dedicated interrupt handler domain is required to handle them.

- V-interrupts (virtual interrupts): Those are asynchronous events sent between domains. Unlike H-interrupts, a V-interrupt targets a specific domain and therefore does not trigger a domain switch, but is instead handled within the domain itself.

We call the first domain that runs on a hart after its reset the *genesis domain* of the hart.

2.3. Capabilities

2.3.1. Width

The width of a capability is 128 bits. We represent this as `CLEN = 128` and `CLENBYTES = 16`. Note that this does not affect the width of a raw address, which is `XLEN = 64` bits, or equivalently, `XLENBYTES = 8` bytes, same as in RV64IZicsr.

2.3.2. Fields

Each capability has the following architecturally-visible fields:

Table 1. Fields in a capability

Name	Range	Description
`valid`	`0..1`	Whether the capability is valid: `0` = invalid, `1` = valid
`type`	`0..6`	The type of the capability: `0` = linear, `1` = non-linear, `2` = revocation, `3` = uninitialised, `4` = sealed, `5` = sealed-return
`cursor`	`0..2^XLEN-1`	Not applicable when `type = 4` (sealed). The memory address the capability points to (to be used for the next memory access)
`base`	`0..2^XLEN-1`	The base memory address of the memory region associated with the capability
`end`	`0..2^XLEN-1`	Not applicable when `type = 4` (sealed) or `type = 5` (sealed-return). The end memory address of the memory region associated with the capability

Name	Range	Description
`perms`	`0..7`	Not applicable when `type = 4` (sealed) or `type = 5` (sealed-return). One-hot encoded permissions associated with the capability: `0` = no access, `1` = execute-only, `2` = write-only, `3` = write-execute, `4` = read-only, `5` = read-execute, `6` = read-write, `7` = read-write-execute
`async`	`0..1`	Only applicable when `type = 4` (sealed) or `type = 5` (sealed-return). How the capability is sealed: `0` = synchronously, `1` = asynchronously
`reg`	`0..31`	Only applicable when `type = 5` (sealed-return). The index of the general-purpose register to restore the capability to

The range of the `perms` field has a partial order `<=p` defined as follows:

```
<=p = {
    (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7),
    (1, 1), (1, 3), (1, 5), (1, 7),
    (2, 2), (2, 3), (2, 6), (2, 7),
    (3, 3), (3, 7),
    (4, 4), (4, 5), (4, 6), (4, 7),
    (5, 5), (5, 7),
    (6, 6), (6, 7),
    (7, 7)
}
```

We say a capability `c` *aliases* with a capability `d` if and only if the intersection between `[c.base, c.end]` and `[d.base, d.end]` is non-empty.

For two revocation capabilities `c` and `d` (i.e., `c.type = d.type = 2`), we say `c <t d` if and only if

- `c` aliases with `d`
- The creation of `c` was earlier than the creation of `d`

In addition to the above fields, an implementation also needs to maintain sufficient metadata to test the `<t` relation. It will be clear that for any pair of aliasing revocation capabilities, the order of their creations is well-defined.

▼ **Note: the implementation of `valid` field**

The `valid` field is involved in revocation, where it might be changed due to a revocation operation on a different capability. A performant implementation, therefore, may prefer not to maintain the `valid` field inline with the other fields.

▼ **Note: addition/compression to capability fields**

Implementations are free to maintain additional fields to capabilities, or compress the representation of the above fields, as long as each capability fits in `CLEN` bits.

It is not required to be able to represent capabilities with all combinations of field values in a compressed representation, as long as the following conditions are satisfied:

1. For load and store instructions that move a capability between a register and memory, the value of the capability is preserved.
2. The resulting capability values of any operation are not more powerful than when the same operation is performed on a Capstone-RISC-V implementation without compression.
    - More specifically, if an execution trace is valid (i.e., without exceptions) on the compressed implementation, then it must also be valid on the uncompressed implementation. For example, a trivial yet useless compression would be to store nothing and always return a capability with `valid = 0`.

For different types of capabilities, a specific subset of the fields is used. The table below summarises the fields used for each type of capabilities.

Table 2. Fields used for each type of capabilities

Type	`type`	`valid`	`cursor`	`base`	`end`	`perms`	`async`	`reg`
Linear	`0`	Yes	Yes	Yes	Yes	Yes	-	-
Non-linear	`1`	Yes	Yes	Yes	Yes	Yes	-	-
Revocation	`2`	Yes	Yes	Yes	Yes	Yes	-	-
Uninitialised	`3`	Yes	Yes	Yes	Yes	Yes	-	-
Sealed	`4`	Yes	-	Yes	-	-	Yes	-
Sealed-return	`5`	Yes		Yes	-	-	Yes	Yes

When the `async` field of a sealed-return capability is `0` (synchronous), some memory accesses are granted by this capability. The following table shows the memory accesses granted in such scenarios, where `size` is the size of the memory access in bytes.

NOTE In an earlier version a sealed-return capability has a cursor and grants direct

load/store accesses. This is no more the case.

In other scenarios and for other capability types without the `perms` field, no read/write/execute memory accesses are granted by the capability.

The following figure shows the overview of different types of capabilities in the Capstone-RISC-V ISA, and the operations that change the type of a capability.

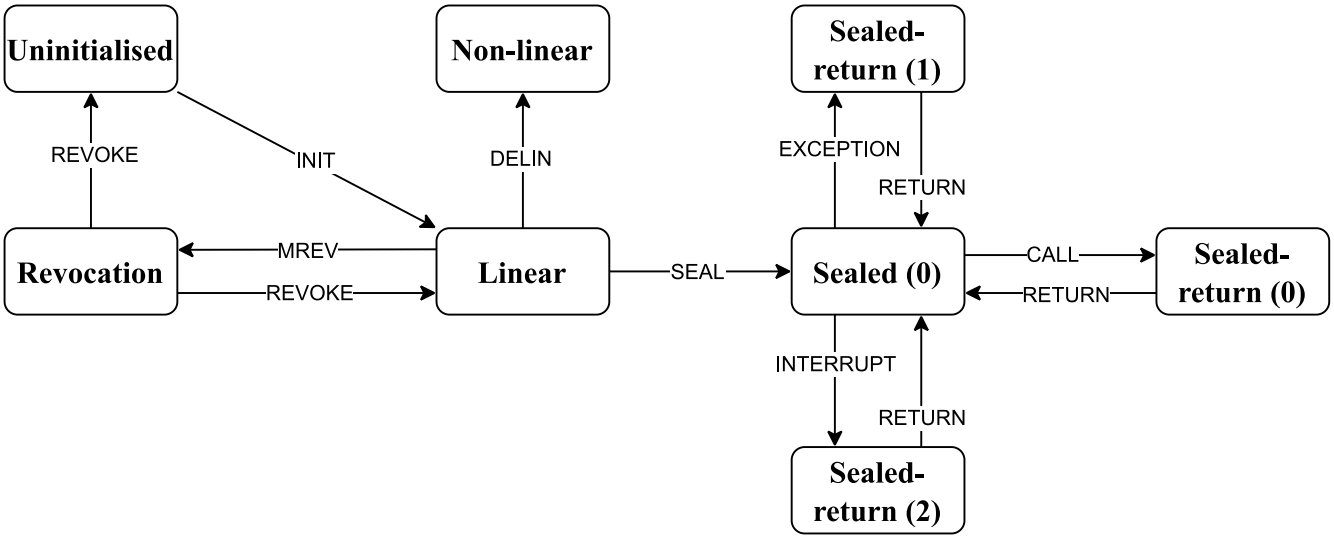

Figure 1. Overview of different types of capabilities in the Capstone-RISC-V ISA

2.4. Extension to General-Purpose Registers

The Capstone-RISC-V ISA extends each of the 32 general-purpose registers, so it contains either a capability or a raw `XLEN`-bit integer. The type of data contained in a register is maintained and type confusion is not allowed, except for `x0`/`c0` as discussed below. In assembly code, the type of data expected in a register operand is indicated by the alias used for the register, as summarised in the following table.

Index	`XLEN`-bit integer	Capability
0	`x0`/`zero`	`c0`/`cnull`
1	`x1`/`ra`	`c1`/`cra`
2	`x2`/`sp`	`c2`/`csp`
3	`x3`/`gp`	`c3`/`cgp`
4	`x4`/`tp`	`c4`/`ctp`
5	`x5`/`t0`	`c5`/`ct0`
6	`x6`/`t1`	`c6`/`ct1`
7	`x7`/`t2`	`c7`/`ct2`
8	`x8`/`s0`/`fp`	`c8`/`cs0`/`cfp`
9	`x9`/`s1`	`c9`/`cs1`
10	`x10`/`a0`	`c10`/`ca0`

Index	XLEN-bit integer	Capability
11	x11/a1	c11/ca1
12	x12/a2	c12/ca2
13	x13/a3	c13/ca3
14	x14/a4	c14/ca4
15	x15/a5	c15/ca5
16	x16/a6	c16/ca6
17	x17/a7	c17/ca7
18	x18/s2	c18/cs2
19	x19/s3	c19/cs3
20	x20/s4	c20/cs4
21	x21/s5	c21/cs5
22	x22/s6	c22/cs6
23	x23/s7	c23/cs7
24	x24/s8	c24/cs8
25	x25/s9	c25/cs9
26	x26/s10	c26/cs10
27	x27/s11	c27/cs11
28	x28/t3	c28/ct3
29	x29/t4	c29/ct4
30	x30/t5	c30/ct5
31	x31/t6	c31/ct6

x0/c0 is a read-only register that can be used both as an integer and as a capability, depending on the context. When used as an integer, it has the value 0. When used as a capability, it has the value { valid = 0, type = 0, cursor = 0, base = 0, end = 0, perms = 0 }. Any attempt to write to x0/c0 will be silently ignored (no exceptions are raised).

In this document, for i = 0, 1, ⋯, 31, we use x[i] to refer to the general-purpose register with index i.

2.5. Extension to Other Registers

2.5.1. Program Counter

The program counter (pc) is changed to contain a capability only.

During the instruction fetch stage, an exception is raised when any of the following conditions is met:

- `Instruction access fault (1)`
  - `pc.valid` is `0` (invalid).
  - `pc.type` is neither `0` (linear) nor `1` (non-linear).
  - `pc.perms` is not executable (i.e., `1 <=p pc.perms` does not hold).
  - `pc.cursor` is not in the range `[pc.base, pc.end - 3]`.
- `Instruction address misaligned (0)`
  - `pc.cursor` is not aligned to `4`.

If no exception is raised:

1. The instruction pointed to by `pc.cursor` is fetched and executed.
2. Set `pc.cursor` to `pc.cursor + 4` at the end of the instruction.

2.6. M-Mode Registers in C-Mode

The following M-mode CSRs are available in C-mode without any changes:

- `mstatus`
- `medeleg`
- `mideleg`
- `mip`
- `mie`
- `mcause`
- `mtval`
- `mtval2`
- `mtinst`

We also refer to the above CSRs using their aliases (with the `m-` prefix replaced `c-`) in the C-mode context.

The M-mode CSRs `mtvec`, `mscratch`, and `mepc` have their corresponding capability-extended replacements available in C-mode, as described in the following section.

2.7. Added Registers

The Capstone-RISC-V ISA adds the following registers. The +h sign indicates that the register is available in the hypervisor extension.

Table 3. Additional Registers in the Capstone-RISC-V ISA

Mnemonic	CCSR encoding	CSR encoding	Description
ctvec	0x000	-	The PC entry for the exception and V-interrupt handler (replacing mtvec)
cih	0x001	-	The sealed capability for the H-interrupt handler
cepc	0x002	-	The exception program counter register (replacing mepc)
cscratch	0x004	-	Replacing mscratch
cpmp0-15	0x010-0x01f	-	Capability for use with MMU-based memory accesses
cis	-	0x800	The H-interrupt status register
cid	-	0x801	The H-interrupt delegation register. If a bit is set, the H-interrupt is transformed into the corresponding V-interrupt directly
cic	-	0x802	The H-interrupt number register. Indicates the type of the H-interrupt
offsetmmu	-	0x803	Offset to add to the logical address to obtain the physical address

Some of the registers only allow capability values and have special semantics related to the system-wide machine state. They are referred to as *capability control and status registers* (CCSRs). Under their respective constraints, CCSRs can be manipulated using *control and status instructions*.

The manipulation constraints for each CCSR are indicated below.

Table 4. Manipulation Constraints for CCSRs

Mnemonic	Read	Write
ctvec	No constraint	No constraint
cih	Not allowed	The original content must not be a capability
cepc	No constraint	No constraint

Some of the registers are added as *control and status registers* (CSRs). These registers are manipulated by the same instructions that manipulate CSRs as in RV64IZicsr. When the manipulation constraints of these additional CSRs are not satisfied, the behaviour of these instructions follows the RV64IZicsr convention for other CSRs.

The manipulation constraints for each additional CSR are indicated below.

Table 5. Manipulation Constraints for Additional CSRs

Mnemonic	Read	Write
cis	No constraint	No constraint
offsetmmu	No constraint	No constraint

▼ **Note: ctvec and cih**

`ctvec` and `cih` should be handled differently.

`ctvec` is about the functionality of a domain only. A domain should be allowed to set `ctvec` for itself. That also means it needs to be switched when switching domains.

`cih` is about the functionality of the system, which should normally only be set once. To prevent any domain from setting `cih`, we require the original content of `cih` to be invalid for an attempt to change it to succeed.

2.8. Register Scopes

Each register has a scope that is either of the following:

- Domain: the register is specific to a domain
- Hart: the register is specific not to a domain, but to a hart

2.8.1. Domain-Scoped Registers

- PC: `pc`
- GPRs: `x1`, `x2`, ..., `x31`
- CCSRs: `ctvec`, `cepc`, `cpmp0-15`, `cscratch`
- 64-bit CSRs: `mstatus`, `mideleg`, `medeleg`, `mip`, `mie`, `mcause`, `mtval`, `mtval2`, `mtinst`, `stvec`, `scause`, `stval`, `sepc`, `sscratch`, `satp`, `offsetmmu`

Note that the complete domain-scope state also includes the privilege level, which is not reflected in any of the registers above. Hence, we propose a modified `mstatus` register that includes the current privilege level (0 — 3) in bits 38 and 39, but only in the context of saving or restoring domain-scoped registers.

2.8.2. Hart-Scoped Registers

Registers `cis`, `cid`, and `cih` are hart-scoped.

2.9. Register Effects

Domain-scoped registers that can immediately affect the behaviours of a domain in C-mode include `pc`, `ctvec`, `cscratch`, `mstatus`, `mideleg`, `medeleg`, `mip`, `mie`. We call such registers *C-effective registers*.

2.10. Context Layout in Memory

Under certain circumstances, a set of registers need to be mapped to memory locations in an architecturally-defined layout. For example, some registers need to be saved or restored upon context switches. In such cases, unless otherwise specified, the layout for both domain-scoped registers and C-effective registers follows the following order without any padding:

- C-effective registers (PC and CCSRs, 16 bytes): `pc`, `ctvec`, `cscratch`
- C-effective registers (CSRs, 8 bytes): `mstatus`, `mideleg`, `medeleg`, `mip`, `mie`
- `offsetmmu` (8 bytes)
- `cpmp0-15`, `cepc` (16 bytes)
- `x1`, `x2`, ..., `x31` (16 bytes)
- `mcause`, `mtval`, `mtval2`, `mtinst`, `stvec`, `scause`, `stval`, `sepc`, `sscratch`, `satp` (8 bytes)

NOTE

The layout described above takes care of the alignment maintains the same register offsets for domain-scoped contexts and C-effective contexts, making it possible to use a domain-scoped context as a C-effective context. When a C-effective context swapped in (loaded into the CPU state), and a domain-scoped context is swapped out (offloaded from the CPU state into memory), registers that are domain-scoped but not C-effective are scrubbed.

2.11. Extension to Memory

The memory is addressed using an `XLEN`-bit integer at byte-level granularity.

In addition to raw integers, each `CLEN`-bit aligned address can also store a capability. The type of data contained in a memory location is maintained and confusion of the type is not allowed.

▼ **Note: maintaining the type of data**

For a store operation that accesses the memory location `[addr, addr + size)`, the type of data contained in the memory location is maintained as follows:

- If a capability is stored to the memory location `[addr, addr + CLENBYTES)`, the type of data contained in the memory location will become a capability, where `addr` is `CLENBYTES`-byte aligned.
- If an integer is stored to the memory location `[addr, addr + size)`, it will make the `CLEN`-bit aligned memory location `[cbase, cend)` an integer, where `cbase = addr & ~(CLENBYTES - 1)` and `cend = cbase + CLENBYTES`.

Note

In this document, when we say the memory location `[addr, addr + CLENBYTES)`, we mean that the following content will be loaded from or stored to the memory location:

- Depending on the type of data contained in the memory location, the content being loaded from the memory location is either a capability at the memory location `[addr, addr + CLENBYTES)`, or an integer at the memory location `[addr, addr + XLENBYTES)`.
- Depending on the type of data being stored to the memory location, the data is either stored as a capability at the memory location `[addr, addr + CLENBYTES]`, or an integer at the memory location `[addr, addr + XLENBYTES)`.

The physical memory can *only* be accessed through capabilities.

Address Space	Access Method
`[0, 2^XLEN)`	Capabilities

▼ **Note: undefined behaviour**

The following load results are *undefined*:

- Load an integer from a memory location when the last capability store to its `CLENBYTES`-byte aligned memory location is more recent than the last integer store to the memory location itself.

2.12. Instruction Set

The Capstone-RISC-V instruction set is based on the RV64IZicsr instruction set. The (uncompressed) instructions are fixed 32-bit wide, and laid out in memory in little-endian order. In the encoding space of the RV64IZicsr instruction set, Capstone-RISC-V instructions occupies the "custom-2" subset, i.e., the opcode of all Capstone-RISC-V instructions is `0b1011011`.

Capstone-RISC-V instruction encodings follow three basic formats: R-type, I-type and S-type, as described below (more details are also provided in the *RISC-V ISA Manual*).

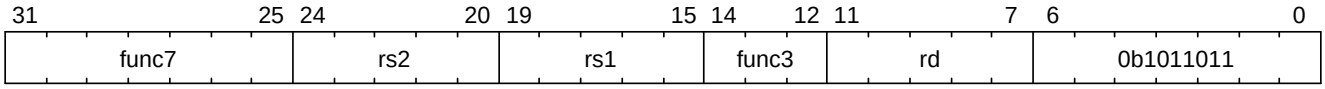

Figure 2. R-type instruction format

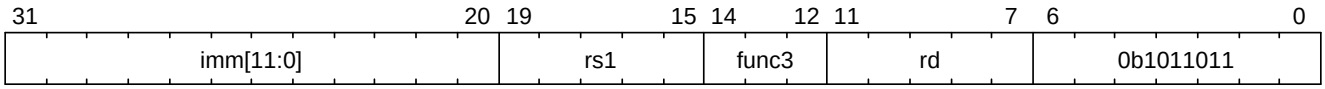

Figure 3. I-type instruction format

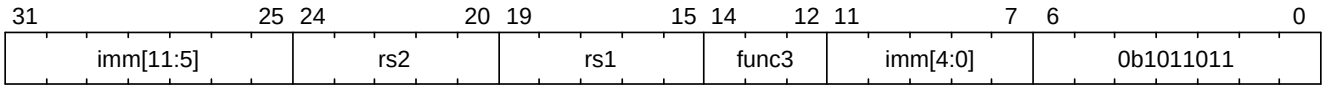

Figure 4. S-type instruction format

R-type instructions receive up to three register operands, and I-type/S-type instructions receive up to two register operands and a 12-bit-wide immediate operand.

The Capstone-RISC-V ISA also uses a register operand of R-type as an immediate operand in some instructions, which is called *register-immediate* (RI) type for convenience in this document.

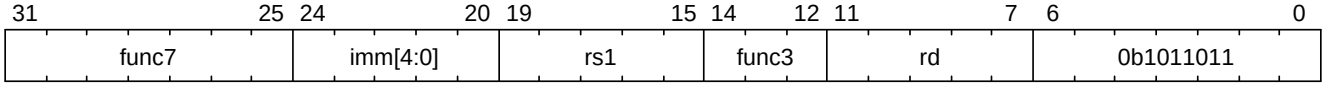

Figure 5. RI-type instruction format

The RI-type instruction format is derived from the R-type instruction format. An RI-type instruction receives up to two register operands and a 5-bit-wide immediate operand.

Unless otherwise specified, the instructions introduced in Capstone-RISC-V on top of the base RV64IZicsr instruction set are available in C-mode only. Attempts to execute them in other modes trigger `Illegal instruction (2)` exceptions.

2.13. System Reset

Upon reset, the system state must conform to the following specifications.

- Each general-purpose register either contains an integer, or a capability with `valid = 0` (invalid).
- No addressable memory location can contain a capability.
- `ctvec`, `cih`, and `cepc` contain either integers or capabilities with `valid = 0` (invalid).
- `cis = 0`.
- `pc` does not contain a capability.
- `CAPSTONE_EN = 0`

3. Capability Manipulation Instructions

Capstone provides instructions for creating, modifying, and destroying capabilities. Note that due to the guarantee of provenance of capabilities, those instructions are the *only* way to manipulate capabilities. In particular, it is not possible to manipulate capabilities by manipulating the content of a memory location or register using other instructions.

3.1. Cursor, Bounds, and Permissions Manipulation

3.1.1. Capability Movement

Capabilities can be moved between registers with the MOVC instruction.

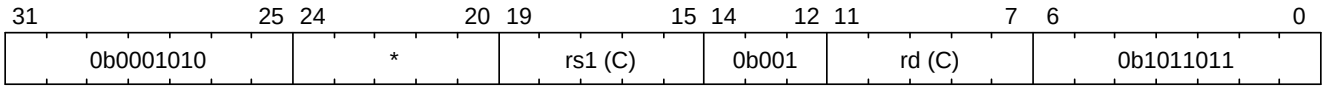

Figure 6. MOVC instruction format

No exception could be raised for MOVC.

- If `rs1 = rd`, the instruction is a no-op.
- Otherwise
  1. Write `x[rs1]` to `x[rd]`.
  2. If `x[rs1]` is not a non-linear capability (i.e., `type != 1`), write `cnull` to `x[rs1]`.

3.1.2. Cursor Increment

The CINCOFFSET and CINCOFFSETIMM instructions increment the `cursor` of a capability by a given amount (offset).

CINCOFFSET

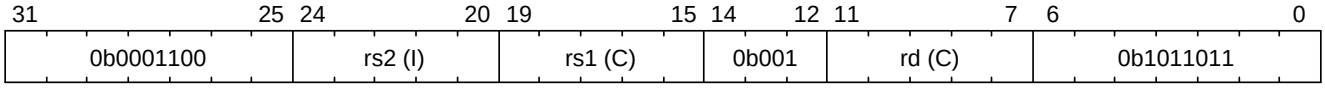

Figure 7. CINCOFFSET instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
  - `x[rs2]` is not an integer.
- `Unexpected capability type (26)`
  - `x[rs1]` has `type = 3` (uninitialised) or `type = 4` (sealed).

If no exception is raised:

1. Set `val` to `x[rs2]`.
2. `MOVC rd, rs1`.
3. Set `x[rd].cursor` to `x[rd].cursor + val`.

CINCOFFSETIMM

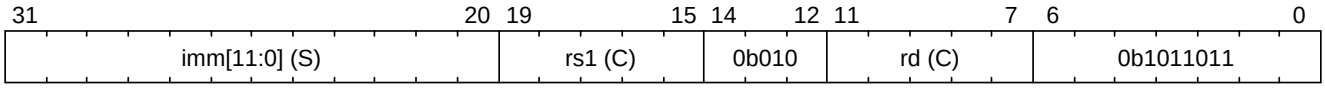

Figure 8. CINCOFFSETIMM instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Unexpected capability type (26)`
  - `x[rs1]` has `type = 3` (uninitialised) or `type = 4` (sealed).

If no exception is raised:

1. `MOVC rd, rs1`.
2. Set `x[rd].cursor` to `x[rd].cursor + imm`.

3.1.3. Cursor Setter

The `cursor` field of a capability can also be directly set with the SCC instruction.

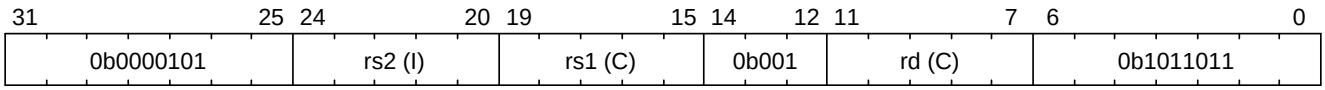

Figure 9. SCC instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
  - `x[rs2]` is not an integer.
- `Unexpected capability type (26)`
  - `x[rs1]` has `type = 3` (uninitialised) or `type = 4` (sealed).

If no exception is raised:

1. Set `val` to `x[rs2]`.
2. `MOVC rd, rs1`.
3. Set `x[rd].cursor` to `val`.

3.1.4. Field Query

The LCC instruction is used to read a field from a capability.

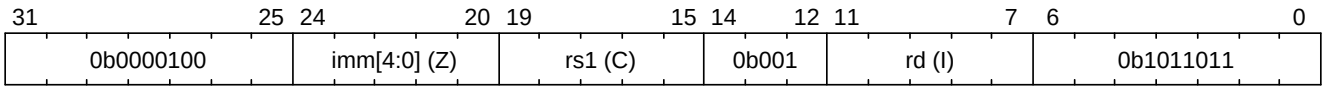

Figure 10. LCC instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Unexpected capability type (26)`
  - `imm = 2` and `x[rs1]` has `type = 4` (sealed).
  - `imm = 4` and `x[rs1]` has `type = 4` (sealed) or `type = 5` (sealed-return).
  - `imm = 5` and `x[rs1]` has `type = 4` (sealed) or `type = 5` (sealed-return).
  - `imm = 6` and `x[rs1]` does not have `type = 4` (sealed) or `type = 5` (sealed-return).
  - `imm = 7` and `x[rs1]` does not have `type = 5` (sealed-return).

If no exception is raised:

- If `imm > 7`, write `zero` to `x[rd]`
- Otherwise, write `field` to `x[rd]` according to the LCC multiplexing table.

Table 6. LCC multiplexing table

imm	field
0	x[rs1].valid
1	x[rs1].type
2	x[rs1].cursor
3	x[rs1].base
4	x[rs1].end
5	x[rs1].perms
6	x[rs1].async

imm	field
7	x[rs1].reg

3.1.5. Bounds Shrinking

The bounds (base and end fields) of a capability can be shrunk with the SHRINK instruction.

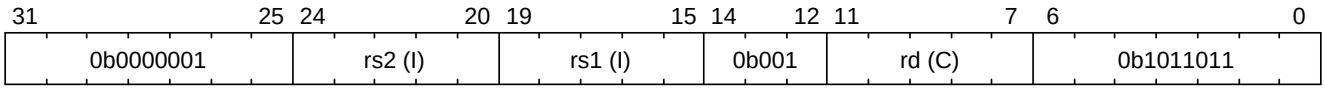

Figure 11. SHRINK instruction format

An exception is raised when any of the following conditions is met:

- Unexpected operand type (24)
  - x[rd] is not a capability.
  - x[rs1] is not an integer.
  - x[rs2] is not an integer.
- Unexpected capability type (26)
  - x[rd].type is not 0, 1, or 3 (linear, non-linear, or uninitialised).
- Illegal operand value (29)
  - x[rs1] >= x[rs2].
  - x[rs1] < x[rd].base or x[rs2] > x[rd].end.

If no exception is raised:

1. Set x[rd].base to x[rs1] and x[rd].end to x[rs2].
2. If x[rd].cursor < x[rs1], set x[rd].cursor to x[rs1].
3. If x[rd].cursor > x[rs2], set x[rd].cursor to x[rs2].

Another instruction, SHRINKTO, provides more convenience for certain common special cases of shrinking.

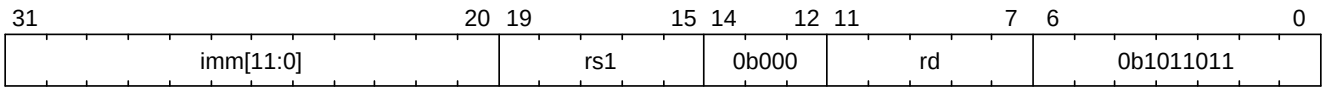

Figure 12. SHRINKTO instruction format

An exception is raised when any of the following conditions is met:

- Unexpected operand type (24)
  - x[rs1] is not a capability.

- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0`, `1`, or `3` (linear, non-linear, or uninitialised).
- `Illegal operand value (29)`
  - `x[rs1].cursor < x[rs1].base` or `x[rs1].cursor + imm > x[rs1].end`.

If no exception is raised:

1. `MOVC rd, rs1`.
2. Set `x[rd].base` to `x[rd].cursor` and `x[rd].end` to `x[rd].cursor + imm - 1`.

3.1.6. Bounds Splitting

The SPLIT instruction can split a capability into two by splitting the bounds. It attempts to split the capability `x[rs1]` into two capabilities, one with bounds `[x[rs1].base, x[rs2])` and the other with bounds `[x[rs2], x[rs1].end)`.

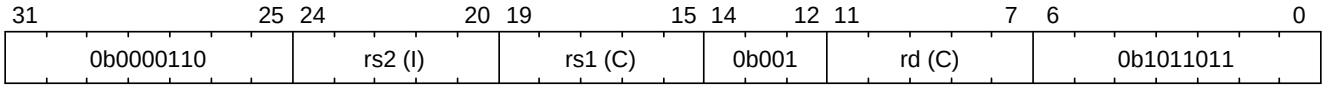

Figure 13. SPLIT instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
  - `x[rs2]` is not an integer.
- `Invalid capability (25)`
  - `x[rs1].valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `x[rs1].type` is neither `0` (linear) nor `1` (non-linear).
- `Illegal operand value (29)`
  - `x[rs2] < x[rs1].base` or `x[rs2] >= x[rs1].end`.

If no exception is raised:

1. If `rs1 = rd`, the instruction is a no-op.
2. Set `val` to `x[rs2]`.
3. Write `x[rs1]` to `x[rd]`.
4. Set `x[rs1].end` to `val`, `x[rs1].cursor` to `x[rs1].base`.

5. Set `x[rd].base` to `val + 1`, `x[rd].cursor` to `val + 1`.

3.1.7. Permission Tightening

The TIGHTEN instruction tightens the permissions (`perms` field) of a capability.

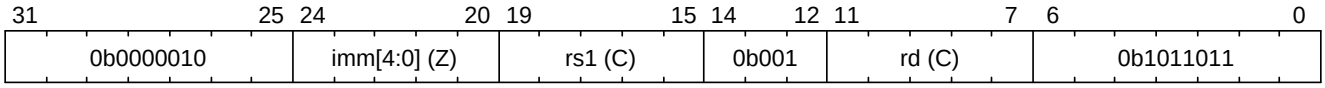

Figure 14. TIGHTEN instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0`, `1`, or `3` (linear, non-linear, or uninitialised).
- `Illegal operand value (29)`
  - `imm <= 7`, and `imm <=p x[rs1].perms` does not hold.

If no exception is raised:

1. `MOVC rd, rs1`.
2. If `imm > 7`, set `x[rd].perms` to `0`. Otherwise, set `x[rd].perms` to `imm`.

3.2. Type Manipulation

Some instructions can affect the `type` field of a capability directly. In general, the `type` field cannot be set arbitrarily. Instead, it is changed as the side effect of certain semantically significant operations.

3.2.1. Delinearisation

The DELIN instruction delinearises a linear capability.

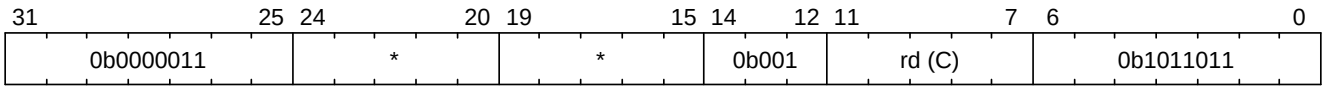

Figure 15. DELIN instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`

- `x[rd]` is not a capability.

- `Unexpected capability type (26)`
  - `x[rd].type` is not `0` (linear).

If no exception is raised:

- Set `x[rd].type` to `1` (non-linear).

3.2.2. Initialisation

The INIT instruction transforms an uninitialised capability into a linear capability after its associated memory region has been fully initialised (written with new data).

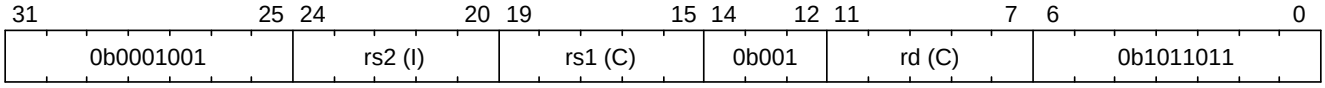

Figure 16. INIT instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
  - `x[rs2]` is not an integer.
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `3` (uninitialised).
- `Illegal operand value (29)`
  - `x[rs1].cursor <= x[rs1].end`.

If no exception is raised:

1. Set `val` to `x[rs2]`.
2. `MOVC rd, rs1`.
3. Set `x[rd].type` to `0` (linear), and `x[rd].cursor` to `x[rd].base + val`.

3.2.3. Sealing

The SEAL instruction seals a linear capability.

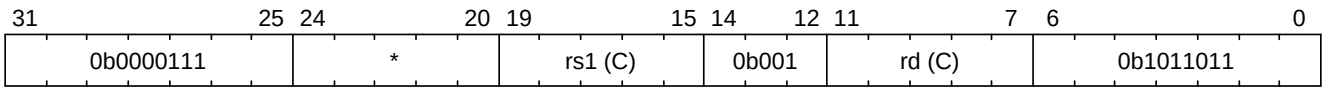

Figure 17. SEAL instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0` (linear).
- `Insufficient capability permissions (27)`
  - `6 <=p x[rs1].perms` does not hold.
- `Illegal operand value (29)`
  - The size of the memory region associated with `x[rs1]` is smaller than `CLENBYTES * 64` bytes (i.e., `x[rs1].end - x[rs1].base + 1 < CLENBYTES * 64`).
  - `x[rs1].base` is not aligned to `CLENBYTES` bytes.
  - The content of the memory region `[x[rs1].base + CLENBYTES, x[rs1].base + 2 * CLENBYTES)` does not contain a capability.

If no exception is raised:

1. `MOVC rd, rs1`.
2. Set `x[rd].type` to `2` (sealed), and `x[rd].async` to `0` (synchronous).

3.3. Dropping

The DROP instruction invalidates a capability.

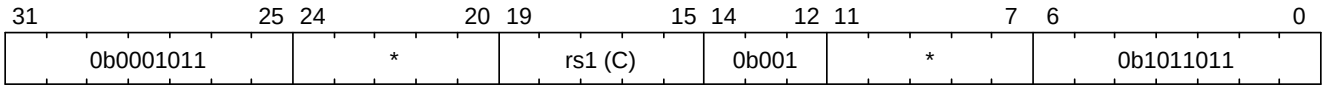

Figure 18. DROP instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.

If no exception is raised:

1. If `x[rs1].valid` is `0` (invalid), the instruction is a no-op.
2. Otherwise, set `x[rs1].valid` to `0` (invalid).

3.4. Revocation

3.4.1. Revocation Capability Creation

The MREV instruction creates a revocation capability.

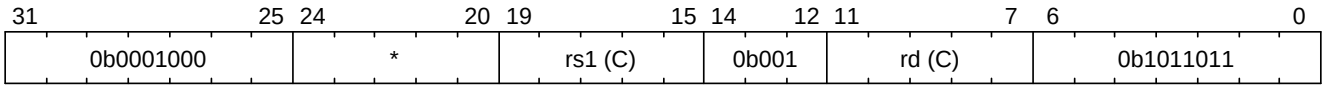

Figure 19. MREV instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Invalid capability (25)`
  - `x[rs1].valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0` (linear).

If no exception is raised:

1. Write `x[rs1]` to `x[rd]`.
2. Set `x[rd].type` to `2` (revocation).

3.4.2. Revocation Operation

The REVOKE instruction revokes a capability.

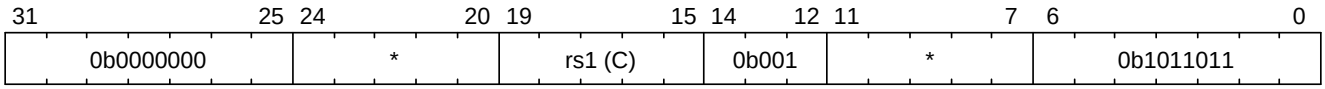

Figure 20. REVOKE instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Invalid capability (25)`
  - `x[rs1].valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `2` (revocation).

If no exception is raised:

1. For each capability `c` in the system (in either a register or memory location), `c.valid` is set to `0` (invalid) if any of the following conditions are met:
   - `c.type` is not `2` (revocation), `c.valid` is `1` (valid), and `c` aliases with `x[rs1]`.
   - `c.type` is `2` (revocation), `c.valid` is `1` (valid), and `x[rs1] <t c`.
2. `x[rs1].type` is set to `0` (linear) if at least one of the following conditions are met:
   - For every invalidated capability `c`, the type of `c` is non-linear (i.e., `c.type` is `1`).
   - `2 <=p x[rs1].perms` does not hold.
3. Otherwise, set `x[rs1].type` to `3` (uninitialised), and `x[rs1].cursor` to `x[rs1].base`.

4. Memory Access Instructions

Capstone provides instructions to load and store capabilities from/to memory regions.

4.1. Load Capabilities

The LDC instruction loads a capability or an integer scalar from the memory depending on the type of data at the specified memory location.

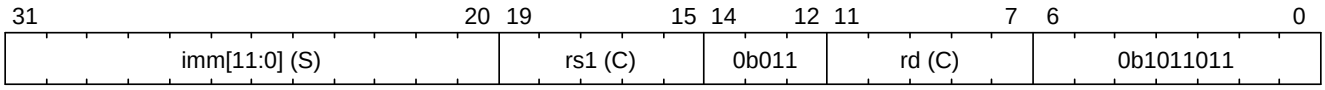

Figure 21. LDC instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Invalid capability (25)`
  - `x[rs1].valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0` (linear), `1` (non-linear) or `5` (sealed-return).
  - `x[rs1].type` is `5` (sealed-return) and `x[rs1].async` is not `0` (synchronous).
- `Insufficient capability permissions (27)`
  - `x[rs1].type` is `0` (linear) or `1` (non-linear) and `4 <=p x[rs1].perms` does not hold.
- `Capability out of bound (28)`
  - `x[rs1].type` is `0` (linear) or `1` (non-linear), and `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - CLENBYTES]`.
  - `x[rs1].type` is `5` (sealed-return), and `x[rs1].cursor + imm` is not in the range `[x[rs1].base + 3 * CLENBYTES, x[rs1].base + 64 * CLENBYTES - CLENBYTES]`.
- `Load address misaligned (4)`
  - `x[rs1].cursor + imm` is not aligned to `CLENBYTES` bytes.
- `Insufficient capability permissions (27)`
  - The data being loaded is not a scalar or a non-linear capability (i.e., `type != 1`), `x[rs1].type` is `0` (linear) or `1` (non-linear), and `2 <=p x[rs1].perms` does not hold.

If no exception is raised:

1. Set `cap` to `x[rs1]`.
2. Load the capability at the memory location `cap.cursor + imm, cap.cursor + imm +`

CLENBYTES)` into `x[rd]`.

3. If `x[rd].type` is not `1` (non-linear), write `cnull` to the memory location `[cap.cursor + imm, cap.cursor + imm + CLENBYTES)`.

4.2. Store Capabilities

The STC instruction stores a capability or an integer scalar to the memory, depending on the type of data contained in the specified source register.

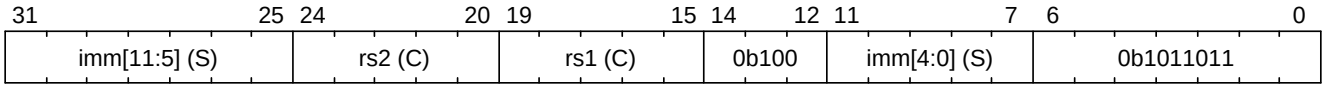

Figure 22. STC instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Invalid capability (25)`
  - `x[rs1].valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0` (linear), `1` (non-linear), `3` (uninitialised) or `5` (sealed-return).
  - `x[rs1].type` is `5` (sealed-return) and `x[rs1].async` is not `0` (synchronous).
- `Insufficient capability permissions (27)`
  - `x[rs1].type` is `0` or `1`, and `2 <=p x[rs1].perms` does not hold.
- `Illegal operand value (29)`
  - `x[rs1].type` is `3` (uninitialised) and `imm` is not `0`.
- `Capability out of bound (28)`
  - `x[rs1].type` is `0`, `1`, or `3`, and `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - CLENBYTES]`.
  - `x[rs1].type` is `5` or `6`, and `x[rs1].cursor + imm` is not in the range `[x[rs1].base + 3 * CLENBYTES, x[rs1].base + 64 * CLENBYTES - CLENBYTES]`.
- `Store/AMO address misaligned (6)`
  - `x[rs1].cursor + imm` is not aligned to `CLENBYTES` bytes.

If no exception is raised:

1. Store `x[rs2]` to the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + CLENBYTES)`.
2. If `x[rs1].type` is `3` (uninitialised), set `x[rs1].cursor` to `x[rs1].cursor + CLENBYTES`.

3. If `x[rs2]` is a capability and `x[rs2].type` is not `1` (non-linear), write `cnull` to `x[rs2]`.

5. Control Flow Instructions

5.1. Jump to Capabilities

The CJALR and CBNZ instructions allow jumping to a capability, i.e., setting the program counter to a given capability, in a unconditional or conditional manner.

5.1.1. CJALR

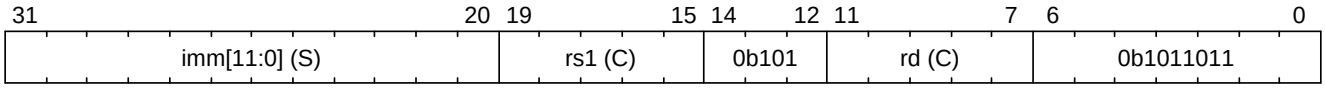

Figure 23. CJALR instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.

If no exception is raised:

1. Set `cap` to `x[rs1]`.
2. Set `pc.cursor` to `pc.cursor + 4`, write `pc` to `x[rd]`.
3. Set `cap.cursor` to `cap.cursor + imm`, write `cap` to `pc`.
4. If `rs1 != rd` and `x[rs1].type != 1`, write `cnull` to `x[rs1]`.

5.1.2. CBNZ

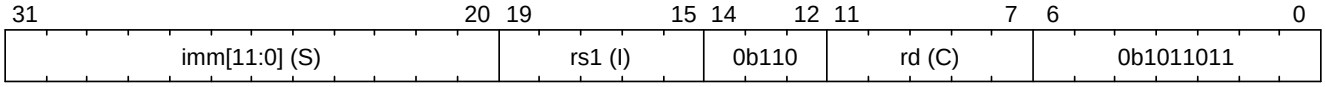

Figure 24. CBNZ instruction format

An exception is raised when any of the following conditions is met:

- `Illegal instruction (2)`
  - `cwrld` is `0` (normal world).
- `Unexpected operand type (24)`
  - `x[rd]` is not a capability.
  - `x[rs1]` is not an integer.

If no exception is raised:

- If `x[rs1]` is `0`, the instruction is a no-op.
- Otherwise
  1. Write `x[rd]` to `pc`.
  2. Set `pc.cursor` to `pc.cursor + imm`.
  3. If `x[rd].type != 1`, write `cnull` to `x[rd]`.

5.2. Domain Crossing

Domains in the Capstone-RISC-V ISA are individual software compartments that are protected by a safe context switching mechanism, i.e., *domain crossing*. The mechanism is provided by the CALL and RETURN instructions.

5.2.1. CALL

The CALL instruction is used to call a sealed capability, i.e., to switch to another *domain*.

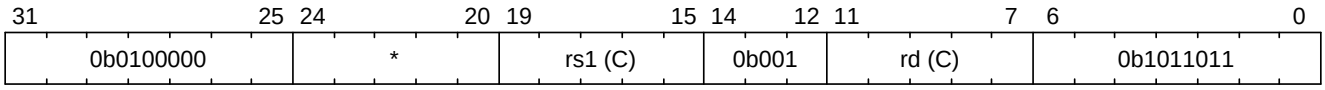

Figure 25. CALL instruction format

We define `d` as the following:

- If `rs1 != 0`, then `d` is the content of `x[rs1]`.
- Otherwise, then `d` is the content of `cih`.

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `d` is not a capability.
- `Invalid capability (25)`
  - `d.valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `d.type` is not `4` (sealed).
  - `d.async` is not `0` (synchronous).

If no exception is raised:

1. If `rs1 != 0`, `MOVC cra, rs1`, or otherwise `CSRRW cra, cih, c0`.
2. Swap C-effective registers with the memory content at address `cra.base`.
3. Set `cra.type` to `5` (sealed-return), `cra.cursor` to `cra.base`, `cra.reg` to `rd`, and `cra.async` to `0`

(synchronous).

5.2.2. RETURN

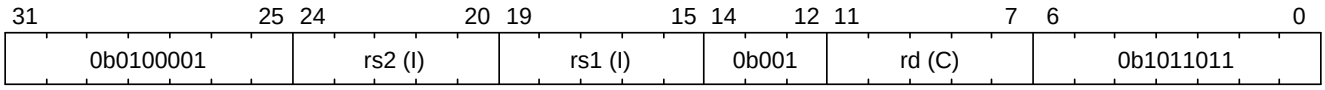

Figure 26. RETURN instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rd]` is not a capability.
  - `x[rs1]` is not an integer.
- `Invalid capability (25)`
  - `x[rd].valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `x[rd].type` is not `4` (sealed) or `5` (sealed-return).
  - `x[rd].type` is `4` (sealed) and `cih != cnull`.

If no exception is raised:

When `x[rd].async = 0` (synchronous):

1. Write `x[rd]` to `cap` and `cnull` to `x[rd]`.
2. Set `pc.cursor` to `x[rs1]`.
3. Swap C-effective registers with the memory content at address `cap.base`.
4. If `cap.type = 5` and `cap.reg != 0`, write `cap` to `x[cap.reg]` and set `x[cap.reg].type` to `4` (sealed). Otherwise, write `cap` to `cih` and set `cih.type` to `4` (sealed).

When `x[rd].async = 1` (asynchronous):

1. Write `x[rs2]` to `posted_ints`.
2. Set `x[rd].type` to `4` (sealed), `x[rd].async` to `0` (synchronous).
3. Write the resulting `x[rd]` to `cih`, and `cnull` to `x[rd]`.
4. Set `pc.cursor` to `x[rs1]`.
5. Swap out the C-effective registers, and swap in domain-scoped registers from the memory content at address `cih.base`.
6. Set `mip` to `mip | posted_ints`.

6. Control and Status Instructions

The CCSRRW instruction is used to read and write specified *capability control and status registers* (CCSRs).

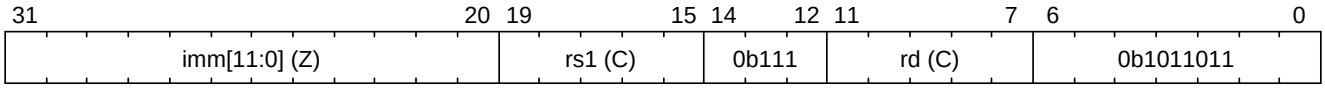

Figure 27. CCSRRW instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Illegal operand value (29)`
  - `imm` does not correspond to the encoding of a valid CCSR.

If no exception is raised:

1. If the read constraint is satisfied
   - The content of the CCSR specified by `imm` is written to `x[rd]`.
   - If `x[rd].type` is not `1` (non-linear), write `cnull` to the CCSR specified by `imm`.
2. Otherwise, write `cnull` to `x[rd]`.
3. If the write constraint is satisfied
   - Write `x[rs1]` to the CCSR specified by `imm`.
   - If `x[rs1].type` is not `1` (non-linear), write `cnull` to `x[rs1]`.
4. Otherwise, preserve the current content of the CCSR specified by `imm`.

The CAPENTER instruction enables C-mode. Unlike the other added instructions which are for C-mode only, CAPENTER can be used in M-mode.

When C-mode is enabled, the CAPENTER instruction also initialises the genesis capabilities and create the initial execution context of the genesis domain.

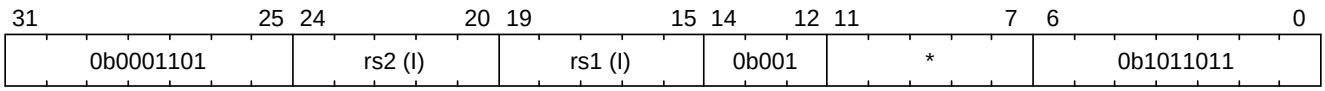

Figure 28. CAPENTER instruction format

An exception is raised when the following condition is met:

- `Illegal instruction (2)`
  - `CAPSTONE_EN = 1`.

If no exception is raised:

- If `rs1 == 0` and `rs2 == 0`
  - Set `pc = { valid = 1, type = 1, cursor = pc, base = INIT_CODE_BASE, end = INIT_CODE_END - 1, perms = 7 }`.
  - Set `x10 = { valid = 1, type = 1, cursor = INIT_DATA_BASE, base = INIT_DATA_BASE, end = INIT_DATA_END - 1, perms = 7 }`.
  - Set `x11 = { valid = 1, type = 2, cursor = 0, base = 0, end = 2^XLEN - 1, perms = 7 }`, which, once revoked, will invalidate the initial capabilities in `pc` and `x10` as defined above.
  - Set `CAPSTONE_EN` to `1`.
- Otherwise
  - Set `pc = { valid = 1, type = 1, cursor = pc, base = x[rs1], end = x[rs2] - 1, perms = 7 }`.
  - Set `x10 = { valid = 1, type = 1, cursor = 0, base = 0, end = x[rs1] - 1, perms = 7 }`.
  - Set `x11 = { valid = 1, type = 1, cursor = x[rs2], base = x[rs2], end = 2^XLEN - 1, perms = 7 }`.
  - Set `CAPSTONE_EN` to `1`.

`INIT_CODE_BASE`, `INIT_CODE_END`, `INIT_DATA_BASE`, and `INIT_DATA_END` are implementation-defined.

7. Adjustments to Existing Instructions

For most of the existing instructions in RV64IZicsr, their behaviour is unmodified. The `cursor` field (if `type != 4`) or `base` field (if `type = 4`) of the capability is used if a register containing a capability is used as an operand.

The following instructions in RV64IZicsr are adjusted in Capstone:

- For memory access instructions, they are adjusted to use capabilities as addresses for memory access.
- For control flow instructions, they are adjusted for the case where the program counter is a capability.
- Some instructions in RV64IZicsr become illegal instructions in the Capstone-RISC-V ISA.

7.1. Memory Access Instructions

In RV64IZicsr, memory access instructions include load instructions (i.e., `lb`, `lh`, `ld`, `lw`, `lbu`, `lhu`, `lwu`), and store instructions (i.e., `sb`, `sh`, `sw`, `sd`). These instructions take an integer as a raw address, and load or store a value from/to this address. In Capstone, these instructions are extended to take a capability as an address *when executed in C-mode*. In non-C modes, the behaviours of those instructions are unchanged.

7.1.1. Load Instructions

In the Capstone-RISC-V ISA, RV64IZicsr load instructions are modified to load integers of different sizes using capabilities.

▼ **Note: `size` of load instructions**

The `size` used in this sections is the size (in bytes) of the integer being loaded.

Mnemonic	`size`
`lb`	1
`lbu`	1
`lh`	2
`lhu`	2
`lw`	4
`lwu`	4
`ld`	8

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`

- `x[rs1]` is not a capability.

- `Invalid capability (25)`
  - `x[rs1].valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0` (linear), `1` (non-linear) or `5` (sealed-return).
  - `x[rs1].type` is `5` (sealed-return) and `x[rs1].async` is not `0` (synchronous).
- `Insufficient capability permissions (27)`
  - `x[rs1].type` is `0` (linear) or 1 (non-linear) and `4 <=p x[rs1].perms` does not hold.
- `Capability out of bound (28)`
  - `x[rs1].type` is `0` (linear) or `1` (non-linear), and `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - size]`.
  - `x[rs1].type` is `5` (sealed-return), and `x[rs1].cursor + imm` is not in the range `[x[rs1].base + 3 * CLENBYTES, x[rs1].base + 64 * CLENBYTES - size]`.
- `Load address misaligned (4)`
  - `x[rs1].cursor + imm` is not aligned to `size` bytes.

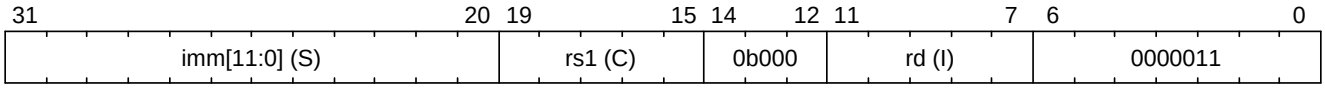

Figure 29. lb instruction format

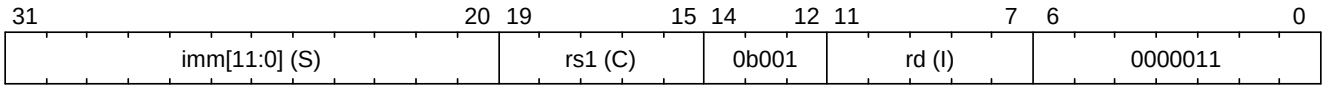

Figure 30. lh instruction format

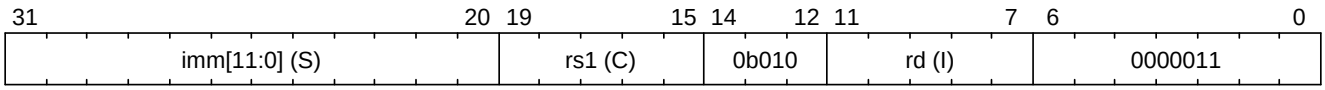

Figure 31. lw instruction format

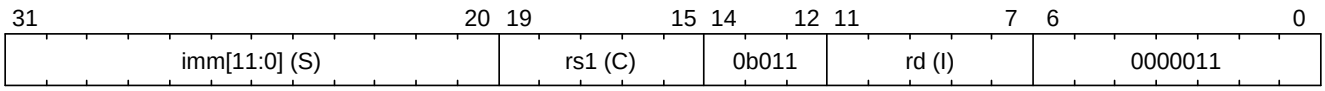

Figure 32. ld instruction format

If no exception is raised:

- Load the content at the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + size)` as a signed integer to `x[rd]`.

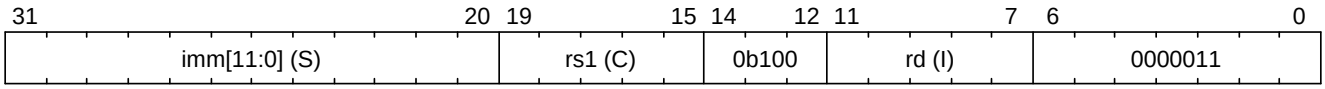

Figure 33. lbu instruction format

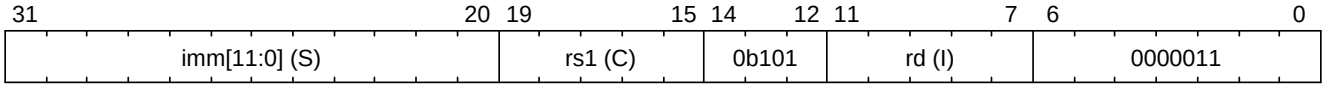

Figure 34. lhu instruction format

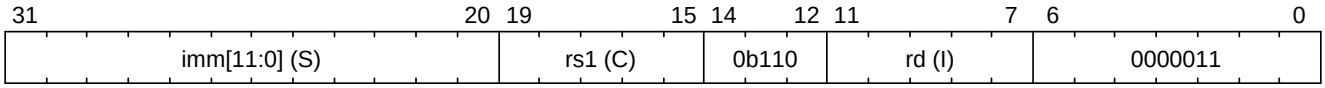

Figure 35. lwu instruction format

If no exception is raised:

- Load the content at the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + size)` as an unsigned integer to `x[rd]`.

7.1.2. Store Instructions

▼ **Note: `size` of store instructions**

The `size` used in this sections is the size (in bytes) of the integer being stored.

Mnemonic	size
sb	1
sh	2
sw	4
sd	8

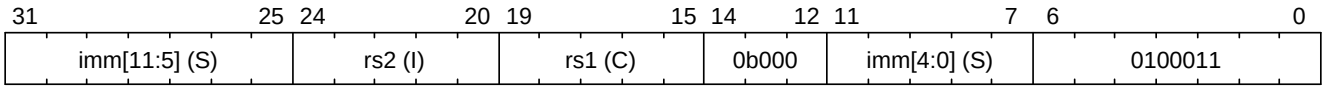

Figure 36. sb instruction format

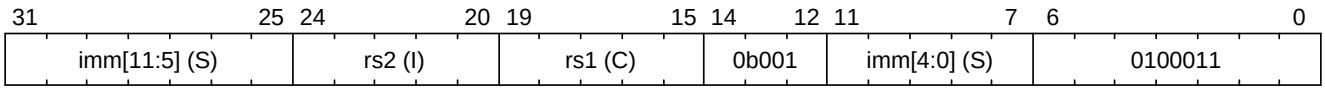

Figure 37. sh instruction format

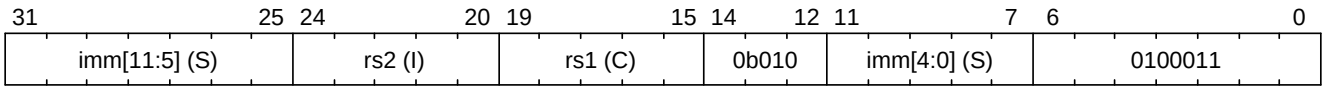

Figure 38. sw instruction format

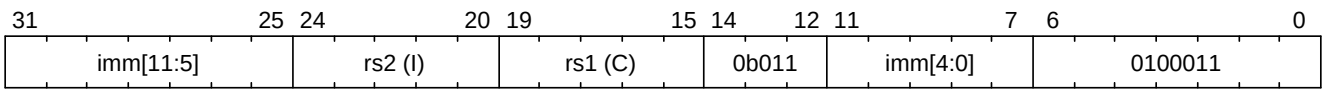

Figure 39. sd instruction format

An exception is raised when any of the following conditions is met:

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
  - `x[rs2]` is not an integer.
- `Invalid capability (25)`
  - `x[rs1].valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0` (linear), `1` (non-linear), `3` (uninitialised) or `5` (sealed-return).
  - `x[rs1].type` is `5` (sealed-return) and `x[rs1].async` is not `0` (synchronous).
- `Insufficient capability permissions (27)`
  - `x[rs1].type` is `0` or `1`, and `2 <=p x[rs1].perms` does not hold.
- `Illegal operand value (29)`
  - `x[rs1].type` is `3` (uninitialised) and `imm` is not `0`.
- `Capability out of bound (28)`
  - `x[rs1].type` is `0`, `1`, or `3`, and `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - size]`.
  - `x[rs1].type` is `5` or `6`, and `x[rs1].cursor + imm` is not in the range `[x[rs1].base + 3 * CLENBYTES, x[rs1].base + 64 * CLENBYTES - size]`.
- `Store/AMO address misaligned (6)`
  - `x[rs1].cursor + imm` is not aligned to `size` bytes.

If no exception is raised:

1. Store `x[rs2]` to the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + size)` as an integer.

7.2. Control Flow Instructions

In RV64IZicsr, conditional branch instructions (i.e., `beq`, `bne`, `blt`, `bge`, `bltu`, and `bgeu`), and unconditional jump instructions (i.e., `jal` and `jalr`) are used to control the flow of execution. In Capstone, these instructions are adjusted to support the situation where the program counter is a capability *when executed in C-mode*. In non-C modes, the behaviours of those instructions are unchanged.

7.2.1. Branch Instructions

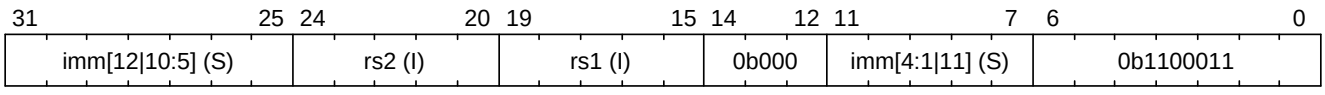

Figure 40. beq instruction format

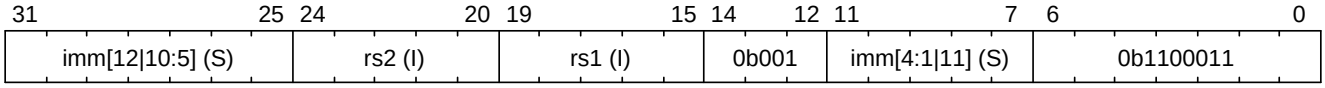

Figure 41. bne instruction format

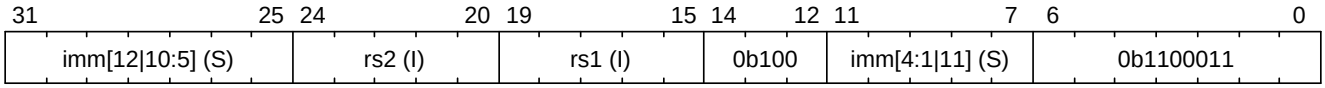

Figure 42. blt instruction format

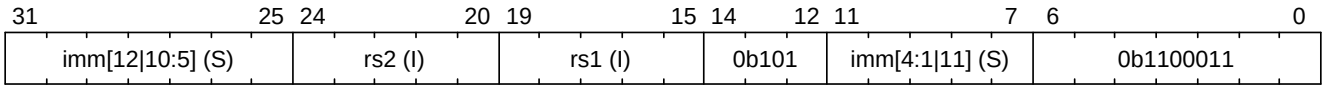

Figure 43. bge instruction format

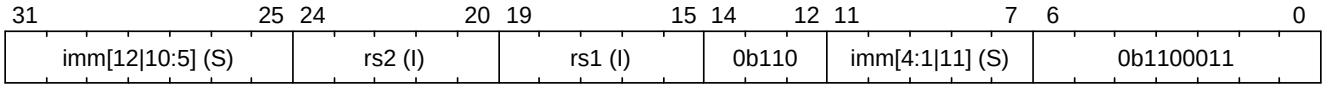

Figure 44. bltu instruction format

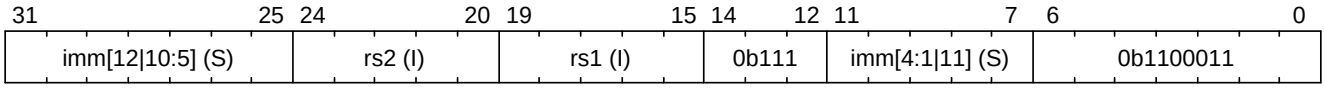

Figure 45. bgeu instruction format

The following adjustments are made to these instructions:

- `pc.cursor`, instead of `pc`, is changed by the instruction.

7.2.2. Jump Instructions

Figure 46. jal instruction format

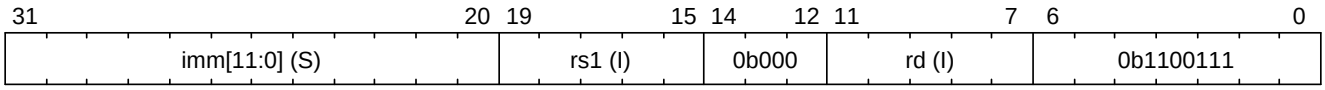

Figure 47. jalr instruction format

The following adjustments are made to these instructions:

- `pc.cursor + 4`, instead of `pc + 4`, is written to `x[rd]`.
- `pc.cursor`, instead of `pc`, is changed by the instruction.

7.3. Illegal Instructions

C-mode can execute the following M-mode privileged instructions defined in Zicsr:

- ECALL
- EBREAK
- MRET (also referred to as CRET in the C-mode context)

When a CSR that is available in M-mode but not available in C-mode is accessed in C-mode, an `illegal instruction (2)` exception is raised.

8. Interrupts and Exceptions

The mechanisms described in this section only apply when C-mode is enabled, i.e., when `CAPSTONE_EN = 1`.

8.1. Exception Codes

We extend RV64IZicsr with the following extra exception codes:

Table 7. Exception codes for the Capstone-RISC-V ISA

Exception	Exception code
Unexpected operand type	24
Invalid capability	25
Unexpected capability type	26
Insufficient capability permissions	27
Capability out of bound	28
Illegal operand value	29
Insufficient system resources	30

Those extra exception codes are only visible to C-mode. In other words, only C-mode software is allowed to handle those exceptions. They cannot be delegated to S-mode or U-mode.

For interrupts, the same encodings as in RV64IZicsr are used.

▼ **Note: Implementation specified exception**

For some of the exception code, where the corresponding exception is raised is not specified as part of the ISA specification. Instead, it is up to the implementation to decide where to raise the exception. These exceptions include:

- `Insufficient system resources (30)`

8.2. Exception Data

To provide the exception handler with extra exception-related information along with the exception code, Capstone-RISC-V follows the base RV64IZicsr and uses the CSRs `xepc`, `xtval`, `xtval2`, and `xtinst`. The only difference is that `cepc` will hold a capability if the excepting state has a capability `pc` as in the case of a horizontal trap in C-mode.

For exceptions defined in RV64IZicsr, the data written to those CSRs remain unchanged. The data provided for the added exception types defined above are as follows:

Table 8. Exception data for added exception types

Exception	xtval	xtval2	xtinst
Unexpected operand type (24)			The instruction itself (or the lowest XLEN bits if it is wider than XLEN)
Invalid capability (25)			The instruction itself (or the lowest XLEN bits if it is wider than XLEN)
Unexpected capability type (26)			The instruction itself (or the lowest XLEN bits if it is wider than XLEN)
Insufficient capability permissions (27)			The instruction itself (or the lowest XLEN bits if it is wider than XLEN)
Capability out of bound (28)			The instruction itself (or the lowest XLEN bits if it is wider than XLEN)
Illegal operand value (29)			The instruction itself (or the lowest XLEN bits if it is wider than XLEN)

8.3. Overview of Interrupt and Exception Handling

With `CAPSTONE_EN = 0`, the interrupt (H-interrupts only, as the notion of the V-interrupt does not apply here) and exception handling mechanisms are same as they are defined in RV64IZicsr.

With `CAPSTONE_EN = 1`, the handling of exceptions and V-interrupts uses the same mechanisms for exception and interrupt handling in RV64IZicsr, with the following minor changes:

- C-mode replaces M-mode.
- `ctvec` does not support the vectored mode, and should be a capability.
- When the CPU traps into C-mode, `ctvec` and `pc` must not both be integers. If `ctvec` is a capability, `pc` is written into `cepc` and replaced with `ctvec`, and if `ctvec` is linear, `ctvec` is replaced with `ctvec.cursor`. If `ctvec` is not a capability and `pc` is a capability, then the `pc.cursor` is written into `cepc` and replaced with `ctvec`. Upon `mret`, if `ctvec` is not a capability, it is replaced with `pc` with `ctvec.cursor` set to the original `ctvec` value.
- `cepc` can contain a capability.
- Extra exception and interrupt types are defined.

The handling of H-interrupts with `CAPSTONE_EN = 1` is new. The hart transfers the control flow to a dedicated H-interrupt handler domain (specified in `cih`). The current context is saved and sealed in a sealed-return capability which is then supplied to the H-interrupt handler domain as an argument. When handling is complete, the H-interrupt handler domain can use the RETURN instruction to resume the execution of the excepted domain. This process resembles that of a CALL-RETURN pair, except that it is asynchronous, rather than synchronous, to the execution of the original domain.

> The figure below shows the overview of domain switch in the Capstone-RISC-V ISA, including synchronous domain crossing and asynchronous interrupt/exception handling.

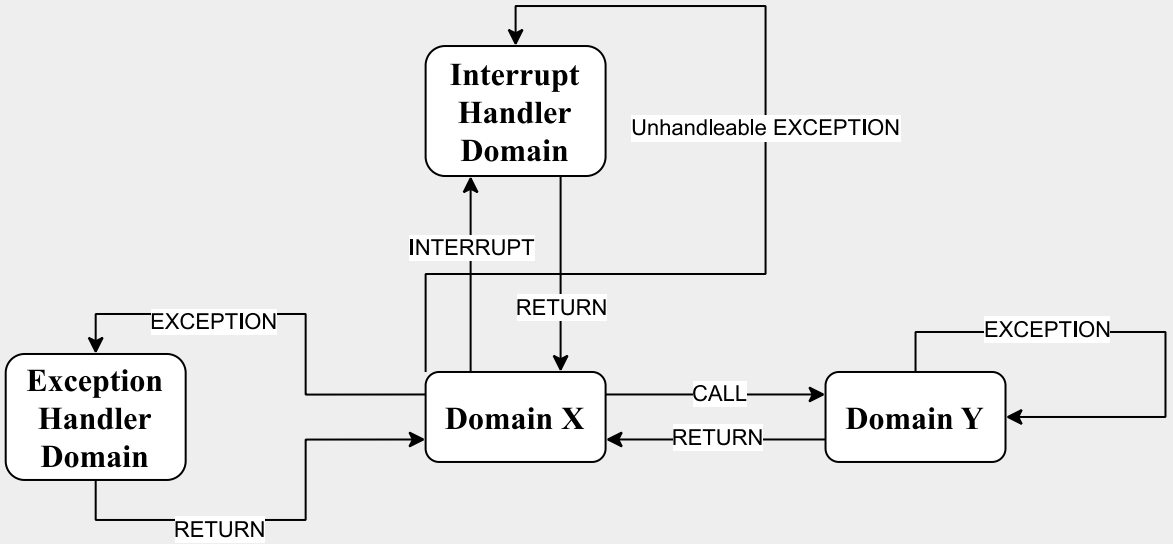

Figure 48. Overview of domain switch in the Capstone-RISC-V ISA

8.4. H-Interrupt Status

The `cis` CSR encodes the control and status associated with H-interrupts. The diagram below shows its layout.

15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIP	0	SEIP	0	MTIP	0	STIP	0	MSIP	0	SSIP	0

Figure 49. `cis` CSR layout

The layout of `cis` follows that of `mip` in RV64IZicsr. When a bit is set, the corresponding H-interrupt is pending.

All the fields are read-write.

8.5. H-Interrupt Delegation

The `cid` CSR specifies types of H-interrupts to delegate as V-interrupts to the running domain. The layout of `cid` is identical to that of `cis`. When a bit is set in `cid`, the corresponding H-interrupt is not taken to the interrupt handling domain defined in `cih`, but handled within the currently running domain as a corresponding C-mode V-interrupt.

8.6. Interaction with the Interrupt Controller

When the interrupt controller attempts to change the pending status of an interrupt line for a hart, either `mip` or `cis` is affected. Assuming the interrupt corresponds to bit `x` in `mip` and `cis`, the following is the criteria for deciding which of the two to operate on:

- If `CAPSTONE_EN = 0`, `mip` is affected.
- If `CAPSTONE_EN = 1`,
  - If `x` is clear in `cid`, `cis` is affected.

- If `x` is set in `cid`, `mip` is affected.

For example, if the interrupt controller wishes to clear the machine timer interrupt pending bit (MTIP) when `CAPSTONE_EN = 1` and the `MTIP` bit in `cid` is set, the `MTIP` bit in `mip` will be cleared.

8.7. H-Interrupt Delivery

The H-interrupt delivery process starts with a certain event typically asynchronous to the execution of the hart. The sources of such events include the external interrupt controller, the timer, and other CPU cores, which correspond to the external, timer, and software H-interrupt types (i.e., `x = E`, `T`, and `S`). When such an event occurs, the `xIP` field in the `cis` register is set to `1` to indicate that the H-interrupt is pending.

At any point during the execution of a hart, if any `xIP` field is set and at the same time the `cih` register contains a capability, the H-interrupt is delivered to the H-interrupt handler domain.

▼ **Note: global H-interrupt enable/disable**

> In the Capstone-RISC-V ISA, the `cih` register acts as a global H-interrupt-enable flag. If `cih` register does not contain a capability, all H-interrupts are disabled globally.

8.8. H-Interrupt Handling

The H-interrupt is ignored if any of the following conditions is met:

- `cih` is not a capability.
- `cih.valid = 0` (invalid).
- `cih.type != 4` (sealed capability).
- `cih.async != 0` (synchronous).

Otherwise:

1. Swap out the domain-scoped registers, and swap in C-effective registers from the memory content at address `cih.base`.
2. Set `cih.type` to `5` (sealed-return), `cih.cursor` to `cih.base`, `cih.reg` to `0`, and `cih.async` to `1` (upon interrupt).
3. Write `cih` to the register `cra`, and `cnull` to the register `cih`.
4. Write the interrupt code to the register `a0`.

Appendix A: Instruction Listing

A.1. Capstone Instructions

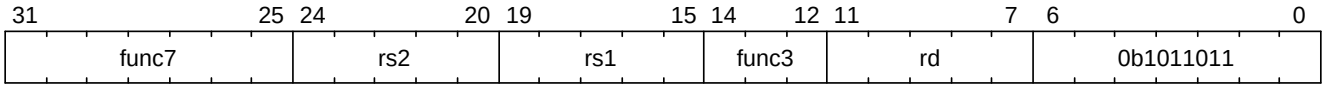

Figure 50. Instruction format: R-type

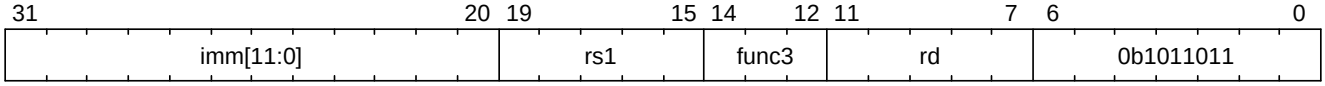

Figure 51. Instruction format: I-type

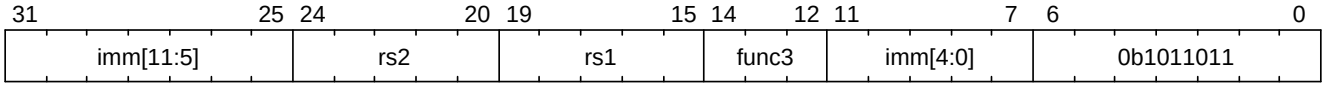

Figure 52. Instruction format: S-type

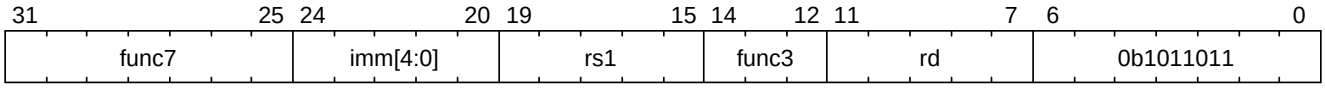

Figure 53. Instruction format: RI-type

Table 9. Capability manipulation instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm [4:0]	imm[11:0]
REVOKE	R	001	0000000	C	-	-	-	-
SHRINK	R	001	0000001	I	I	C	-	-
SHRINKTO	I	000	-	C	-	C	-	Z
TIGHTEN	RI	001	0000010	C	-	C	Z	-
DELIN	R	001	0000011	-	-	C	-	-
LCC	RI	001	0000100	C	-	I	Z	-
SCC	R	001	0000101	C	I	C	-	-
SPLIT	R	001	0000110	C	I	C	-	-
SEAL	R	001	0000111	C	-	C	-	-
MREV	R	001	0001000	C	-	C	-	-
INIT	R	001	0001001	C	I	C	-	-
MOVC	R	001	0001010	C	-	C	-	-
DROP	R	001	0001011	C	-	-	-	-
CINCOFFSET	R	001	0001100	C	I	C	-	-
CINCOFFSETIMM	I	010	-	C	-	C	-	S

Table 10. Memory access instructions

Mnemonic	Format	emode	Func3	Func7	rs1	rs2	rd	imm[11:0]
LDC	I	-	011	-	C	-	C	S
STC	S	-	100	-	C	C	-	S

Table 11. Control flow instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]
CALL	R	001	0100000	C	-	C	-
RETURN	R	001	0100001	C	I	-	-
CJALR	I	101	-	C	-	C	S
CBNZ	I	110	-	I	-	C	S

Table 12. Control and status instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]
CCSRRW	I	111	-	C	-	C	Z
CAPENTER	R	001	0001101	I	I	-	-

A.2. Adjusted RV64IZicsr Memory Access Instructions

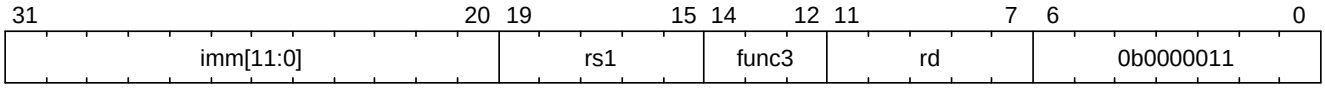

Figure 54. Instruction format: I-type

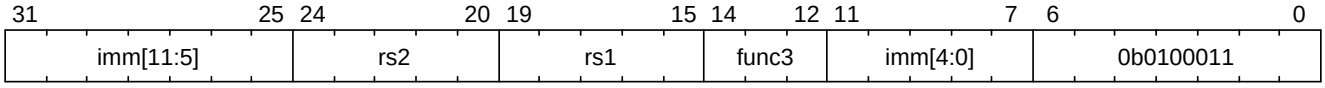

Figure 55. Instruction format: S-type

Table 13. Extended RV64IZicsr load instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]
lb	I	000	-	C	-	I	S
lh	I	001	-	C	-	I	S
lw	I	010	-	C	-	I	S
ld	I	011	-	C	-	I	S
lbu	I	100	-	C	-	I	S
lhu	I	101	-	C	-	I	S
lwu	I	110	-	C	-	I	S

Table 14. Extended RV64IZicsr store instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]
sb	S	000	-	C	I	-	S

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]
sh	S	001	-	C	I	-	S
sw	S	010	-	C	I	-	S
sd	S	011	-	C	I	-	S

▼ **Note: the meaning of abbreviations in the table**

For instruction operands:

I
: Integer register

C
: Capability register

-
: Not used

For immediates:

S
: Sign-extended

Z
: Zero-extended

-
: Not used