# The Capstone-RISC-V Instruction Set Reference

## Table of Contents

Contributors to this document include (in alphabetical order): Jason Zhijingcheng Yu, Mingkai Li

**Version Information:** Draft version. Refer to the commit hash.

# 1. Introduction

The Capstone project is an effort to explore the design of a new CPU instruction set architecture that achieves multiple security goals including memory safety and isolation with one unified hardware abstraction.

## 1.1. Goals

The ultimate goal of Capstone is to unify the numerous hardware abstracts that have been added as extensions to existing architectures as afterthought mitigations to security vulnerabilities. This goal requires a high level of flexibility and extensibility of the Capstone architecture. More specifically, we aim to support the following in a unified manner.

**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, unlike traditional platforms which follow a static hierarchy of hypervisor-kernel-user. 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 of context switching without trusting any other software component should be provided. This allows for a minimal TCB if necessary in case of a highly security-critical application.

## 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. Capstone extends the traditional capability model with new capability types including the following.

**Linear capabilities**

Linear capabilities are guaranteed not to alias with other capabilities. Operations on linear capabilities maintain this property. For example, linear capabilities cannot be duplicated. Instead, they can only be moved around across different registers or between registers and memory. 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) capability derived from the same linear capability. 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", that is, 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 extension to RISC-V due to its open nature and the availability of toolchains and simulators.

The Capstone-RISC-V ISA is an RV64IZicsr extension 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.
- 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.
- New instructions for machine state control are added.
- Semantics of a small number of existing instructions are changed to support capabilities.
- Semantics of interrupts and exceptions are changed to support capabilities.

## 1.4. Capstone Variants

In addition to Capstone, which is referred to as *Pure Capstone* in the Capstone-RISC-V ISA, we propose a variant of Capstone, called *TransCapstone*. While memory accesses and control flow transfers are only possible using capabilities in *Pure Capstone*, *TransCapstone* fuses capabilities with privilege levels and virtual memory found in traditional architectures, which allows for a smooth transition from existing architectures to Capstone.

The following types of changes are made to *Pure Capstone* to obtain *TransCapstone*:

- The physical memory is partitioned into two disjoint regions, one exclusively for accesses through capabilities and the other exclusively for accesses through the virtual memory.
- Software components are allowed to run in either of the 2 *worlds*, i.e., the *normal world* and the *secure world*.
    - The *normal world* follows the traditional privilege levels, allows both capability-based accesses and virtual memory accesses, and is therefore compatible with existing softwares.

    - The *secure world* follows the *Pure Capstone* design, limits memory accesses to capability-based accesses and provides the security guarantees of Capstone.
- A world switching mechanism is added to support the secure switching between the 2 *worlds*.
- Semantics of a small number of *Pure Capstone* instructions are changed to support the *normal world* and the *secure world*.
- Semantics of interrupts and exceptions are extended to support the *normal world* and the *secure world*.

*Table 1. Memory Accesses in TransCapstone*

| World | MMU | Capabilities |
|---|---|---|
| *Normal World* | Yes | Yes |
| *Secure World* | - | Yes |

## 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 design of Capstone has 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. Capabilities

### 2.1.1. 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.

### 2.1.2. Fields

Each capability has the following architecturally-visible fields:

*Table 2. 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, `6` = exit |
| `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` | Not applicable when `type = 6` (exit). The base memory address of the memory region associated with the capability |
| `end` | `0..2^XLEN-1` | Not applicable when `type = 4` (sealed), `type = 5` (sealed-return), or `type = 6` (exit). The end memory address of the memory region associated with the capability |

| Name | Range | Description |
|---|---|---|
| `perms` | `0..7` | Not applicable when `type = 4` (sealed), `type = 5` (sealed-return), or `type = 6` (exit). 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..2` | Only applicable when `type = 4` (sealed) or `type = 5` (sealed-return). Whether the capability is sealed asynchronously: `0` = synchronously, `1` = upon exception, `2` = upon interrupt |
| `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 `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.

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:

- For load and store instructions that move a capability between a register and memory, the value of the capability is preserved.
- 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` (TODO: double-check this claim).

### Note

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 3. 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 | Yes |
| Exit | `6` | Yes | Yes | Yes | - | - | - | - |

### Note

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 by sealed and sealed-return capabilities in different scenarios.

*Table 4. Memory accesses granted by sealed and sealed-return capabilities*

| **Capability type** | `async` | **Read** | **Write** | **Execute** |
|---|---|---|---|---|
| Sealed | `0` | No | No | No |
| Sealed | `1` | No | No | No |
| Sealed-return | `0` | `cursor in [base + 3 * CLENBYTES, base + 34 * CLENBYTES - size]` | `cursor in [base + 3 * CLENBYTES, base + 34 * CLENBYTES - size]` | No |
| Sealed-return | `1` | No | No | No |

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

## 2.2. 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 confusion of the type 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` |
| 11 | `x11`/`a1` | `c11`/`ca1` |
| 12 | `x12`/`a2` | `c12`/`ca2` |
| 13 | `x13`/`a3` | `c13`/`ca3` |

| Index | XLEN-bit integer | Capability |
|---|---|---|
| 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.

# 2.3. Extension to Other Registers

## 2.3.1. Program Counter

The following changes are made to the program counter (pc):

- *Pure Capstone*: The program counter (pc) is extended to contain a capability only.
- *TransCapstone*: Similar to the general-purpose registers, the program counter (pc) is also extended to contain a capability or an integer.

**During the instruction fetch stage, an exception is raised when any of the following conditions are met (in priority order):**

*Pure Capstone*

- 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 - 4]`.
- Instruction address misaligned (0)
  - `pc.cursor` is not aligned to `4`.

*TransCapstone*

- `cwrld` is `1` (secure world) and any of the conditions for *Pure Capstone* are met.
- `cwrld` is `0` (normal world) and any of the conditions for RV64IZicsr are met.
- Instruction access fault (1)
  - `cwrld` is `0` (normal world) and `pc` does not contain an integer.

**If no exception is raised:**

- *Pure Capstone*: The instruction pointed to by `pc.cursor` is fetched and executed. The `pc.cursor` is then incremented by `4` (i.e., `pc.cursor += 4`).
- *TransCapstone*:
  - `cwrld` is `1` (secure world): Same as *Pure Capstone*.
  - `cwrld` is `0` (normal world): The instruction pointed to by `pc` is fetched and executed. The `pc` is then incremented by `4` (i.e., `pc += 4`).

## 2.4. 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.

In *Pure Capstone*, the memory can only be accessed through capabilities.

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

In *TransCapstone*, the physical memory is divided into two disjoint regions: the *normal memory* and the *secure memory*. While the normal memory is only accessible through MMU (Memory Management Unit), the secure memory can only be accessed through capabilities. The bounds of the secure memory `[SBASE, SEND)` are implementation-defined.

| Memory Region | Address Space | Access Method |
|---|---|---|
| *Normal Memory* | `[0, SBASE) U [SEND, 2^XLEN)` | MMU |
| *Secure Memory* | `[SBASE, SEND)` | Capabilities |

## 2.5. Added Registers

The Capstone-RISC-V ISA adds the following registers:

*Table 5. Additional Registers in Capstone-RISC-V ISA*

| Capstone Variant | Additional Registers | | | |
|---|---|---|---|---|
| *Pure Capstone* | **Mnemonic** | **CCSR encoding** | **CSR encoding** | **Description** |
| | `ceh` | `0x000` | - | The sealed capability or PC entry for the exception handler |
| | `cih` | `0x001` | - | The sealed capability for the interrupt handler |
| | `epc` | `0x002` | - | The exception program counter register |
| | `cinit` | `0x003` | - | The initial capability covering the entire address space of the memory |
| | `cis` | - | `0x800` | The interrupt status register |
| | `tval` | - | `0x801` | The exception data (trap value) register |
| | `cause` | - | `0x802` | The exception cause register |

<table>
<tr><th>Capstone Variant</th><th>Additional Registers</th></tr>
<tr><td>TransCapstone</td><td>
<table>
<tr><th>Mnemonic</th><th>CCSR encoding</th><th>CSR encoding</th><th>Description</th></tr>
<tr><td>ceh</td><td>0x000</td><td>-</td><td>The sealed capability or PC entry for the exception handler</td></tr>
<tr><td>epc</td><td>0x002</td><td>-</td><td>The exception program counter register</td></tr>
<tr><td>cinit</td><td>0x003</td><td>-</td><td>The initial capability covering the entire address space of the secure memory</td></tr>
<tr><td>emode</td><td>0x004</td><td>-</td><td>The encoding mode of the system. 0 = integer encoding mode, 1 = capability encoding mode</td></tr>
<tr><td>cwrld</td><td>-</td><td>-</td><td>The currently executed world. 0 = normal world, 1 = secure world</td></tr>
<tr><td>normal_pc</td><td>-</td><td>-</td><td>The program counter for the normal world before the secure world is entered</td></tr>
<tr><td>normal_sp</td><td>-</td><td>-</td><td>The stack pointer for the normal world before the secure world is entered</td></tr>
<tr><td>switch_reg</td><td>-</td><td>-</td><td>The index of the general-purpose register used when switching worlds</td></tr>
<tr><td>switch_cap</td><td>0x005</td><td>-</td><td>The capability used to store contexts when switching worlds asynchronously</td></tr>
<tr><td>exit_reg</td><td>-</td><td>-</td><td>The index of the general-purpose register for receiving the exit code when exiting the secure world</td></tr>
<tr><td>tval</td><td>-</td><td>0x801</td><td>The exception data (trap value) register</td></tr>
<tr><td>cause</td><td>-</td><td>0x802</td><td>The exception cause register</td></tr>
</table>
</td></tr>
</table>

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 CCSR manipulation instructions.

The manipulation constraints for each CCSR are indicated below.

*Table 6. Manipulation Constraints for CCSRs*

| Mnemonic | Read | Write |
|---|---|---|
| ceh | Pure Capstone or secure world | Pure Capstone or secure world |
| cih | - | Pure Capstone or secure world; the original content must not be a capability |

| Mnemonic | Read | Write |
|---|---|---|
| `cinit` | Pure Capstone or normal world; one-time only | - |
| `switch_cap` | Normal world | Normal world |

The manipulation constraints for each additional CSR are indicated below.

*Table 7. Manipulation Constraints for Additional CSRs*

| Mnemonic | Read | Write |
|---|---|---|
| `cis` | `cih` must not be a capability | `cih` must not be a capability |

**Note**

`ceh` and `cih` should be handled differently. `ceh` is about the functionality of a domain only. A domain should be allowed to set `ceh` 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.

**Note**

`cinit` is a special CCSR that is used to bootstrap capabilities after a system reset. CCSR manipulation instructions can be used to read this initial capability and store it in a general-purpose register. This operation can only be performed once after each reset. Any attempt to write `cinit` will be silently ignored, and any attempt to read it after the first time will return the content of `cnull`.

# 2.6. 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 two basic formats: R-type and I-type, as described below (more details are also provided in the *RISC-V ISA Manual*).

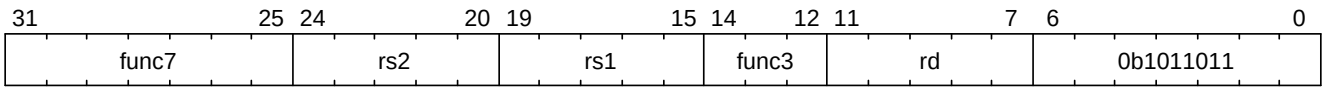


*Figure 1. R-type instruction format*

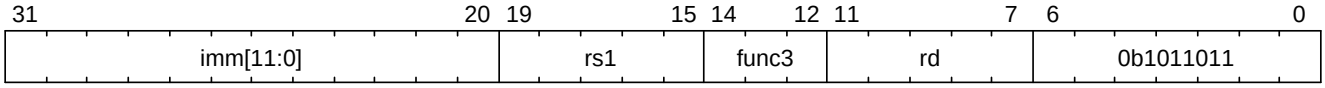


*Figure 2. I-type instruction format*

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

# 2.7. Reset

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

## 2.7.1. Pure Capstone

- Each general-purpose register either contains an integer, or a capability with `valid = 0`.
- No addressable memory location can contain a capability.
- `ceh`, `cih`, and `epc` contain either integers or capabilities with `valid = 0` (invalid).
- `cis = 0`.
- `cinit = { valid = 1, type = 0, cursor = INIT_DATA_BASE, base = INIT_DATA_BASE, end = INIT_DATA_END, perms = 7 }`, and `pc = { valid = 1, type = 0, cursor = INIT_CODE_BASE, base = INIT_CODE_BASE, end = INIT_CODE_END, perms = 7 }`, where `INIT_DATA_BASE`, `INIT_DATA_END`, `INIT_CODE_BASE`, and `INIT_CODE_END` are implementation-defined, and `[INIT_CODE_BASE, INIT_CODE_END)` does not overlap with `[INIT_DATA_BASE, INIT_DATA_END)`.

## 2.7.2. TransCapstone

- Each general-purpose register either contains an integer, or a capability with `valid = 0`.
- No addressable memory location can contain a capability.
- `ceh` contains either an integer or a capability with `valid = 0` (invalid).
- `cwrld = 0` (normal world).
- `cinit = { valid = 1, type = 0, cursor = SBASE, base = SBASE, end = SEND, perms = 7 }`.
- Specifications for RV64IZicsr.

# 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.

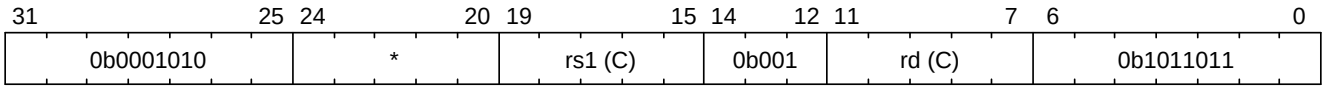


*Figure 3. MOVC instruction format*

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

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

**If no exception is raised:**

- 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 give amount (offset).

**CINCOFFSET**

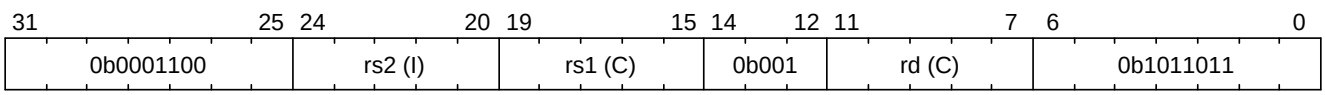


*Figure 4. CINCOFFSET instruction format*

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

- `Unexpected operand type (24)`

- `x[rs1]` is not a capability.
- `x[rs2]` is not an integer.

- `Unexpected capability type (26)`
  - `x[rs1]` does not have `type = 0` (linear) or `type = 1` (non-linear).

**If no exception is raised:**

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

**CINCOFFSETIMM**

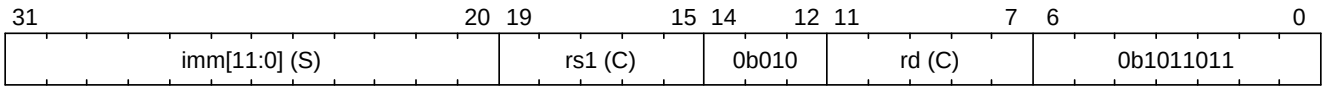


*Figure 5. CINCOFFSETIMM instruction format*

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

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Unexpected capability type (26)`
  - `x[rs1]` does not have `type = 0` (linear) or `type = 1` (non-linear).

**If no exception is raised:**

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

### 3.1.3. Cursor Setter

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

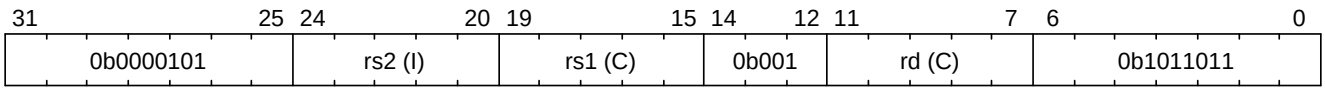


*Figure 6. SCC instruction format*

**An exception is raised if any of the following conditions are met:**

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

- x[rs2] is not an integer.

- Unexpected capability type (26)
  - x[rs1] does not have type = 0 (linear) or type = 1 (non-linear).

**If no exception is raised:**

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

### 3.1.4. Field Query

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

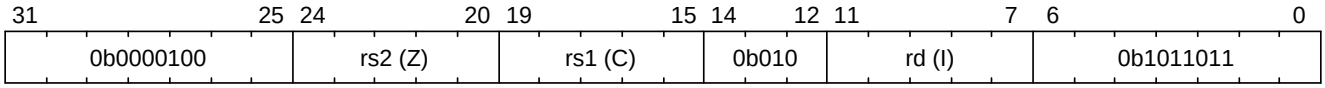


*Figure 7. LCC instruction format*

**An exception is raised if any of the following conditions are met:**

- Unexpected operand type (24)
  - x[rs1] is not a capability.
- Unexpected capability type (26)
  - rs2 = 2 and x[rs1] has type = 4 (sealed).
  - rs2 = 4 and x[rs1] has type = 4 (sealed), type = 5 (sealed-return), or type = 6 (exit).
  - rs2 = 5 and x[rs1] has type = 4 (sealed), type = 5 (sealed-return), or type = 6 (exit).
  - rs2 = 6 and x[rs1] does not have type = 4 (sealed) or type = 5 (sealed-return).
  - rs2 = 7 and x[rs1] does not have type = 5 (sealed-return).

**If no exception is raised:**

- If rs2 > 7, write zero to x[rd]
- Otherwise, write field to x[rd] according to the LCC multiplexing table.

*Table 8. LCC multiplexing table*

| rs2 | field |
|---|---|
| 0 | x[rs1].valid |
| 1 | x[rs1].type |
| 2 | x[rs1].cursor |

| rs2 | field |
|---|---|
| 3 | x[rs1].base |
| 4 | x[rs1].end |
| 5 | x[rs1].perms |
| 6 | x[rs1].async |
| 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.

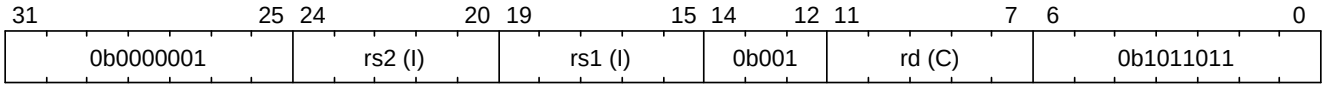


*Figure 8. SHRINK instruction format*

**An exception is raised when any of the following conditions are 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].type is 3 (uninitialised) and x[rd].cursor < x[rs1], set x[rd].cursor to x[rs1].

### 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).

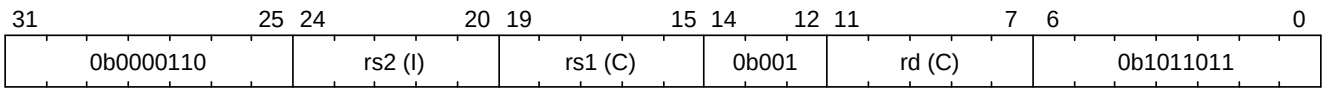


*Figure 9. SPLIT instruction format*

**An exception is raised when any of the following conditions are 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. Write `x[rs1]` to `x[rd]`.
2. Set `x[rs1].end` to `x[rs2]`.
3. Set `x[rd].base` to `x[rs2]`.

### 3.1.7. Permission Tightening

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

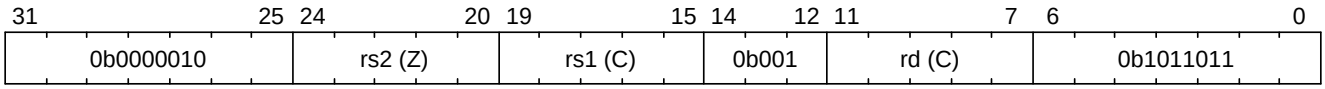


*Figure 10. TIGHTEN instruction format*

**An exception is raised when any of the following conditions are 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)`
  - `rs2 <= 7` and `rs2 <=p x[rs1].perms` does not hold.

**If no exception is raised:**

1. If `rs2 > 7`, set `x[rs1].perms` to `0`. Otherwise, set `x[rs1].perms` to `rs2`.

2. `MOVC rd, rs1`.

## 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.

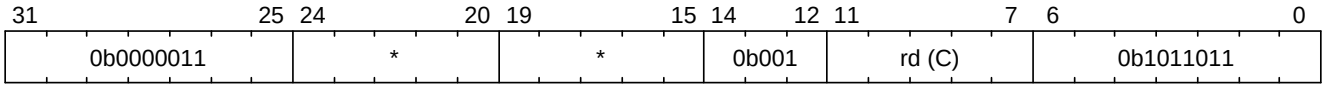


*Figure 11. DELIN instruction format*

**An exception is raised when any of the following conditions are 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).

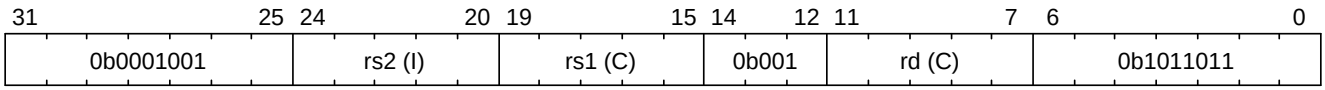


*Figure 12. INIT instruction format*

**An exception is raised when any of the following conditions are 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` and `x[rs1].end` are not equal.

**If no exception is raised:**

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

### 3.2.3. Sealing

The SEAL instruction seals a linear capability.

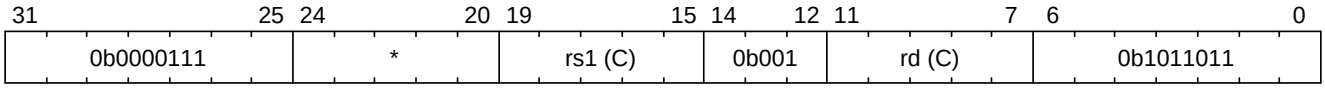


*Figure 13. SEAL instruction format*

**An exception is raised when any of the following conditions are 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.
- `Capability out of bound (28)`
  - The size of the memory region associated with `x[rs1]` is smaller than `CLENBYTES * 33` bytes (i.e., `x[rs1].end - x[rs1].base < CLENBYTES * 33`).

**If no exception is raised:**

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

## 3.3. Dropping

The DROP instruction invalidates a capability.

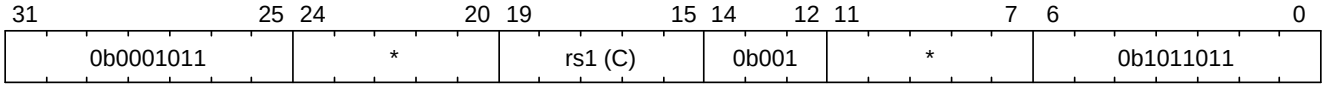


*Figure 14. DROP instruction format*

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

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

**If no exception is raised:**

- Set `x[rs1].valid` to `0` (invalid).

# 3.4. Revocation

## 3.4.1. Revocation Capability Creation

The MREV instruction creates a revocation capability.

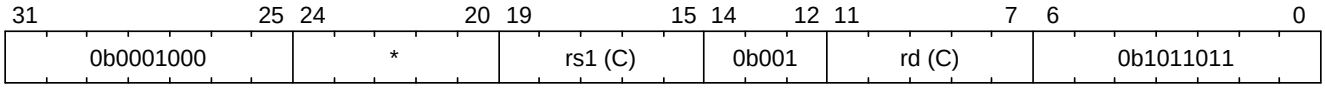


*Figure 15. MREV instruction format*

**An exception is raised when any of the following conditions are 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.

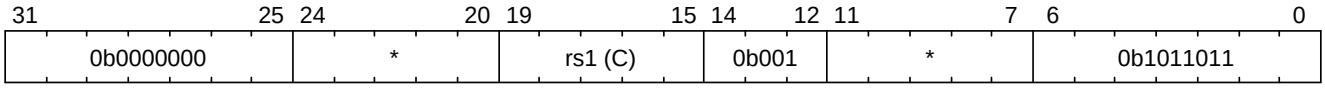


*Figure 16. REVOKE instruction format*

**An exception is raised when any of the following conditions are 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 any 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, for any invalidated capability `c`, at least one of the following conditions are met:
   - The type of `c` is non-linear (i.e., `c.type` is `1`).
   - `2 <=p c.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. *Pure Capstone*

In *Pure Capstone*, two specific instructions (i.e., LDC and LTC) are used to load and store capabilities.

### 4.1.1. Load Capabilities

The LDC instruction loads a capability from memory.

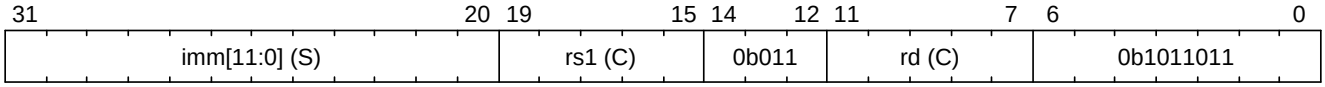


*Figure 17. LDC instruction format*

**An exception is raised when any of the following conditions are 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), `5` (sealed-return), or `6` (exit).
- 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` is not in the range `[x[rs1].base, x[rs1].end - CLENBYTES]`.
  - `x[rs1].type` is `5` (sealed-return) or `6` (exit) and `x[rs1].cursor` is not in the range `[x[rs1].base + 3 * CLENBYTES, x[rs1].base + 34 * CLENBYTES - CLENBYTES]`.
- Load address misaligned (4)
  - `x[rs1].cursor + imm` is not aligned to `CLEN` bits.
- Unexpected operand type (24) (TODO)
  - The data contained in the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + CLENBYTES)` is not a capability.
- Insufficient capability permissions (27)
  - The capability being loaded is not 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:** Load the capability at the memory location `[x[rs1].cursor + imm,`

`x[rs1].cursor + imm + CLENBYTES)` into `x[rd]`. If the capability being loaded is not a non-linear capability (i.e., `type != 1`), the data contained in the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + CLENBYTES)` will be set to the content of `cnull`.

### 4.1.2. Store Capabilities

The STC instruction stores a capability to memory.

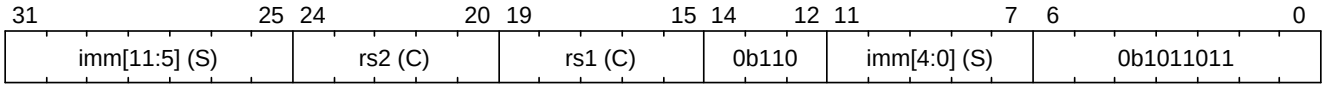


*Figure 18. STC instruction format*

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

- Unexpected operand type (24)
  - `x[rs1]` is not a capability.
  - `x[rs2]` is not a capability.
- Invalid capability (25)
  - `x[rs1].valid` is `0` (invalid).
- Unexpected capability type (26)
  - `x[rs1].type` is not `0`, `1`, `3`, `5`, or `6` (linear, non-linear, uninitialized, sealed-return, or exit).
- Illegal operand value (29)
  - `x[rs1].type` is `3` (uninitialised) and `imm` is not `0`.
- Insufficient capability permissions (27)
  - `x[rs1].type` is `0` or `1` and `2 ⇐p x[rs1].perms` does not hold.
- 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, x[rs1].base + 34 * CLENBYTES - CLENBYTES]`.
- Store/AMO address misaligned (6)
  - `x[rs1].cursor + imm` is not aligned to `CLEN` bits.

**If no exception is raised:** Store `x[rs2]` to the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + CLENBYTES)`. If `x[rs1]` is an uninitialised capability (i.e., `x[rs1].type` is `3`), `x[rs1].cursor` is set to `x[rs1].cursor + CLENBYTES`. If `x[rs2]` is not a non-linear capability (i.e., `type != 1`), `x[rs2]` will be set to the content of `cnull`.

## 4.2. *TransCapstone*

In *TransCapstone*, the LDC and STC instructions are extended to support loading and storing capabilities from/to the normal memory using raw addresses.

- In the secure world (i.e., `cwrld` is `1`), the LDC and STC instructions remain the same as in *Pure Capstone*.
- In the normal world (i.e., `cwrld` is `0`), the LDC and STC instructions behave differently in different *encoding modes*. When `emode` is `1` (capability encoding mode), the LDC and STC instructions behave the same as in *Pure Capstone*. When `emode` is `0` (integer encoding mode), the LDC and STC instructions are used to load and store capabilities from/to the normal memory using raw addresses.

### 4.2.1. Load Capabilities in integer encoding mode

When `cwrld` is `0` (normal world) and `emode` is `0` (integer encoding mode), the LDC instruction loads a capability from the normal memory using raw addresses.

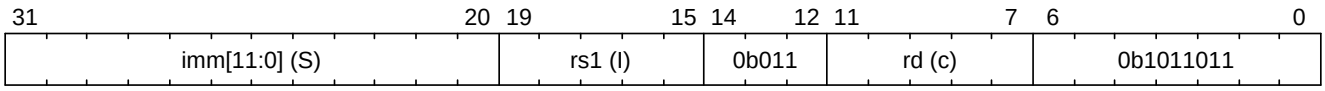


*Figure 19. LDC instruction format*

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

- Unexpected operand type (24)
  - `x[rs1]` is not an integer.
- Load address misaligned (4)
  - `x[rs1] + imm` is not aligned to `CLEN` bits.
- Load access fault (5)
  - `x[rs1] + imm` is in the range `[SBASE, SEND)`.
- Unexpected operand type (24)
  - The data contained in the memory location `[x[rs1] + imm, x[rs1] + imm + CLENBYTES)` is not a capability.

**If no exception is raised:** Load the capability at the memory location `[x[rs1] + imm, x[rs1] + imm + CLENBYTES)` into `x[rd]`. If the `type` field of the capability being loaded is neither `1` (non-linear) nor `6` (exit), the data contained in the memory location `[x[rs1] + imm, x[rs1] + imm + CLENBYTES)` will be set to `cnull`.

### 4.2.2. Store Capabilities in integer encoding mode

When `cwrld` is `0` (normal world) and `emode` is `0` (integer encoding mode), the STC instruction stores a capability to the normal memory using raw addresses.

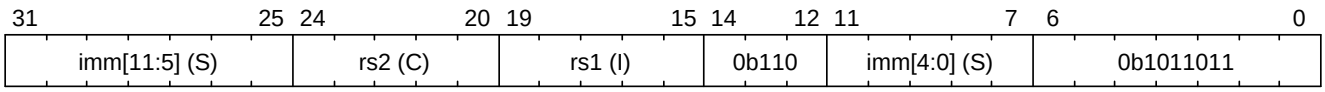


*Figure 20. STC instruction format*

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

- Unexpected operand type (24)

    - `x[rs1]` is not an integer.
    - `x[rs2]` is not a capability.
- Store/AMO address misaligned (6)
    - `x[rs1] + imm` is not aligned to `CLEN` bits.
- Store/AMO access fault (7)
    - `x[rs1] + imm` is in the range `[SBASE, SEND)`.

**If no exception is raised:** Store `x[rs2]` to the memory location `[x[rs1] + imm, x[rs1] + imm + CLENBYTES)`. If the `type` field of the capability being stored is neither `1` (non-linear) nor `6` (exit), `x[rs2]` will be set to `cnull`.

# 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

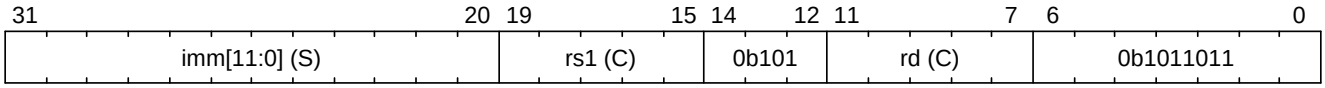


*Figure 21. CJALR instruction format*

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

*Pure Capstone*

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

*TransCapstone*

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

**If no exception is raised:**

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

### 5.1.2. CBNZ

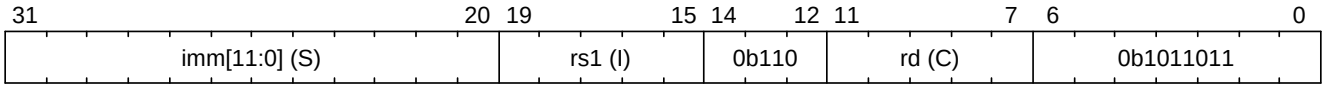


*Figure 22. CBNZ instruction format*

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

*Pure Capstone*

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

*TransCapstone*

- `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. Set `x[rd].cursor` to `x[rd].cursor + imm`.
  2. Write `x[rd]` to `pc`.
  3. If `x[rd].type != 1`, write `cnull` to `x[rd]`.

# 5.2. Domain Crossing

*Domains* in Capstone-RISC-V 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*.

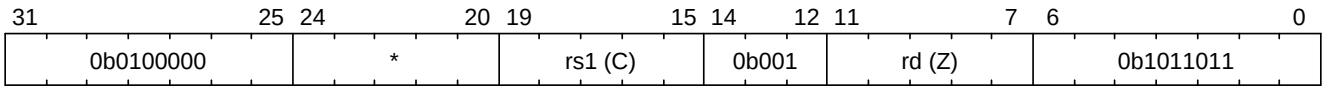


*Figure 23. CALL instruction format*

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

*TransCapstone*

- `Illegal instruction (2)`
  - `cwrld` is `0` (normal world).

*Pure Capstone* or *TransCapstone*

- `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 `4` (sealed).
  - `x[rs1].async` is not `0` (synchronous).

**If no exception is raised:**

1. `MOVC cra, rs1`.
2. Swap the program counter (`pc`) with the content at the memory location `[cra.base, cra.base + CLENBYTES)`.
3. Swap `ceh` with the content at the memory location `[cra.base + CLENBYTES, cra.base + 2 * CLENBYTES)`.
4. Swap `csp` with the content at the memory location `[cra.base + 2 * CLENBYTES, cra.base + 3 * CLENBYTES)`.
5. 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

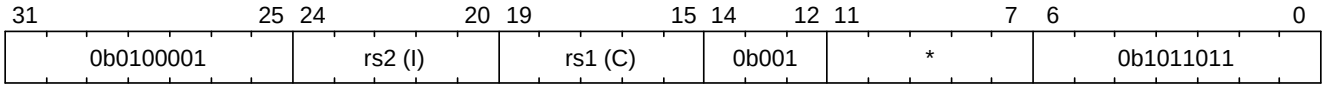


*Figure 24. RETURN instruction format*

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

*TransCapstone*

- `Illegal instruction (2)`
  - `cwrld` is `0` (normal world).

*Pure Capstone* or *TransCapstone*

- `Unexpected operand type (24)`
  - `rs1 != 0` and `x[rs1]` is not a capability.
  - `x[rs2]` is not an integer.
- `Invalid capability (25)`
  - `rs1 != 0` and `x[rs1].valid` is `0` (invalid).

- `Unexpected capability type (26)`
  - `rs1 != 0` and `x[rs1].type` is not `5` (sealed-return).

**If no exception is raised:**

**If `rs1 = 0`:**

1. Set `pc.cursor` to `x[rs2]`.
2. Write `epc` to `pc`, and `pc` to `ceh`.
3. If `epc.type != 1`, write `cnull` to `epc`.

**Otherwise:**

**When `x[rs1].async = 0` (synchronous):**

1. Write `x[rs1]` to `cap` and `cnull` to `x[rs1]`.
2. Set `pc.cursor` to `x[rs2]`, and swap the program counter (`pc`) with the content at the memory location `[cap.base, cap.base + CLENBYTES)`.
3. Swap `ceh` with the content at the memory location `[cap.base + CLENBYTES, cap.base + 2 * CLENBYTES)`.
4. Swap `csp` with the content at the memory location `[cap.base + 2 * CLENBYTES, cap.base + 3 * CLENBYTES)`.
5. Write `cap` to `x[cap.reg]` and set `x[cap.reg].type` to `4` (sealed).

**When `x[rs1].async = 1` (upon exception):**

1. Set `pc.cursor` to `x[rs2]`, and swap the program counter (`pc`) with the content at the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)`.
2. Store `ceh` to the memory location `[x[rs1].base + CLENBYTES, x[rs1].base + 2 * CLENBYTES)`.
3. Set `x[rs1].type` to `4` (sealed), `x[rs1].async` to `0` (synchronous).
4. Write the resulting `x[rs1]` to `ceh`, and `cnull` to `x[rs1]`.
5. For `i = 1, 2, ⋯, 31`, Swap `x[i]` with the content at the memory location `[ceh.base + (i + 1) * CLENBYTES, ceh.base + (i + 2) * CLENBYTES)`.

**When `x[rs1].async = 2` (upon interrupt):**

1. Set `pc.cursor` to `x[rs2]`, and swap the program counter (`pc`) with the content at the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)`.
2. Swap `ceh` with the content at the memory location `[x[rs1].base + CLENBYTES, x[rs1].base +`

`2 * CLENBYTES)`.

3. Set `x[rs1].type` to `4` (sealed), `x[rs1].async` to `0` (synchronous).
4. Write the resulting `x[rs1]` to `cih`, and `cnull` to `x[rs1]`.
5. For `i = 1, 2, ⋯, 31`, Swap `x[i]` with the content at the memory location `[cih.base + (i + 1) * CLENBYTES, cih.base + (i + 2) * CLENBYTES)`.

## 5.3. A World Switching Extension for *TransCapstone*

In *TransCapstone*, a pair of extra instructions, i.e., CAPENTER and CAPEXIT, is added to support switching between the *secure world* and the *normal world*.

### 5.3.1. CAPENTER

The CAPENTER instruction causes an entry into the secure world from the normal world. And it is only available in the normal world.

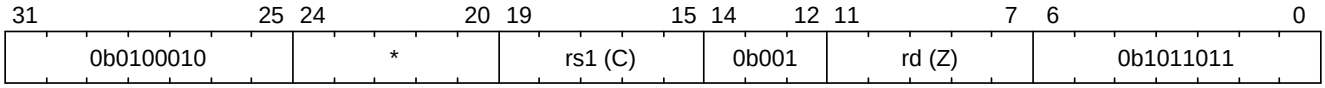


*Figure 25. CAPENTER instruction format*

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

- `Illegal instruction (0)`
  - `cwrld` is `1` (secure world).
- `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 `4` (sealed).

**If no exception is raised:**

**When `x[rs1].async = 0` (synchronous):**

1. `MOVC cra, rs1`.
2. Write `pc` and `sp` to `normal_pc` and `normal_sp` respectively.
3. Write `cnull` to `pc`, `csp` and `ceh`.
4. Swap the program counter (`pc`) with the content at the memory location `[cra.base, cra.base + CLENBYTES)`.

5. Swap `ceh` with the content at the memory location `[cra.base + CLENBYTES, cra.base + 2 * CLENBYTES)`.
6. Swap `csp` with the content at the memory location `[cra.base + 2 * CLENBYTES, cra.base + 3 * CLENBYTES)`.
7. Set `cra.type` to `6` (exit), `cra.cursor` to `cra.base`.
8. Write `rs1` to `switch_reg`, `rd` to `exit_reg`.
9. Set `cwrld` to `1` (secure world).

**When `x[rs1].async` is `1` (upon exception) or `2` (upon interrupt):**

1. Write `x[rs1]` to `switch_cap`, and `cnull` to `x[rs1]`.
2. Write `pc` and `sp` to `normal_pc` and `normal_sp` respectively.
3. Write `cnull` to `pc` and `ceh`.
4. Swap the program counter (`pc`) with the content at the memory location `[cra.base, cra.base + CLENBYTES)`.
5. Swap `ceh` with the content at the memory location `[cra.base + CLENBYTES, cra.base + 2 * CLENBYTES)`.
6. For `i = 1, 2, ⋯, 31`, write `zero` to `x[i]`, and then swap `x[i]` with the content at the memory location `[switch_cap.base + (i + 1) * CLENBYTES, switch_cap.base + (i + 2) * CLENBYTES)`.
7. Set `switch_cap.async` to `0` (synchronous).
8. Write `rs1` to `switch_reg`, `rd` to `exit_reg`.
9. Set `cwrld` to `1` (secure world).

▼ **Note: the purpose of the `rd` operand**

> The `rd` register will be set to a value indicating the cause of exit when the CPU core exits from the secure world synchronously or asynchronously.

### 5.3.2. CAPEXIT

The CAPEXIT instruction causes an exit from the secure world into the normal world. It is only available in the secure world and can only be used with an exit capability.

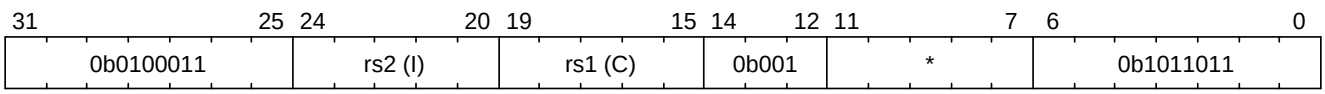


*Figure 26. CAPEXIT instruction format*

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

- `Illegal instruction (2)`
  - `cwrld` is `0` (normal world).
- `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 `6` (exit).

**If no exception is raised:**

1. Write `x[rs1]` to `cap`, and `cnull` to `x[rs1]`.
2. Set `pc.cursor` to `x[rs2]`, and write `pc`, `ceh`, and `csp` to the memory location `[cap.base, cap.base + CLENBYTES)`, `[cap.base + CLENBYTES, cap.base + 2 * CLENBYTES)`, and `[cap.base + 2 * CLENBYTES, cap.base + 3 * CLENBYTES)` respectively.
3. Write the content of `normal_pc` and `normal_sp` to `pc` and `sp` respectively.
4. Write `cnull` to `ceh`.
5. Set `cap.type` to `4` (sealed), `cap.async` to `0` (synchronous), and write the resulting `cap` to `x[switch_reg]`.
6. Set `x[exit_reg]` to `0` (normal exit).
7. Set `cwrld` to `0` (normal world).

# 6. Control and Status Instructions

The CCSRRW instruction is used to read and write specified capability CSRs (CCSRs).

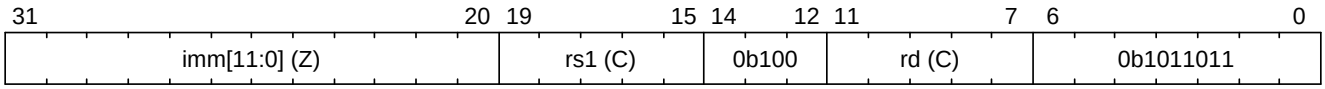


*Figure 27. CCSRRW instruction format*

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

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

**If no exception is raised:**

1. Read from capability CSR
   - If the read constraint is satisfied, the content of the capability CSR specified by the immediate value `imm` is written to `x[rd]`. If the current content of the capability CSR is neither a non-linear capability (i.e., `type != 1`) nor an exit capability (i.e., `type != 6`), it will be set to the content of `cnull`.
   - Otherwise, `x[rd]` is set to the content of `cnull`.
2. Write to capability CSR
   - If the write constraint is satisfied, `x[rs1]` is written to the capability CSR specified by the immediate value `imm`. If `x[rs1]` is neither a non-linear capability (i.e., `type != 1`) nor an exit capability (i.e., `type != 6`), it will be set to the content of `cnull`.
   - Otherwise, the original current of the capability CSR is preserved.

# 7. Adjustments to Existing Instructions

For most existing instructions in RV64IZicsr, the adjustments are straightforward. Their behaviour is unchanged, and an "unexpected operand type (24)" exception is raised if any of the operands (i.e., `x[rs1]`, `x[rs2]` or `x[rd]`) is a capability. Apart from this operand constraint, the following instructions in RV64IZicsr are adjusted in Capstone:

- For memory access instructions, they are extended to use capabilities as addresses for memory access.
- For control flow instructions, they are slightly adjusted to support capability-aware control flow.
- Certain instructions, especially those belonging to the privileged ISA, are illegal under certain circumstances.

## 7.1. Memory Access Instructions

In RV64IZicsr, memory access instructions include load instructions (i.e., `lb`, `lh`, `lw`, `lbu`, `lhu`, `lwu`, `ld`), and store instructions (i.e., `sb`, `sh`, `sw`, `sd`). In RV64IZicsr, 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.

### 7.1.1. *Pure Capstone*

#### Load Instructions

In *Pure Capstone*, RV64IZicsr load instructions are modified to load integers of different sizes using capabilities.

##### Signed Load Instructions

`lb`, `lh`, `lw`, `ld` instructions are modified to load an integer in the size of byte, halfword, word, and doubleword (i.e., `XLENBYTES/8`, `XLENBYTES/4`, `XLENBYTES/2`, and `XLENBYTES` bytes), using capabilities respectively. The loaded integer is sign-extended to the size of the destination register (i.e., `XLEN` bits).

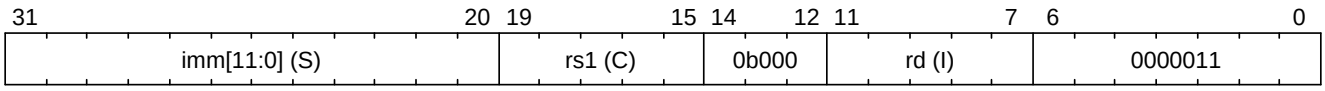


*Figure 28. lb instruction format (I-type)*

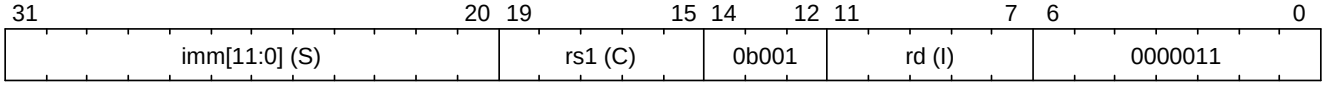


*Figure 29. lh instruction format (I-type)*

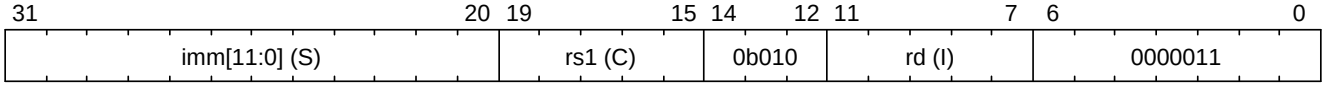


*Figure 30. lw instruction format (I-type)*

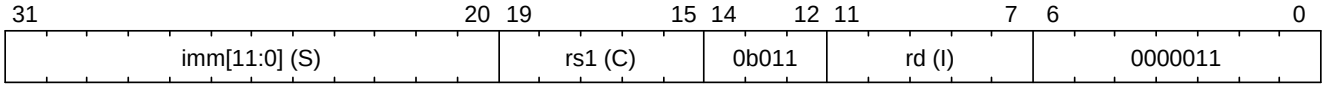


*Figure 31. ld instruction format (I-type)*

**An exception is raised when any of the following conditions are 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 neither `0` (linear) nor `1` (non-linear).
- Insufficient capability permissions (27)
  - `4 <=p x[rs1].perms` does not hold.
- Capability out of bound (28)
  - `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - size]`, where `size` is the size (in bytes) of the integer being loaded.
- Load address misaligned (4)
  - `x[rs1].cursor + imm` is not aligned to the size of the integer being loaded.

**If no exception is raised:** The content at the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + size)` is loaded as a signed integer to `x[rd]`, where `size` is the size of the integer being loaded (i.e., `XLENBYTES/8`, `XLENBYTES/4`, `XLENBYTES/2`, and `XLENBYTES` bytes for `lb`, `lh`, `lw`, and `ld` respectively).

#### Unsigned Load Instructions

`lbu`, `lhu`, `lwu` instructions are modified to load an integer in the size of byte, halfword, and word (i.e., `XLENBYTES/8`, `XLENBYTES/4`, and `XLENBYTES/2` bytes), using capabilities respectively. The loaded integer is zero-extended to the size of the destination register (i.e., `XLEN` bits).

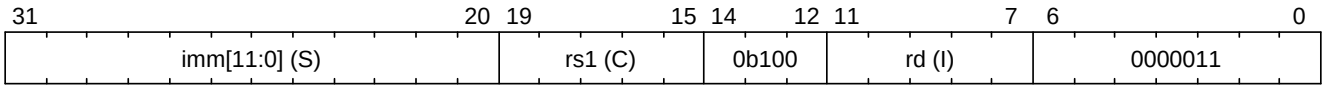


*Figure 32. lbu instruction format (I-type)*

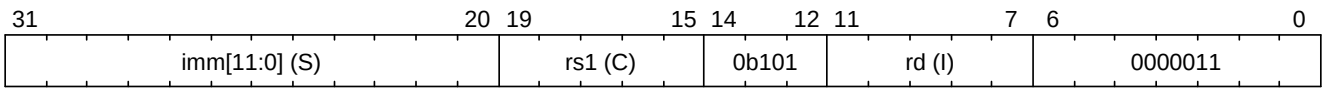


*Figure 33. lhu instruction format (I-type)*

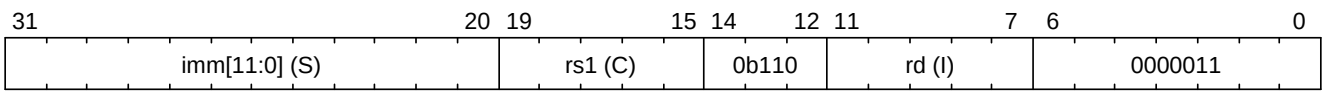


*Figure 34. lwu instruction format (I-type)*

**An exception is raised when any of the following conditions are 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 neither `0` (linear) nor `1` (non-linear).
- Insufficient capability permissions (27)
  - `4 <=p x[rs1].perms` does not hold.
- Capability out of bound (28)
  - `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - size]`, where `size` is the size (in bytes) of the integer being loaded.
- Load address misaligned (4)
  - `x[rs1].cursor + imm` is not aligned to the size of the integer being loaded.

**If no exception is raised:** The content at the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + size)` is loaded as an unsigned integer to `x[rd]`, where `size` is the size of the integer being loaded (i.e., `XLENBYTES/8`, `XLENBYTES/4`, and `XLENBYTES/2` bytes for `lbu`, `lhu`, and `lwu` respectively).

#### Store Instructions

`sb`, `sh`, `sw`, `sd` instructions are modified to store an integer in the size of byte, halfword, word, and doubleword (i.e., `XLENBYTES/8`, `XLENBYTES/4`, `XLENBYTES/2`, and `XLENBYTES` bytes), using capabilities respectively.

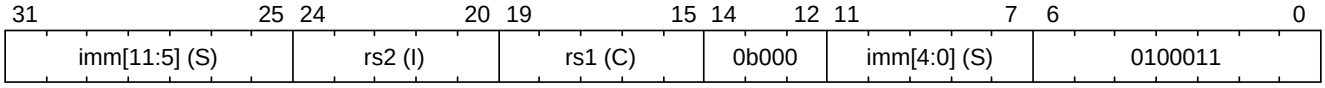


*Figure 35. sb instruction format (S-type)*

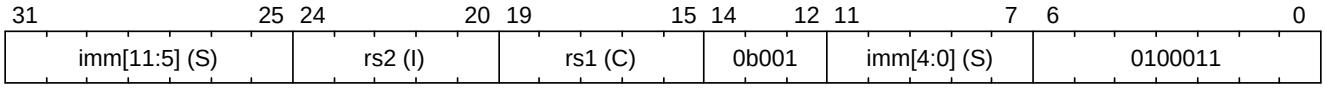


*Figure 36. sh instruction format (S-type)*

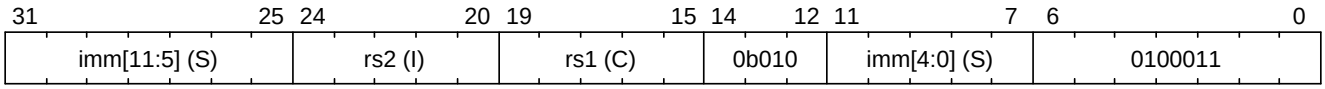


*Figure 37. sw instruction format (S-type)*

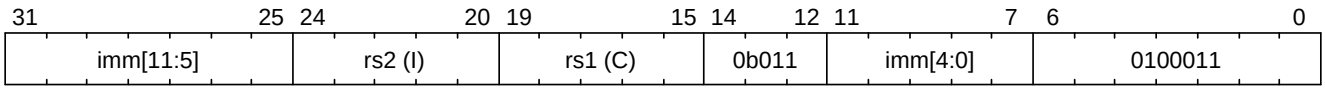


*Figure 38. sd instruction format (S-type)*

**An exception is raised when any of the following conditions are 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`, `1`, or `3` (linear, non-linear, or uninitialised).
- Illegal operand value (29)
    - `x[rs1].type` is `3` (uninitialised) and `imm` is not `0`.
- Insufficient capability permissions (27)
    - `x[rs1].perms` is neither `6` (read-write) nor `7` (read-write-execute).
- Capability out of bound (28)
    - `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - size]`, where `size` is the size (in bytes) of the integer being stored.
- Store/AMO address misaligned (6)
    - `x[rs1].cursor + imm` is not aligned to the size of the integer being stored.

**If no exception is raised:**

1. The content of `x[rs2]` is stored as an integer to the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + size)`, where `size` is the size of the integer being stored (i.e., `XLENBYTES/8`, `XLENBYTES/4`, `XLENBYTES/2`, and `XLENBYTES` bytes for `sb`, `sh`, `sw`, and `sd` respectively).
2. If `x[rs1].type` is `3` (uninitialised), `x[rs1].cursor` is set to `x[rs1].cursor + size`.
3. The content in the `CLEN`-bit aligned memory location `[cbase, cend)`, which alias with memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + size)`, is set to integer type, where `cbase = (x[rs1].cursor + imm) & ~(CLENBYTES - 1)` and `cend = cbase + CLENBYTES`.

### 7.1.2. *TransCapstone*

In *TransCapstone*, RV64IZicsr memory access instructions behave the same as in *Pure Capstone* if `cwrld` is `1` (secure world). However, if `cwrld` is `0` (normal world), these instructions behave differently in different *encoding modes*.

- When `cwrld` is `0` (normal world) and `emode` is `1` (capability encoding mode), these instructions behave the same as in *Pure Capstone*.
- When `cwrld` is `0` (normal world) and `emode` is `0` (integer encoding mode), these instructions behave the same as in RV64IZicsr except that the following adjustments are made to these instructions:
    - An “Unexpected operand type (24)” exception is raised if any of `x[rs1]`, `x[rs2]` and `x[rd]` contains a capability.
    - An “Capability out of bound (28)” exception is raised if the address to be accessed (i.e., `x[rs1] + imm`) is within the range `(SBASE - size, SEND)`, where `size` is the size (in bytes) of the integer to be loaded/stored.

- For store instructions (i.e., `sb`, `sh`, `sw`, `sd`), the content in the `CLEN`-bit aligned memory location `[cbase, cend)`, which alias with memory location `[x[rs1] + imm, x[rs1] + imm + size)`, is set to integer type, where `cbase = (x[rs1] + imm) & ~(CLENBYTES - 1)` and `cend = cbase + CLENBYTES`.

**Note**

In Capstone-RISC-V, when using these instructions to access the memory location that does not contain an integer-type value, the result of the following operations is undefined:

- Load an integer from the memory location.
- Store an integer to the memory location and then load an integer from the rest of the `CLEN`-bit aligned memory location.

# 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.

## 7.2.1. Branch Instructions

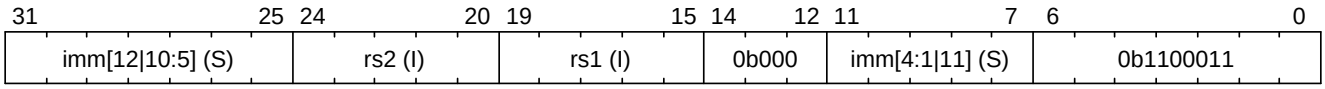


*Figure 39. beq instruction format (B-type)*

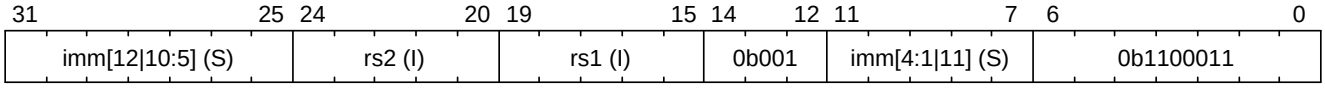


*Figure 40. bne instruction format (B-type)*

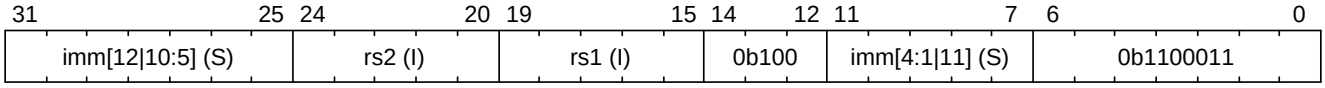


*Figure 41. blt instruction format (B-type)*

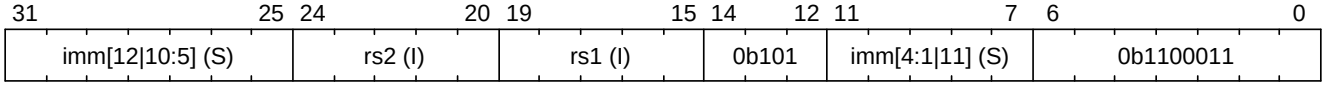


*Figure 42. bge instruction format (B-type)*

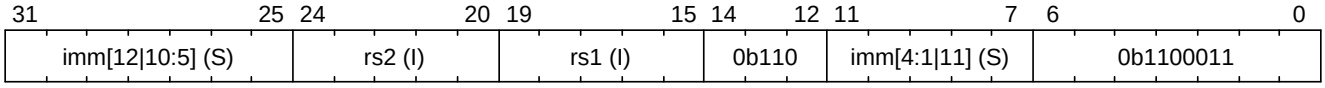


*Figure 43. bltu instruction format (B-type)*

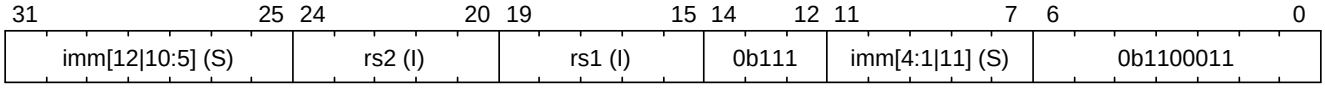


*Figure 44. bgeu instruction format (B-type)*

**The following adjustments are made to these instructions:**

*Pure Capstone*

- An “Unexpected operand type (24)” exception is raised if `x[rs1]` or `x[rs2]` is a capability.
- `pc.cursor`, instead of `pc`, is changed by the instruction.

*TransCapstone*

- An “Unexpected operand type (24)” exception is raised if `x[rs1]` or `x[rs2]` is a capability.
- When `cwrld` is `1` (secure world), `pc.cursor`, instead of `pc`, is changed by the instruction.

### 7.2.2. Jump Instructions

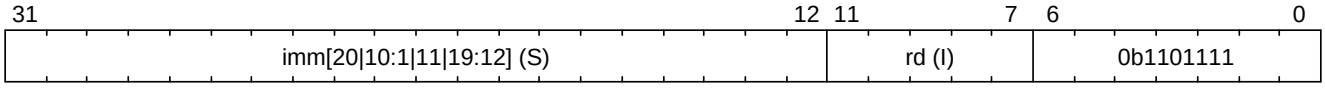


*Figure 45. jal instruction format (J-type)*

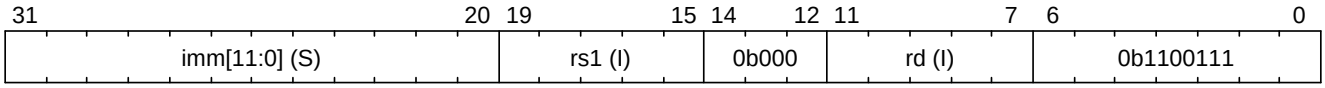


*Figure 46. jalr instruction format (I-type)*

**The following adjustments are made to these instructions:**

*Pure Capstone*

- An “Unexpected operand type (24)” exception is raised if `x[rs1]` (if existed) or `x[rd]` is a capability.
- `pc.cursor`, instead of `pc`, is changed by the instruction.
- `pc.cursor` (i.e., the address of the next instruction) is written to `x[rd]`.

*TransCapstone*

- An “Unexpected operand type (24)” exception is raised if `x[rs1]` (if existed) or `x[rd]` is a capability.
- When `cwrld` is `1` (secure world), `pc.cursor`, instead of `pc`, is changed by the instruction.
- When `cwrld` is `1` (secure world), former `pc.cursor` (i.e., the address of the next instruction) is written to `x[rd]`.

## 7.3. Instructions Made Illegal

Some instructions in RV64IZicsr now raise “illegal instruction (2)” exceptions when executed in

Pure Capstone or TransCapstone secure world, under all or some circumstances.

- All instructions defined in the privileged ISA of RV64IZicsr
- All instructions defined in the Zicsr extension, namely instructions that directly access CSRs, when the CSR specified is not one defined in Capstone-RISC-V, or when the read/write constraints are not satisfied
- ECALL
- EBREAK

# 8. Interrupts and Exceptions

## 8.1. Exception and Exit Codes

▼ **Note: where are the *exception codes* relevant?**

> For *Pure Capstone*, there is only one place where exception codes are relevant, which is the argument to pass to the *exception handler domain*.
>
> For *TransCapstone*, however, there are three places where we need to consider:
>
> 1. **Handleable Exception:** The argument to pass to the *exception handler domain*.
> 2. **Unhandleable Exception:** The value returned to the CAPENTER instruction in the user process.
> 3. **Interrupt:** The exception code that the OS sees.
>
> The argument passed to the *exception handler domain* will be in the register `cra` and `a0`, and the exit code the user process receives will be in the register specified by `exit_reg`.

The *exception code* is what the *exception handler domain* receives as an argument when an exception occurs on *Pure Capstone* or in *TransCapstone* secure world. It is an integer value that indicates what the type of the exception is.

*TransCapstone* also has *exit codes*, which are the values returned to the CAPENTER instruction in case the exception cannot be handled in the secure world.

We define the exception code and the exit code for each type of exception below. It aligns with the exception codes defined in RV64IZicsr, where applicable, for ease of implementation and interoperability.

*Table 9. Exception codes and exit codes for Pure Capstone and TransCapstone secure world*

| Exception | Exception code | TransCapstone exit code |
|---|---|---|
| Instruction address misaligned | 0 | 1 |
| Instruction access fault | 1 | 1 |
| Illegal instruction | 2 | 1 |
| Breakpoint | 3 | 1 |
| Load address misaligned | 4 | 1 |
| Load access fault | 5 | 1 |
| Store/AMO address misaligned | 6 | 1 |
| Store/AMO access fault | 7 | 1 |
| Unexpected operand type | 24 | 1 |
| Invalid capability | 25 | 1 |

| Exception | Exception code | TransCapstone exit code |
|---|---|---|
| Unexpected capability type | 26 | 1 |
| Insufficient capability permissions | 27 | 1 |
| Capability out of bound | 28 | 1 |
| Illegal operand value | 29 | 1 |
| Unhandleable exception | 63 | N/A in *TransCapstone* |

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

▼ **Note: *TransCapstone* exit code**

> Currently, we use the same exit code `1` for all exception types to protect the confidentiality of the secure world execution.

## 8.2. Exception Data

For *Pure Capstone* and the secure world in *TransCapstone*, the exception-related data is stored in the `tval` CSR, similar to RV64IZicsr. The exception handler can use the value to decide how to handle the exception. However, such data is available **only** for in-domain exception handling, where the exception handling process does not involve a domain switch.

For exception handling that crosses domain (i.e., when `ceh` is a valid sealed capability) or world boundaries (i.e., when the normal world ends up handling the exception), the exception data (i.e., the data in `tval`) is not available. This is to protect the confidentiality of domain execution. Note that this design does not stop the excepted domain from selectively trusting a different domain with such data.

For exceptions defined in RV64IZicsr, the same data as in it is written to `tval`. For the added exceptions, the following data is written to `tval`:

*Table 10. Exception data for Pure Capstone and TransCapstone secure world*

| Exception | Data |
|---|---|
| `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) |

| Exception | Data |
|---|---|
| `Illegal operand value (29)` | The instruction itself (or the lowest XLEN bits if it is wider than XLEN) |
| `Unhandleable exception (63)` | N/A |

## 8.3. *Pure Capstone*

For *Pure Capstone*, the handling of interrupts and exceptions is relatively straightforward. Regardless of whether the event is an interrupt or an exception (and what the type of the interrupt or exception is), the processor core will always transfer the control flow to the corresponding handler domain (specified in the `ceh` register for exceptions and the `cih` register for interrupts).

The current context is saved and sealed in a sealed-return capability which is then supplied to the exception/interrupt handler domain as an argument.

When exception/interrupt handling is complete, the exception/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.

### 8.3.1. Interrupt Status

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

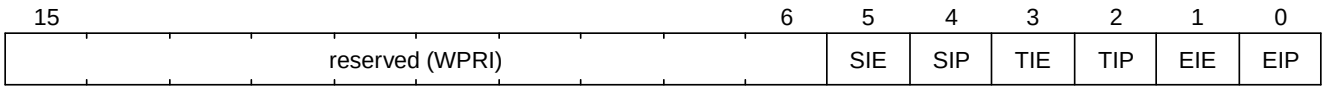


*Figure 47. `cis` CSR layout*

Each pair of `xIP` and `xIE` fields describes the status of the interrupt type `x`. The interrupt type `x` is pending if the `xIP` field is set to `1`, and enabled if the `xIE` field is set to `1`. Currently, three types of interrupts are supported: external interrupts (`E`), timer interrupts (`T`), and software interrupts (`S`). The definitions for those interrupt types match those in RV64IZicsr.

All the fields are read-write, but only when `cih` contains a capability.

▼ **Note: why not require a valid sealed capability?**

> We can require that the fields in `cis` are read-write only when `cih` contain a *valid sealed* capability, but that would be more costly than a simple check of the type of data in `cih`.

### 8.3.2. Interrupt Delivery

The interrupt delivery process starts with a certain event typically asynchronous to the execution of the hardware thread. The sources of such events include the external interrupt controller, the timer, and other CPU cores, which correspond to the external, timer, and software 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 interrupt is pending.

At any point during the execution of a hardware thread, if any pair of `xIP` and `xIE` fields are both `1` and at the same time the `cih` register contains a capability, the interrupt is delivered to the interrupt handler domain.

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

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

### 8.3.3. Handling of Interrupts

**The 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).

**Otherwise:**

1. Swap `pc` with the content at the memory location `[cih.base, cih.base + CLENBYTES)`.
2. Swap `ceh` with the content at the memory location `[cih.base + CLENBYTES, cih.base + 2 * CLENBYTES)`.
3. For `i = 1, 2, ⋯, 31`, Swap `x[i]` with the content at memory location `[cih.base + (i + 1) * CLENBYTES, cih.base + (i + 2) * CLENBYTES)`.
4. Set `cih.type` to `5` (sealed-return), `cih.cursor` to `cih.base`, `cih.reg` to `0`, and `cih.async` to `2` (upon interrupt).
5. Write `cih` to the register `cra`, and `cnull` to the register `cih`.
6. Write the exception code to the register `a0`.

### 8.3.4. Handling of Exceptions

▼ **Note: the stack of exception handler domains**

> Allowing anyone to set `ceh` can lead to DoS (when `ceh` is set to invalid values). Ideally, there should be a stack of exception handlers. Each domain can only choose to push extra exception handlers onto the stack. The bottom one will be provided by the kernel which is responsible for the liveness of the system.
>
> As this can be costly to implement, we limit the size of the stack to 2 for now, with the bottom one provided by the interrupt handler domain `cih`.

> Exceptions seem to be the dual of interrupts. Interrupt handling should be delegated bottom-up, while exception handling should be delegated top-down.

**Follow the interrupt handling procedure with exception code `unhandleable exception (63)` if any of the following conditions is met:**

- The `ceh` register does not contain a capability.
- The capability in `ceh` is invalid (`valid = 0`).
- The capability in `ceh` is not a sealed (`type != 4`), linear (`type != 0`), or non-linear capability (`type != 1`).

**Otherwise:**

**If the content in `ceh` is a valid sealed capability:**

1. Swap `pc` with the content at the memory location `[ceh.base, ceh.base + CLENBYTES)`.
2. For `i = 1, 2, ⋯, 31`, Swap `x[i]` with the content at the memory location `[ceh.base + (i + 1) * CLENBYTES, ceh.base + (i + 2) * CLENBYTES)`.
3. Set `ceh.type` to `5` (sealed-return), `ceh.cursor` to `ceh.base`, `ceh.reg` to `0`, and `ceh.async` to `1` (upon exception).
4. Write `ceh` to the register `cra`, and `cnull` to the register `ceh`.
5. Swap `ceh` with the content at the memory location `[cra.base + CLENBYTES, cra.base + 2 * CLENBYTES)`.
6. Write the exception code to the register `a0`.

**If the content is `ceh` is a valid *executable* non-linear capability or linear capability:**

1. Write `pc` to `epc`.
2. Write `ceh` to `pc`. If `ceh.type != 1`, write `cnull` to `ceh`.
3. Write the exception code to `cause`.
4. Write extra exception data to `tval`.

**Otherwise, the CPU core enters the state of *panic*.**

▼ **Note: sealing mechanism of in-domain exception handling**

> As the exception handler is in the same domain as the code that caused the exception, it is not necessary to seal the content of `csp` (or any other general purpose registers), or otherwise prevent the excepted code from accessing it.

### 8.3.5. Panic

When a CPU core is unable to handle an exception, it enters a state called *panic*. The actual behaviour of the CPU core in this state is implementation-defined, but must be one of the following:

- Reset.
- Enter an infinite loop.
- Scrub all general-purpose registers, and then load a capability that is not otherwise available into `pc`, and a set of capabilities that are not otherwise available into general-purpose registers.

The aim of the constraints above is to uphold the invariants of the capability model and in turn the security guarantees of the system.

## 8.4. *TransCapstone*

*TransCapstone* retains the same interrupt and exception handling mechanism for the normal world as in RV64IZicsr. For the secure world in *TransCapstone*, the handling of interrupts and exceptions is more complex, and it becomes relevant whether the event is an interrupt or an exception.

▼ **Note: overview of interrupt handling in the secure world**

> For interrupts, in order to prevent denial-of-service attacks by the secure world (e.g. a timer interrupt), the processor core needs to always transfer the control back to the normal world safely.
>
> The interrupt will be translated to one in the normal world that occurs at the CAPENTER instruction used to enter the secure world.
>
> Since interrupts are typically relevant only to the management of system resources, the interrupt should be transparent to both the secure world and the user process in the normal world. In other words, the secure world will simply resume execution from where it was interrupted after the interrupt is handled by the normal-world OS.

▼ **Note: overview of exception handling in the secure world**

> For exceptions, we want to give the secure world the chance to handle them first. If the secure world manages to handle the exception, the normal world will not be involved. The end result is that the whole exception or its handling is not even visible to the normal world.
>
> If the secure world fails to handle an exception (i.e., when it would end up panicking in the case of *Pure Capstone*, such as when `ceh` is not a valid sealed capability), however, the normal world will take over.
>
> The exception will **not** be translated into an exception in the normal world, but instead indicated in the *exit code* that the CAPENTER instruction in the user process receives. The user process can then decide what to do based on the exit code (e.g., terminate the domain

in the secure world).

Below we discuss the details of the handling of interrupts and exceptions generated in the secure world.

### 8.4.1. Handling of Secure-World Interrupts

When an interrupt occurs in the secure world, the processor core directly saves the full context, scrubs it, and exits to the normal world. It then generates a corresponding interrupt in the normal world, and follows the normal-world interrupt handling process thereafter.

**If the content in `switch_cap` is a valid sealed capability:**

1. Store `pc` to the memory location `[switch_cap.base, switch_cap.base + CLENBYTES)`.
2. Store `ceh` to the memory location `[switch_cap.base + CLENBYTES, switch_cap.base + 2 * CLENBYTES)`, and write `cnull` to `ceh`.
3. For `i = 1, 2, ···, 31`, store the content of `x[i]` to the memory location `[switch_cap.base + (i + 1) * CLENBYTES, switch_cap.base + (i + 2) * CLENBYTES)`.
4. Load the program counter `pc` and the stack pointer `sp` from `normal_pc` and `normal_sp` respectively.
5. Set `switch_cap.aync` to `2` (upon interrupt).
6. Write `switch_cap` to the register `x[switch_reg]`, and `cnull` to `switch_cap`.
7. Scrub the other general-purpose registers (i.e., write `zero` to `x[i]` where `i != 2` and `i != switch_reg`).
8. Set the `cwrld` register to `0` (normal world).
9. Trigger an interrupt in the normal world.

**Otherwise:**

1. Load the program counter `pc` and the stack pointer `sp` from `normal_pc` and `normal_sp` respectively.
2. Write `cnull` to `x[switch_reg]`.
3. Scrub the other general-purpose registers (i.e., write `zero` to `x[i]` where `i != 2` and `i != switch_reg`).
4. Set the `cwrld` register to `0` (normal world).
5. Trigger an interrupt in the normal world.

Note that in this case, there will be another exception in the normal world when the user process resumes execution after the interrupt has been handled by the OS, due to the invalid `switch_cap` value written to the CAPENTER operand.

### 8.4.2. Handling of Secure-World Exceptions

When an exception occurs, the processor core first attempts to handle the exception in the secure world, in the similar way as in *Pure Capstone*. If this fails, the processor core saves the full context if it can and exits to the normal world with a proper error code.

**If the content in `ceh` is a valid sealed capability:**

1. Swap `pc` with the content at memory location `[ceh.base, ceh.base + CLENBYTES)`.
2. For `i = 1, 2, ⋯, 31`, Swap `x[i]` with the content at the memory location `[ceh.base + (i + 1) * CLENBYTES, ceh.base + (i + 2) * CLENBYTES)`.
3. Set the `ceh.type` to `5` (sealed-return), `ceh.cursor` to `ceh.base`, and `ceh.async` to `1` (upon exception).
4. Write `ceh` to the register `cra`, and `cnull` to the register `ceh`.
5. Swap `ceh` with the content at the memory location `[cra.base + CLENBYTES, cra.base + 2 * CLENBYTES)`.
6. Write the exception code to the register `a0`.

Note that this is exactly the same as the handling of exceptions in *Pure Capstone*.

**If the content is `ceh` is a valid *executable* non-linear capability or linear capability:**

1. Write `pc` to `epc`.
2. Write `ceh` to `pc`. If `ceh.type != 1`, write `cnull` to `ceh`.
3. Write the exception code to `cause`
4. Write extra exception data to `tval`.

**Otherwise:**

**If the content in `switch_cap` is a valid sealed capability:**

1. Store the current value of the program counter (`pc`) to the memory location `[switch_cap.base, switch_cap.base + CLENBYTES)`.
2. Store `ceh` to the memory location `[switch_cap.base + CLENBYTES, switch_cap.base + 2 * CLENBYTES)`, and write `cnull` to `ceh`.
3. For `i = 1, 2, ⋯, 31`, store the content of `x[i]` to the memory location `[switch_cap.base + (i + 1) * CLENBYTES, switch_cap.base + (i + 2) * CLENBYTES)`.
4. Load the program counter `pc` and the stack pointer `sp` from `normal_pc` and `normal_sp` respectively.
5. Set `switch_cap.async` to `1` (upon exception).
6. Write the content of `switch_cap` to `x[switch_reg]`, and `cnull` to `switch_cap`.

7. Scrub the other general-purpose registers (i.e., write `zero` to `x[i]` where `i != 2` and `i != switch_reg`).
8. Write the exit code to `x[exit_reg]`.
9. Set the `cwrld` register to `0` (normal world).

**Otherwise:**

1. Load the program counter `pc` and the stack pointer `sp` from `normal_pc` and `normal_sp` respectively.
2. Write `cnull` to `x[switch_reg]`.
3. Scrub the other general-purpose registers (i.e., write `zero` to `x[i]` where `i != 2` and `i != switch_reg`).
4. Write the exit code to `x[exit_reg]`.
5. Set the `cwrld` register to `0` (normal world).

▼ **Note: comparison between synchronous and asynchronous exit**

Compare this with CAPEXIT. We require that CAPEXIT be provided with a valid sealed-return capability rather than use the latent capability in `switch_cap`. This allows us to enforce containment of domains in the secure world, so that a domain is prevented from escaping from the secure world when such a behaviour is undesired.

# 9. Memory Consistency Model

TODO

# Appendix A: Debugging Instructions (Non-Normative)

## A.1. World Switching

The instructions SETWORLD and ONPARTITION are related to world switching in TransCapstone.

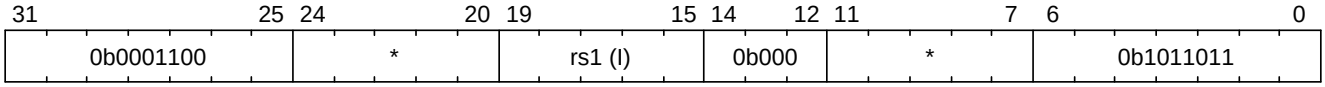


*Figure 48. SETWORLD instruction format*

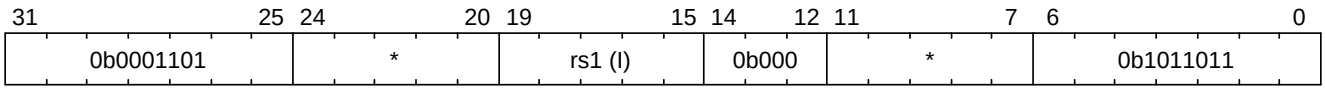


*Figure 49. ONPARTITION instruction format*

The instructions load their operands from the register `x[rs1]`, which expects an integer. SETWORLD directly sets the core to the specified world (`0` for normal world and non-zero for secure world). The program counter will also be made into a capability or an integer correspondingly while retaining the `cursor` value. ONPARTITION switches on (non-zero) or off (`0`) the world partitioning checks in memory.

The instructions make it easy to set up the environment for testing either Pure Capstone or TransCapstone:

- Pure Capstone: secure world, world partitioning checks off
- TransCapstone: normal world, world partitioning checks on

## A.2. Exception Handling

The instructions SETEH and ONNORMALEH affect the behaviours of interrupt and exception handling.

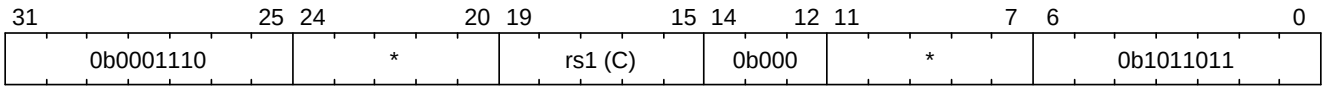


*Figure 50. SETEH instruction format*

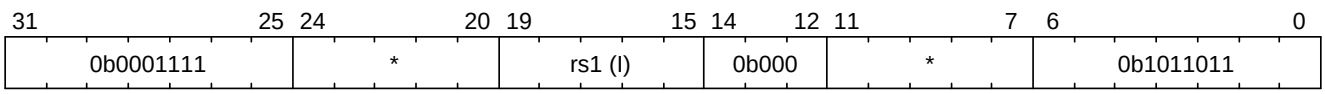


*Figure 51. ONNORMALEH instruction format*

The SETEH instruction sets the secure-world exception handler domain (i.e., `ceh`) to the specified capability `x[rs1]`. The ONNORMALEH instruction checks `x[rs1]` and switches on (non-zero) or off ( `0`) normal world handling of secure-world exceptions. When this is on, an exception that occurs in the secure world will trap to the normal world first before being handled by the secure-world exception handler (`ceh`), which is the expected behaviour in TransCapstone. When it is off, the

exception will be directly handled by the secure-world exception handler, as is expected in Pure Capstone.

# Appendix B: Instruction Listing

## B.1. Debugging Instructions

*Table 11. Debugging instructions*

| Mnemonic | Format | Func3 | Func7 | rs1 | rs2 | rd | imm[11:0] | World | Variant |
|---|---|---|---|---|---|---|---|---|---|
| QUERY | R | 000 | 0000000 | I | - | - | - | * | * |
| RCUPDATE | R | 000 | 0000001 | I | - | I | - | * | * |
| ALLOC | R | 000 | 0000010 | I | - | I | - | * | * |
| REV | R | 000 | 0000011 | I | - | - | - | * | * |
| CAPCREATE | R | 000 | 0000100 | - | - | C | - | * | * |
| CAPTYPE | R | 000 | 0000101 | I | - | C | - | * | * |
| CAPNODE | R | 000 | 0000110 | I | - | C | - | * | * |
| CAPPERM | R | 000 | 0000111 | I | - | C | - | * | * |
| CAPBOUND | R | 000 | 0001000 | I | I | C | - | * | * |
| CAPPRINT | R | 000 | 0001001 | I | - | - | - | * | * |
| TAGSET | R | 000 | 0001010 | I | I | - | - | * | * |
| TAGGET | R | 000 | 0001011 | I | - | I | - | * | * |
| SETWORLD | R | 000 | 0001100 | I | - | - | - | * | T |
| ONPARTITION | R | 000 | 0001101 | I | - | - | - | * | T |
| SETEH | R | 000 | 0001110 | C | - | - | - | * | T |
| ONNORMALEH | R | 000 | 0001111 | I | - | - | - | * | T |

## B.2. Capstone Instructions

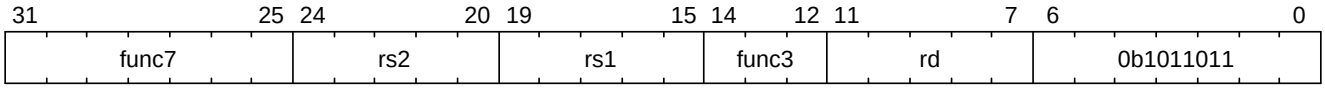


*Figure 52. Instruction format: R-type*

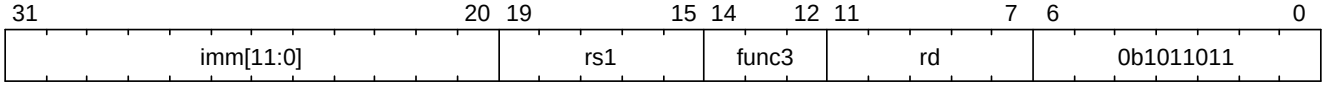


*Figure 53. Instruction format: I-type*

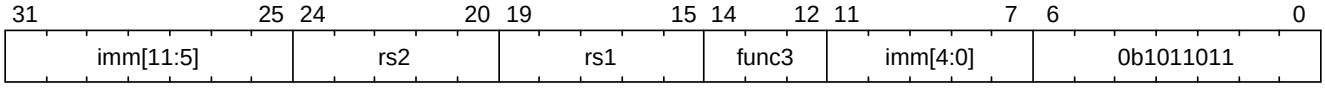


*Figure 54. Instruction format: S-type*

*Table 12. Capability manipulation instructions*

| Mnemonic | Format | Func3 | Func7 | rs1 | rs2 | rd | imm[11:0] | World | Variant |
|---|---|---|---|---|---|---|---|---|---|
| REVOKE | R | 001 | 0000000 | C | - | - | - | * | * |
| SHRINK | R | 001 | 0000001 | I | I | C | - | * | * |
| TIGHTEN | R | 001 | 0000010 | C | Z | C | - | * | * |
| DELIN | R | 001 | 0000011 | - | - | C | - | * | * |
| LCC | R | 001 | 0000100 | C | Z | I | - | * | * |
| SCC | R | 001 | 0000101 | 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 13. Memory access instructions*

| Mnemonic | Format | emode | Func3 | Func7 | rs1 | rs2 | rd | imm[11:0] | World | Variant |
|---|---|---|---|---|---|---|---|---|---|---|
| LDC | I | 0 | 011 | - | I | - | C | S | N | T |
| | I | 1 | 011 | - | C | - | C | S | N | T |
| | I | - | 011 | - | C | - | C | S | S | T |
| | I | - | 011 | - | C | - | C | S | - | P |
| STC | S | 0 | 100 | - | I | C | - | S | N | T |
| | I | 1 | 100 | - | C | C | - | S | N | T |
| | I | - | 100 | - | C | C | - | S | S | T |
| | I | - | 100 | - | C | C | - | S | - | P |

*Table 14. Control flow instructions*

| Mnemonic | Format | Func3 | Func7 | rs1 | rs2 | rd | imm[11:0] | World | Variant |
|---|---|---|---|---|---|---|---|---|---|
| CALL | R | 001 | 0100000 | C | - | Z | - | S | T |
| | R | 001 | 0100000 | C | - | Z | - | - | P |
| RETURN | R | 001 | 0100001 | C | I | - | - | S | T |
| | R | 001 | 0100001 | C | I | - | - | - | P |
| CJALR | I | 101 | - | C | - | C | S | S | T |
| | I | 101 | - | C | - | C | S | - | P |

| Mnemonic | Format | Func3 | Func7 | rs1 | rs2 | rd | imm[11:0] | World | Variant |
|---|---|---|---|---|---|---|---|---|---|
| CBNZ | I | 110 | - | I | - | C | S | S | T |
| | I | 110 | - | I | - | C | S | - | P |
| CAPENTER | R | 001 | 0100010 | C | - | Z | - | N | T |
| CAPEXIT | R | 001 | 0100011 | C | I | - | - | S | T |

*Table 15. Control and status instructions*

| Mnemonic | Format | Func3 | Func7 | rs1 | rs2 | rd | imm[11:0] | World | Variant |
|---|---|---|---|---|---|---|---|---|---|
| CCSRRW | I | 111 | - | C | - | C | Z | * | * |

# B.3. Extended RV64IZicsr Memory Access Instructions

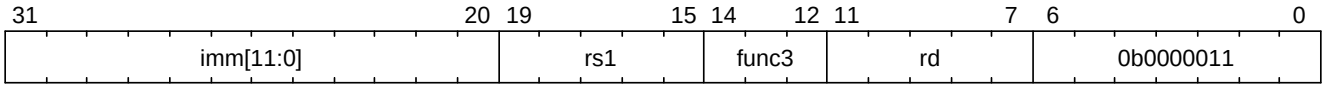


*Figure 55. Instruction format: I-type*

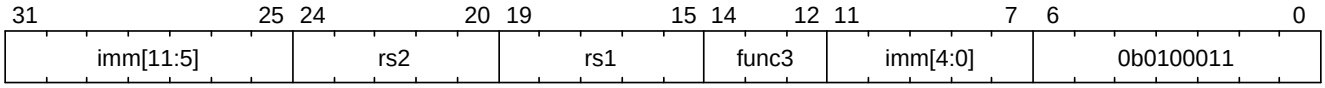


*Figure 56. Instruction format: S-type*

*Table 16. Extended RV64IZicsr load instructions*

| Mnemonic | Format | emode | Func3 | Func7 | rs1 | rs2 | rd | imm[11:0] | World | Variant |
|---|---|---|---|---|---|---|---|---|---|---|
| lb | I | 0 | 000 | - | I | - | I | S | N | T |
| | I | 1 | 000 | - | C | - | I | S | N | T |
| | I | - | 000 | - | C | - | I | S | S | T |
| | I | - | 000 | - | C | - | I | S | - | P |
| lh | I | 0 | 001 | - | I | - | I | S | N | T |
| | I | 1 | 001 | - | C | - | I | S | N | T |
| | I | - | 001 | - | C | - | I | S | S | T |
| | I | - | 001 | - | C | - | I | S | - | P |
| lw | I | 0 | 010 | - | I | - | I | S | N | T |
| | I | 1 | 010 | - | C | - | I | S | N | T |
| | I | - | 010 | - | C | - | I | S | S | T |
| | I | - | 010 | - | C | - | I | S | - | P |
| ld | I | 0 | 011 | - | I | - | I | S | N | T |
| | I | 1 | 011 | - | C | - | I | S | N | T |
| | I | - | 011 | - | C | - | I | S | S | T |
| | I | - | 011 | - | C | - | I | S | - | P |

| Mnemonic | Format | emode | Func3 | Func7 | rs1 | rs2 | rd | imm[11:0] | World | Variant |
|---|---|---|---|---|---|---|---|---|---|---|
| lbu | I | 0 | 100 | - | I | - | I | S | N | T |
| | I | 1 | 100 | - | C | - | I | S | N | T |
| | I | - | 100 | - | C | - | I | S | S | T |
| | I | - | 100 | - | C | - | I | S | - | P |
| lhu | I | 0 | 101 | - | I | - | I | S | N | T |
| | I | 1 | 101 | - | C | - | I | S | N | T |
| | I | - | 101 | - | C | - | I | S | S | T |
| | I | - | 101 | - | C | - | I | S | - | P |
| lwu | I | 0 | 110 | - | I | - | I | S | N | T |
| | I | 1 | 110 | - | C | - | I | S | N | T |
| | I | - | 110 | - | C | - | I | S | S | T |
| | I | - | 110 | - | C | - | I | S | - | P |

*Table 17. Extended RV64IZicsr store instructions*

| Mnemonic | Format | emode | Func3 | Func7 | rs1 | rs2 | rd | imm[11:0] | World | Variant |
|---|---|---|---|---|---|---|---|---|---|---|
| sb | S | 0 | 000 | - | I | I | - | S | N | T |
| | S | 1 | 000 | - | C | I | - | S | N | T |
| | S | - | 000 | - | C | I | - | S | S | T |
| | S | - | 000 | - | C | I | - | S | - | P |
| sh | S | 0 | 001 | - | I | I | - | S | N | T |
| | S | 1 | 001 | - | C | I | - | S | N | T |
| | S | - | 001 | - | C | I | - | S | S | T |
| | S | - | 001 | - | C | I | - | S | - | P |
| sw | S | 0 | 010 | - | I | I | - | S | N | T |
| | S | 1 | 010 | - | C | I | - | S | N | T |
| | S | - | 010 | - | C | I | - | S | S | T |
| | S | - | 010 | - | C | I | - | S | - | P |
| sd | S | 0 | 011 | - | I | I | - | S | N | T |
| | S | 1 | 011 | - | C | I | - | S | N | T |
| | S | - | 011 | - | C | I | - | S | S | T |
| | S | - | 011 | - | C | I | - | S | - | P |

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

**For instruction operands:**

**I**

Integer register

**C**

Capability register

**S**

Used as sign-extended immediate

**Z**

Used as zero-extended immediate

**-**

Not used

**For immediates:**

**S**

Sign-extended

**Z**

Zero-extended

**-**

Not used

**For worlds:**

**N**

Normal world

**S**

Secure world

*****

Either world

**For variants:**

**P**

*Pure Capstone*

**T**

*TransCapstone*

*****

Either variant

# Appendix C: Assembly Code Examples

TODO

# Appendix D: Abstract Binary Interface (Non-Normative)

TODO