The Capstone-RISC-V Academic Version Instruction Set Reference

Table of Contents

Version Information: Version 1.0

1. Introduction

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

1.1. Properties to Support

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

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

1.2. Major Design Elements

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

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

1.3. Capstone-RISC-V Academic Version ISA Overview

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

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

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

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

1.4. Assembly Mnemonics

Each Capstone-RISC-V Academic Version instruction is given a mnemonic prefixed with `CS.`. In contexts where it is clear we are discussing Capstone-RISC-V Academic Version 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.

1.5. 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.

1.6. Bibliography

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

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

2. Programming Model

The Capstone-RISC-V Academic Version 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 1. Fields in a capability

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

Name	Range	Description
`perms`	`0..7`	Not applicable when `type = 4` (sealed) or `type = 5` (sealed-return). One-hot encoded permissions associated with the capability: `0` = no access, `1` = execute-only, `2` = write-only, `3` = write-execute, `4` = read-only, `5` = read-execute, `6` = read-write, `7` = read-write-execute
`async`	`0..2`	Only applicable when `type = 4` (sealed) or `type = 5` (sealed-return). How the capability is sealed: `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 implementation of `valid` field**

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

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

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

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

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

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

Table 2. Fields used for each type of capabilities

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

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

Table 3. Memory accesses granted by sealed-return

Capability type	async	Read	Write	Execute
Sealed-return	`0`	`cursor in [base + 3 * CLENBYTES, base + 33 * CLENBYTES - size]`	`cursor in [base + 3 * CLENBYTES, base + 33 * CLENBYTES - size]`	No

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

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

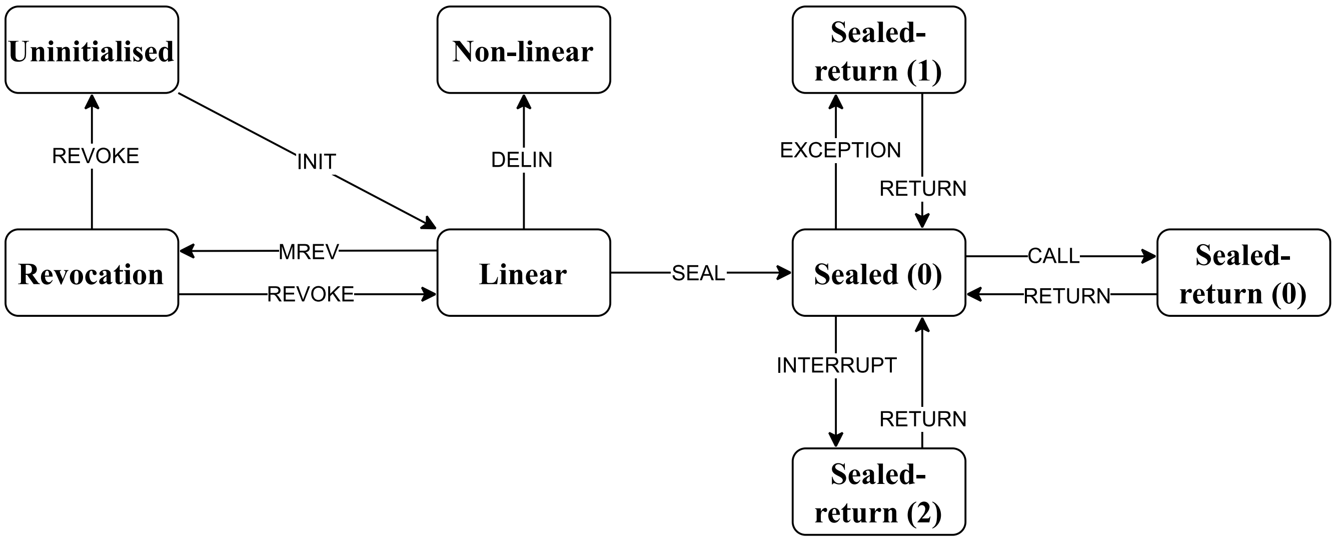

Figure 1. Overview of different types of capabilities in Capstone-RISC-V Academic Version

2.2. Extension to General-Purpose Registers

The Capstone-RISC-V Academic Version 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`

Index	`XLEN`-bit integer	Capability
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`
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 program counter (`pc`) is changed to contain a capability only.

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

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

If no exception is raised:

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

2.4. Added Registers

The Capstone-RISC-V Academic Version ISA adds the following registers.

Table 4. Additional Registers in Capstone-RISC-V Academic Version ISA

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
`cinit`	`0x002`	-	The initial capability covering the entire address space of the memory
`epc`	`0x003`	-	The exception program counter register
`cis`	-	`0x800`	The interrupt status register
`tval`	-	`0x801`	The exception data (trap value) register
`cause`	-	`0x802`	The exception cause register

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

The manipulation constraints for each CCSR are indicated below.

Table 5. Manipulation Constraints for CCSRs

Mnemonic	Read	Write
`ceh`	No constraint	No constraint
`cih`	-	The original content must not be a capability
`cinit`	Oone-time only	-
`epc`	No constraint	No constraint

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

The manipulation constraints for each additional CSR are indicated below.

Table 6. Manipulation Constraints for Additional CSRs

Mnemonic	Read	Write
`cis`	No constraint	No constraint
`tval`	No constraint	No constraint
`cause`	No constraint	No constraint

▼ **Note: `ceh` and `cih`**

`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`**

`cinit` is a CCSR that is used to bootstrap capabilities after a system reset. control and status instructions can be used to read the initial capability in `cinit` and write it to 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`.

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

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

2.6. Instruction Set

The Capstone-RISC-V Academic Version 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 Academic Version instructions occupies the “custom-2” subset, i.e., the opcode of all Capstone-RISC-V Academic Version instructions is `0b1011011`.

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

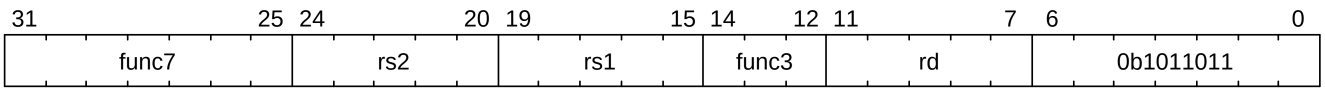

Figure 2. R-type instruction format

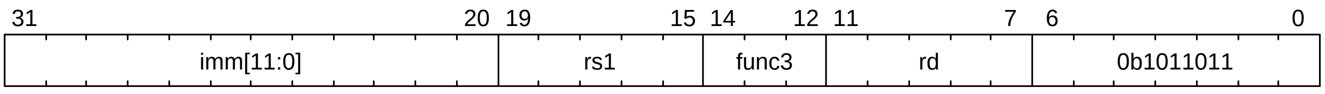

Figure 3. I-type instruction format

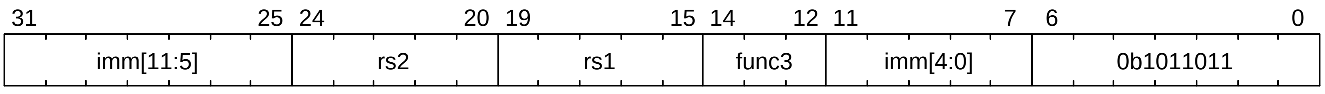

Figure 4. S-type instruction format

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

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

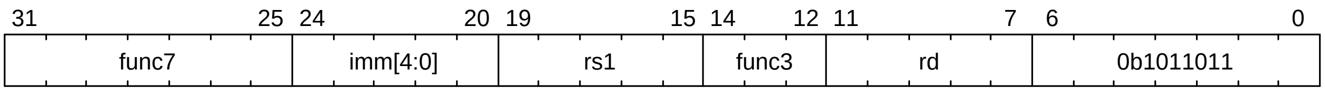

Figure 5. RI-type instruction format

The so-called RI-type instructions are actually *derivatives* of R-type instructions. They receive up to two register operands and a 5-bit-wide immediate operand.

2.7. System Reset

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

- Each general-purpose register either contains an integer, or a capability with `valid = 0` (invalid).
- No addressable memory location can contain a capability.
- `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)`.

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.

31 25	24 20	19 15	14 12	11 7	6 0
0b0001010	*	rs1 (C)	0b001	rd (C)	0b1011011

Figure 6. MOVC instruction format

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

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

If no exception is raised:

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

3.1.2. Cursor Increment

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

CINCOFFSET

31 25	24 20	19 15	14 12	11 7	6 0
0b0001100	rs2 (I)	rs1 (C)	0b001	rd (C)	0b1011011

Figure 7. CINCOFFSET instruction format

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

- `Unexpected operand type (24)`

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

If no exception is raised:

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

CINCOFFSETIMM

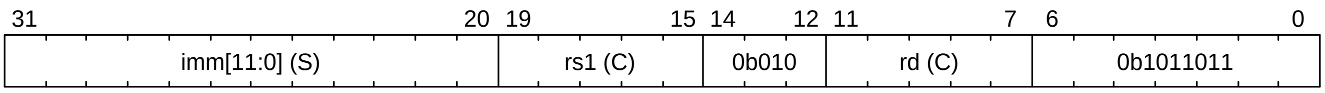

Figure 8. CINCOFFSETIMM instruction format

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

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

If no exception is raised:

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

3.1.3. Cursor Setter

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

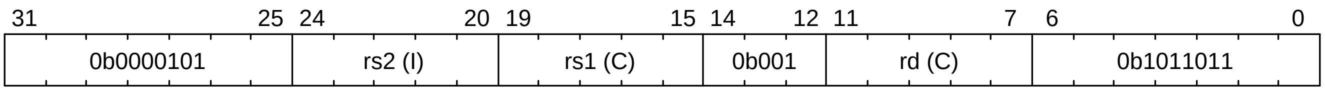

Figure 9. SCC instruction format

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

- `Unexpected operand type (24)`

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

If no exception is raised:

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

3.1.4. Field Query

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

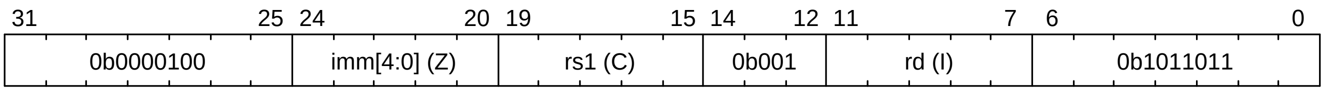

Figure 10. LCC instruction format

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

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

If no exception is raised:

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

Table 7. LCC multiplexing table

`imm`	`field`
`0`	`x[rs1].valid`

imm	field
1	x[rs1].type
2	x[rs1].cursor
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.

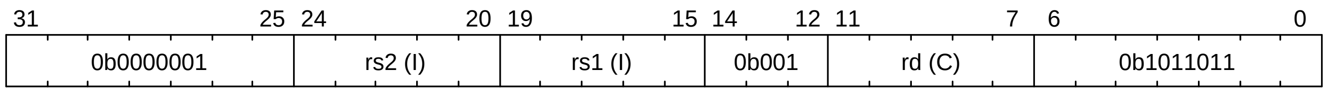

Figure 11. SHRINK instruction format

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

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

If no exception is raised:

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

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).

31 25	24 20	19 15	14 12	11 7	6 0
0b0000110	rs2 (I)	rs1 (C)	0b001	rd (C)	0b1011011

Figure 12. SPLIT instruction format

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

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

If no exception is raised:

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

3.1.7. Permission Tightening

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

31 25	24 20	19 15	14 12	11 7	6 0
0b0000010	imm[4:0] (Z)	rs1 (C)	0b001	rd (C)	0b1011011

Figure 13. TIGHTEN instruction format

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

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0`, `1`, or `3` (linear, non-linear, or uninitialised).
- `Illegal operand value (29)`

    ◦ `imm <= 7`, and `imm <=p x[rs1].perms` does not hold.

If no exception is raised:

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

3.2. Type Manipulation

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

3.2.1. Delinearisation

The DELIN instruction delinearises a linear capability.

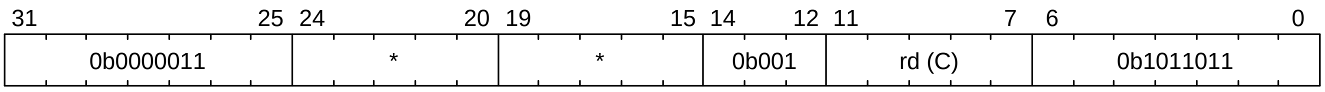

Figure 14. DELIN instruction format

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

- `Unexpected operand type (24)`
    ◦ `x[rd]` is not a capability.
- `Unexpected capability type (26)`
    ◦ `x[rd].type` is not `0` (linear).

If no exception is raised:

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

3.2.2. Initialisation

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

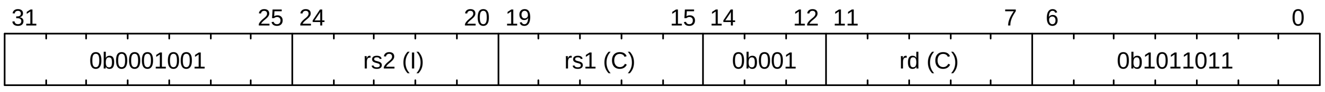

Figure 15. INIT instruction format

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

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

If no exception is raised:

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

3.2.3. Sealing

The SEAL instruction seals a linear capability.

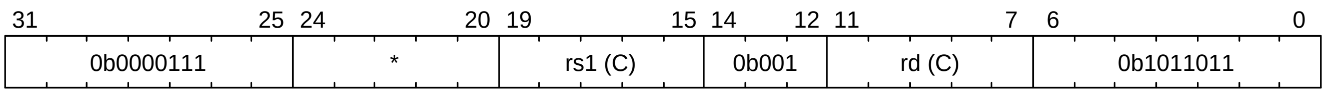

Figure 16. SEAL instruction format

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

- Unexpected operand type (24)
    - x[rs1] is not a capability.
- Unexpected capability type (26)
    - x[rs1].type is not 0 (linear).
- Insufficient capability permissions (27)
    - 6 <=p x[rs1].perms does not hold.
- Illegal operand value (29)
    - The size of the memory region associated with x[rs1] is smaller than CLENBYTES * 33 bytes (i.e., x[rs1].end - x[rs1].base < CLENBYTES * 33).
    - x[rs1].base is not aligned to CLENBYTES bytes.

If no exception is raised:

1. MOVC rd, rs1.

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

3.3. Dropping

The DROP instruction invalidates a capability.

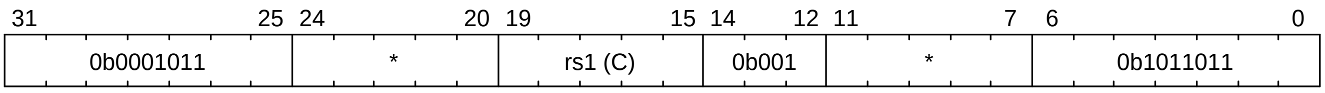

Figure 17. DROP instruction format

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

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

If no exception is raised:

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

3.4. Revocation

3.4.1. Revocation Capability Creation

The MREV instruction creates a revocation capability.

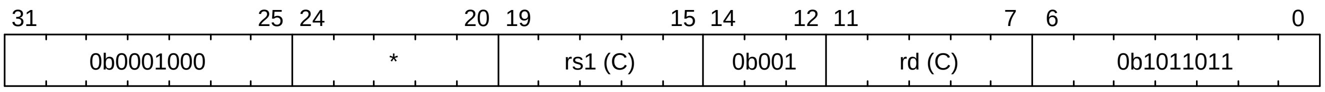

Figure 18. MREV instruction format

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

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

If no exception is raised:

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

3.4.2. Revocation Operation

The REVOKE instruction revokes a capability.

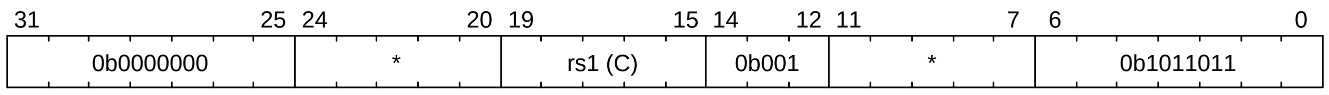

Figure 19. REVOKE instruction format

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

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

If no exception is raised:

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

4. Memory Access Instructions

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

4.1. Load Capabilities

The LDC instruction loads a capability from the memory.

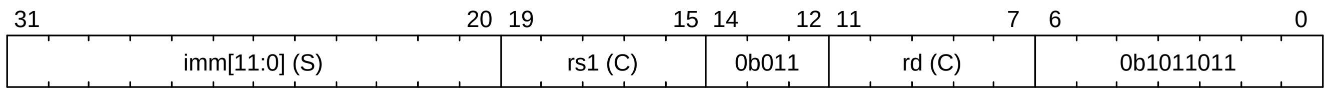

Figure 20. LDC instruction format

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

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
- `Invalid capability (25)`
  - `x[rs1].valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0` (linear), `1` (non-linear) or `5` (sealed-return).
  - `x[rs1].type` is `5` (sealed-return) and `x[rs1].async` is not `0` (synchronous).
- `Insufficient capability permissions (27)`
  - `x[rs1].type` is `0` (linear) or `1` (non-linear) and `4 <=p x[rs1].perms` does not hold.
- `Capability out of bound (28)`
  - `x[rs1].type` is `0` (linear) or `1` (non-linear), and `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - CLENBYTES]`.
  - `x[rs1].type` is `5` (sealed-return), and `x[rs1].cursor + imm` is not in the range `[x[rs1].base + 3 * CLENBYTES, x[rs1].base + 33 * CLENBYTES - CLENBYTES]`.
- `Load address misaligned (4)`
  - `x[rs1].cursor + imm` is not aligned to `CLENBYTES` bytes.
- `Load access fault (5)`
  - 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:

1. Set `cap` to `x[rs1]`.
2. Load the capability at the memory location `cap.cursor + imm, cap.cursor + imm + CLENBYTES)` into `x[rd]`.
3. If `x[rd].type` is not `1` (non-linear), write `cnull` to the memory location `[cap.cursor + imm, cap.cursor + imm + CLENBYTES)`.

4.2. Store Capabilities

The STC instruction stores a capability to the memory.

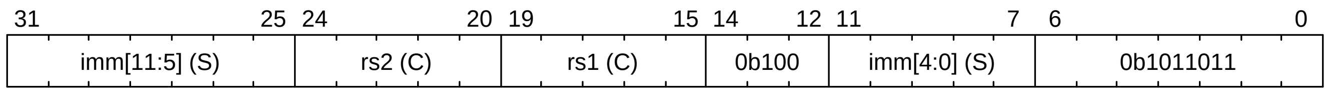

Figure 21. STC instruction format

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

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
  - `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` (linear), `1` (non-linear), `3` (uninitialised) or `5` (sealed-return).
  - `x[rs1].type` is `5` (sealed-return) and `x[rs1].async` is not `0` (synchronous).
- `Insufficient capability permissions (27)`
  - `x[rs1].type` is `0` or `1`, and `2 <=p x[rs1].perms` does not hold.
- `Illegal operand value (29)`
  - `x[rs1].type` is `3` (uninitialised) and `imm` is not `0`.
- `Capability out of bound (28)`
  - `x[rs1].type` is `0`, `1`, or `3`, and `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - CLENBYTES]`.
  - `x[rs1].type` is `5` or `6`, and `x[rs1].cursor + imm` is not in the range `[x[rs1].base + 3 * CLENBYTES, x[rs1].base + 33 * CLENBYTES - CLENBYTES]`.
- `Store/AMO address misaligned (6)`
  - `x[rs1].cursor + imm` is not aligned to `CLENBYTES` bytes.

If no exception is raised:

1. Store `x[rs2]` to the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + CLENBYTES)`.
2. If `x[rs1].type` is `3` (uninitialised), set `x[rs1].cursor` to `x[rs1].cursor + CLENBYTES`.
3. If `x[rs2].type` is not `1` (non-linear), write `cnull` to `x[rs2]`.

5. Control Flow Instructions

5.1. Jump to Capabilities

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

5.1.1. CJALR

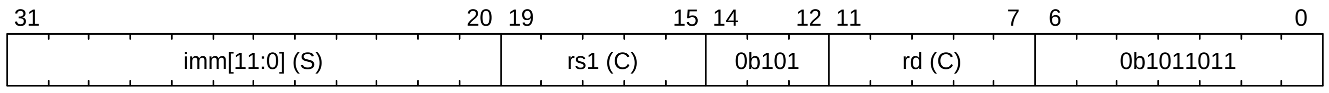

Figure 22. CJALR instruction format

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

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

If no exception is raised:

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

5.1.2. CBNZ

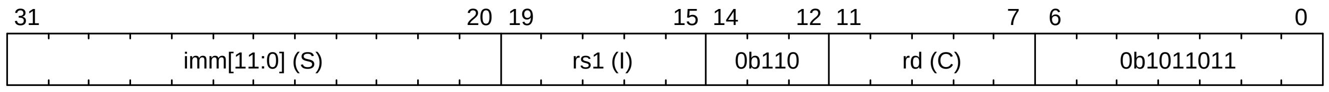

Figure 23. CBNZ instruction format

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

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

If no exception is raised:

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

5.2. Domain Crossing

Domains in Capstone-RISC-V Academic Version 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*.

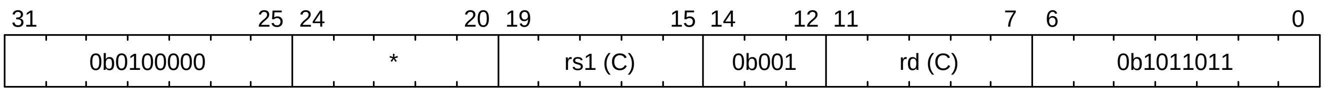

Figure 24. CALL instruction format

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

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

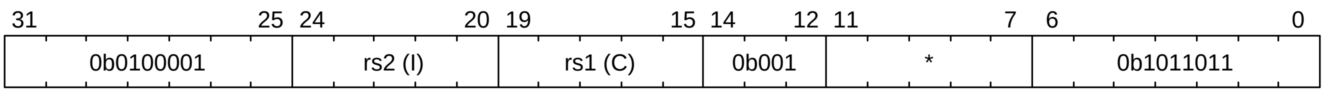

Figure 25. RETURN instruction format

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

- 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 pc to ceh, and epc to pc.
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)`.

6. Control and Status Instructions

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

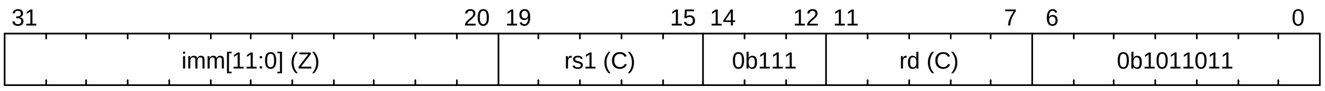

Figure 26. CCSRRW instruction format

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

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

If no exception is raised:

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

7. Adjustments to Existing Instructions

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

The following instructions in RV64IZicsr are adjusted in Capstone:

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

7.1. Memory Access Instructions

In RV64IZicsr, memory access instructions include load instructions (i.e., `lb`, `lh`, `ld`, `lw`, `lbu`, `lhu`, `lwu`), and store instructions (i.e., `sb`, `sh`, `sw`, `sd`). These instructions take an integer as a raw address, and load or store a value from/to this address. In Capstone, these instructions are extended to take a capability as an address.

7.1.1. Load Instructions

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

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

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

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

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

- `Unexpected operand type (24)`

    - `x[rs1]` is not a capability.
- `Invalid capability (25)`
    - `x[rs1].valid` is `0` (invalid).
- `Unexpected capability type (26)`
    - `x[rs1].type` is not `0` (linear), `1` (non-linear) or `5` (sealed-return).
    - `x[rs1].type` is `5` (sealed-return) and `x[rs1].async` is not `0` (synchronous).
- `Insufficient capability permissions (27)`
    - `x[rs1].type` is `0` (linear) or 1 (non-linear) and `4 <=p x[rs1].perms` does not hold.
- `Capability out of bound (28)`
    - `x[rs1].type` is `0` (linear) or `1` (non-linear), and `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - size]`.
    - `x[rs1].type` is `5` (sealed-return), and `x[rs1].cursor + imm` is not in the range `[x[rs1].base + 3 * CLENBYTES, x[rs1].base + 33 * CLENBYTES - size]`.
- `Load address misaligned (4)`
    - `x[rs1].cursor + imm` is not aligned to `size` bytes.

31 20	19 15	14 12	11 7	6 0
imm[11:0] (S)	rs1 (C)	0b000	rd (I)	0000011

Figure 27. lb instruction format

31 20	19 15	14 12	11 7	6 0
imm[11:0] (S)	rs1 (C)	0b001	rd (I)	0000011

Figure 28. lh instruction format

31 20	19 15	14 12	11 7	6 0
imm[11:0] (S)	rs1 (C)	0b010	rd (I)	0000011

Figure 29. lw instruction format

31 20	19 15	14 12	11 7	6 0
imm[11:0] (S)	rs1 (C)	0b011	rd (I)	0000011

Figure 30. ld instruction format

If no exception is raised:

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

31 20	19 15	14 12	11 7	6 0
imm[11:0] (S)	rs1 (C)	0b100	rd (I)	0000011

Figure 31. lbu instruction format

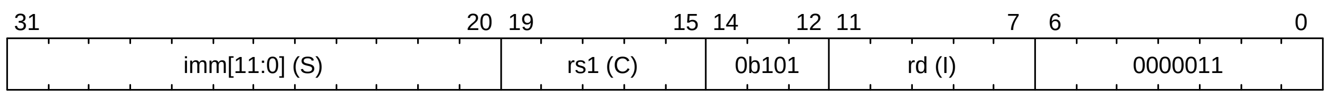

Figure 32. lhu instruction format

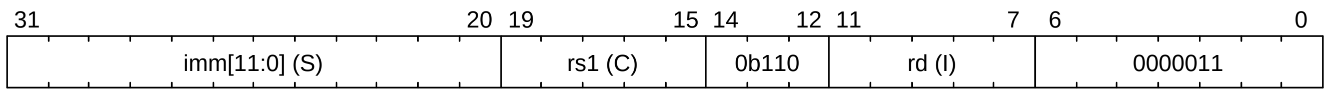

Figure 33. lwu instruction format

If no exception is raised:

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

7.1.2. Store Instructions

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

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

Mnemonic	`size`
`sb`	`1`
`sh`	`2`
`sw`	`4`
`sd`	`8`

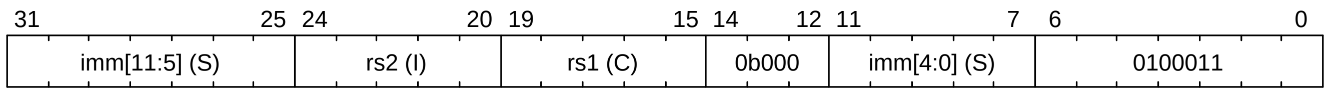

Figure 34. sb instruction format

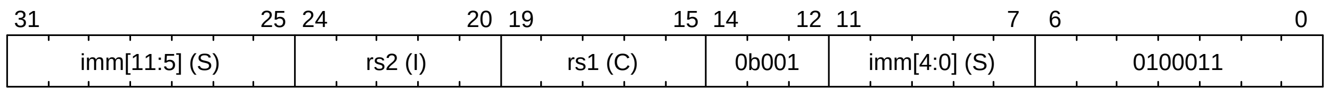

Figure 35. sh instruction format

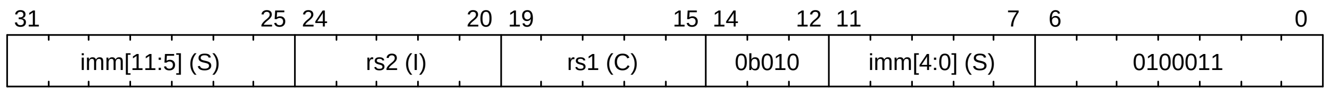

Figure 36. sw instruction format

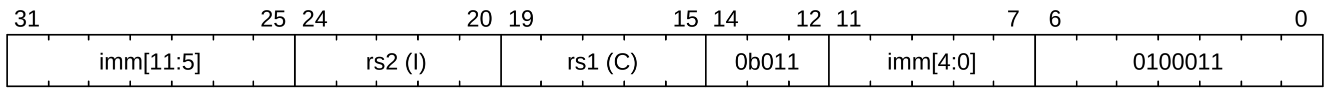

Figure 37. sd instruction format

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

- `Unexpected operand type (24)`
  - `x[rs1]` is not a capability.
  - `x[rs2]` is not an integer.
- `Invalid capability (25)`
  - `x[rs1].valid` is `0` (invalid).
- `Unexpected capability type (26)`
  - `x[rs1].type` is not `0` (linear), `1` (non-linear), `3` (uninitialised) or `5` (sealed-return).
  - `x[rs1].type` is `5` (sealed-return) and `x[rs1].async` is not `0` (synchronous).
- `Insufficient capability permissions (27)`
  - `x[rs1].type` is `0` or `1`, and `2 <=p x[rs1].perms` does not hold.
- `Illegal operand value (29)`
  - `x[rs1].type` is `3` (uninitialised) and `imm` is not `0`.
- `Capability out of bound (28)`
  - `x[rs1].type` is `0`, `1`, or `3`, and `x[rs1].cursor + imm` is not in the range `[x[rs1].base, x[rs1].end - size]`.
  - `x[rs1].type` is `5` or `6`, and `x[rs1].cursor + imm` is not in the range `[x[rs1].base + 3 * CLENBYTES, x[rs1].base + 33 * CLENBYTES - size]`.
- `Store/AMO address misaligned (6)`
  - `x[rs1].cursor + imm` is not aligned to `size` bytes.

If no exception is raised:

1. Store `x[rs2]` to the memory location `[x[rs1].cursor + imm, x[rs1].cursor + imm + size)` as an integer.
2. The content in the `CLENBYTES`-byte aligned memory location `[cbase, cend)`, which aliases with the 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`.
3. If `x[rs1].type` is `3` (uninitialised), set `x[rs1].cursor` to `x[rs1].cursor + size`.

▼ **Note: undefined behaviour**

The following load results are *undefined*:

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

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

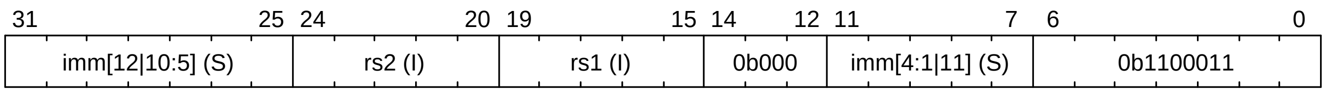

Figure 38. beq instruction format

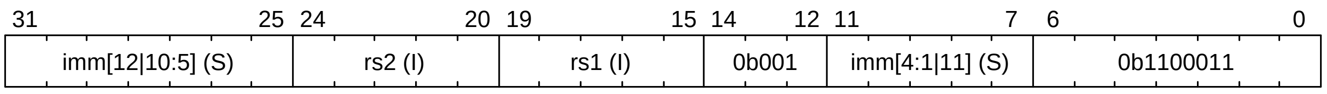

Figure 39. bne instruction format

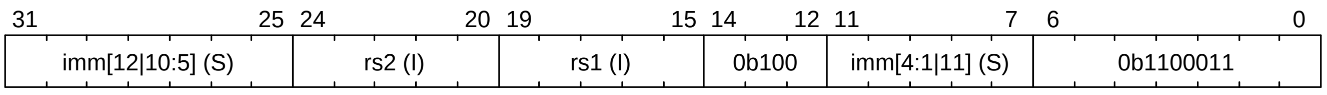

Figure 40. blt instruction format

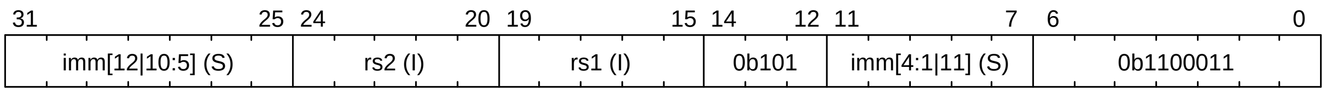

Figure 41. bge instruction format

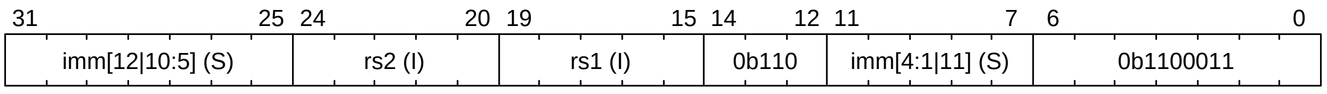

Figure 42. bltu instruction format

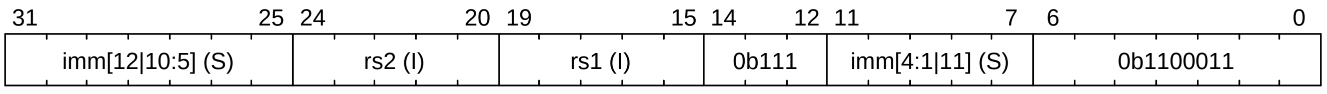

Figure 43. bgeu instruction format

The following adjustments are made to these instructions:

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

7.2.2. Jump Instructions

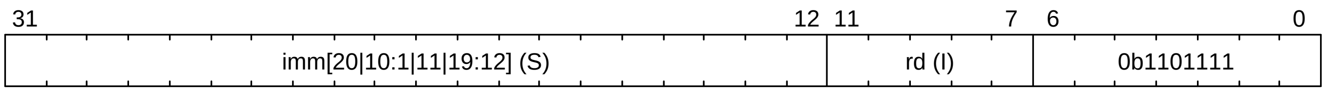

Figure 44. jal instruction format

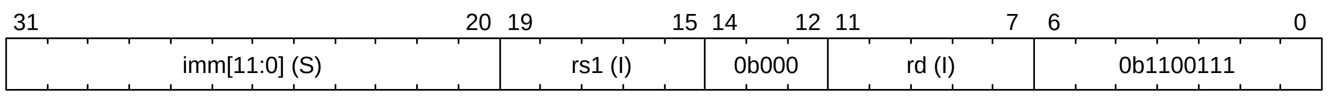

Figure 45. jalr instruction format

The following adjustments are made to these instructions:

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

7.3. Illegal Instructions

Some instructions in RV64IZicsr now raise `illegal instruction (2)` exceptions when executed in Capstone-RISC-V Academic Version ISA, under all or some circumstances.

These instructions are:

- 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 Academic Version, 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 8. 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
Insufficient system resources	30	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.

▼ **Note: Implementation specified exception**

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

- `Insufficient system resources (30)`

8.2. Exception Data

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.

▼ **Note: `tval` is only available in in-domain exception handling**

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 9. 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)
`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.

The figure below shows the overview of domain switch in *Pure Capstone*, including synchronous domain crossing and asynchronous interrupt/exception handling.

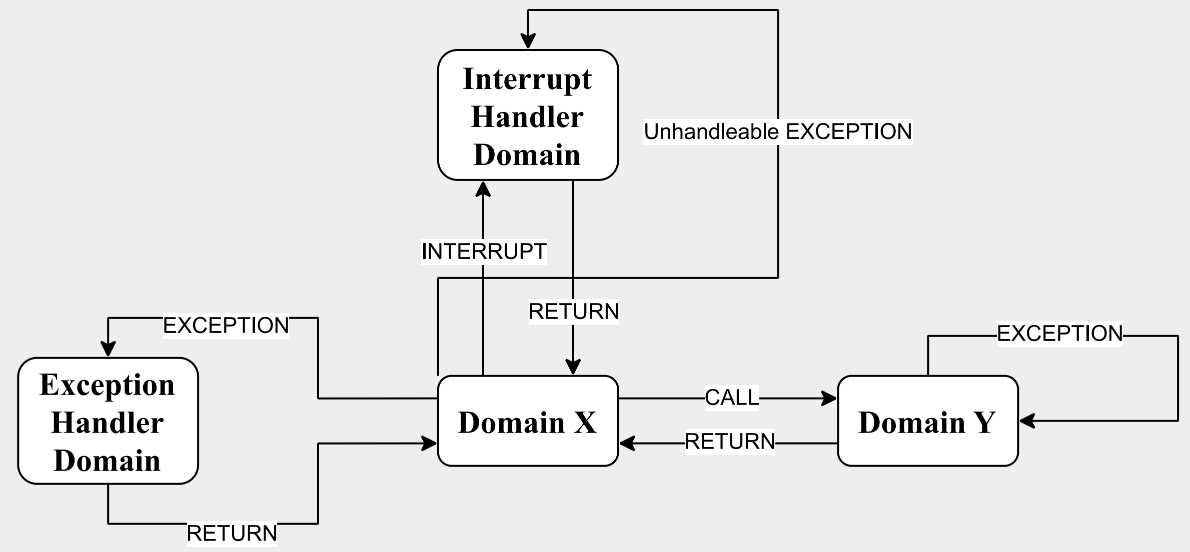

Figure 46. Overview of domain switch in Pure Capstone

8.3.1. Interrupt Status

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

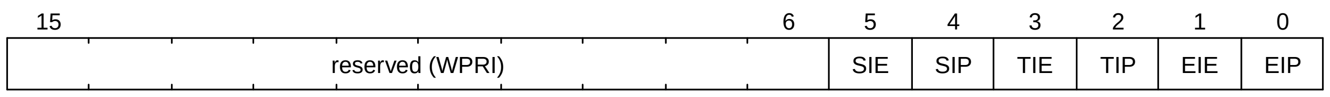

Figure 47. `cis` CSR layout

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).
- `cih.async != 0` (synchronous).

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`).
- The capability in `ceh` is a sealed capability (`type = 4`) and the `ceh.async` field is not `0` (synchronous).

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.

The figure below shows the overview of interrupt handling in *TransCapstone*.

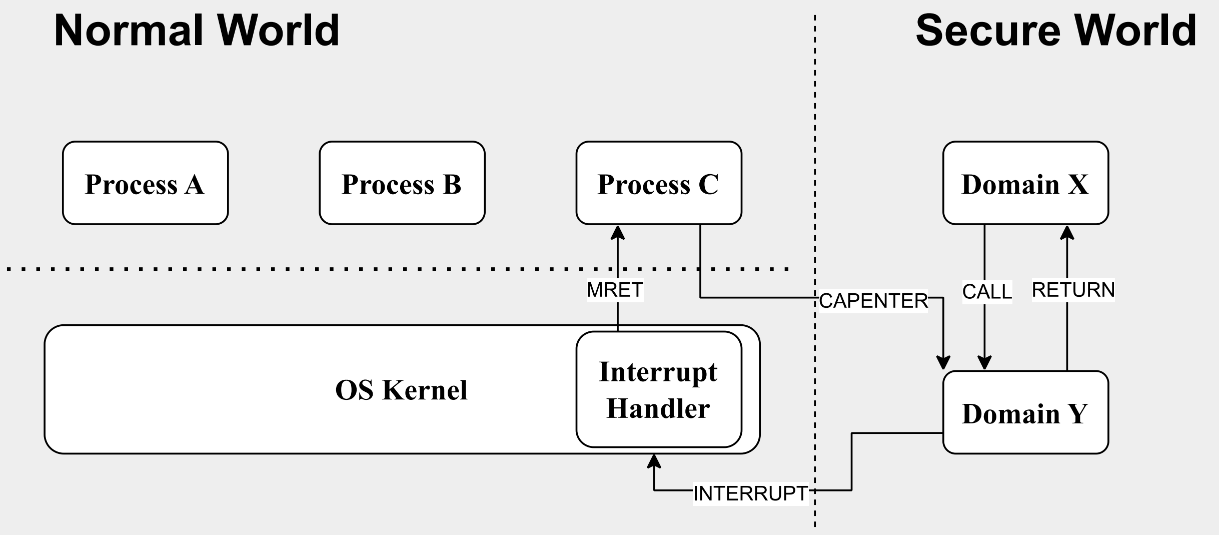

Figure 48. Overview of interrupt handling in TransCapstone

▼ **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).

The figure below shows the overview of exception handling in *TransCapstone*.

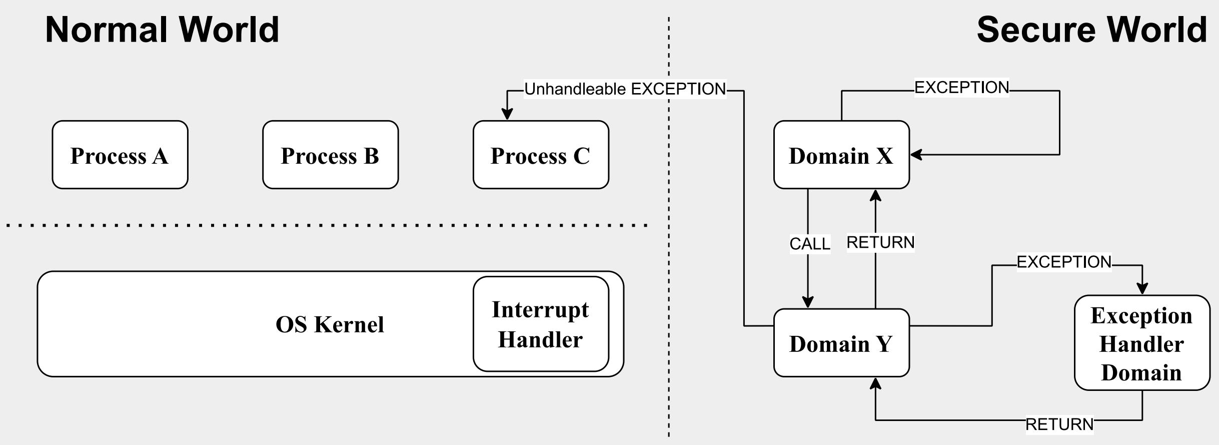

Figure 49. Overview of exception handling in TransCapstone

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` satisfies the following conditions:

- `switch_cap` is a capability.
- `switch_cap.valid` is 1 (valid).
- `switch_cap.type` is 0 (linear) or 3 (uninitialised).
- `switch_cap.base` is aligned to `CLENBYTES`.
- `6 <=p switch_cap.perms` holds.
- `switch_cap.end - switch_cap.base >= CLENBYTES * 33` holds.

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.type` to `4` (sealed), `switch_cap.async` 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` satisfies the following conditions:

- `ceh` is a capability.
- `ceh.type` is `4` (sealed).
- `ceh.valid` is `1` (valid).
- `ceh.async` is `0` (synchronous)

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 satisfies the following conditions:

- switch_cap is a capability.
- switch_cap.valid is 1 (valid).
- switch_cap.type is 0 (linear) or 3 (uninitialised).
- switch_cap.base is aligned to CLENBYTES.
- 6 <=p switch_cap.perms holds.
- switch_cap.end - switch_cap.base >= CLENBYTES * 33 holds.

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. Write normal_pc + 4 and normal_sp to pc and sp respectively.
6. Set switch_cap.type to 4 (sealed), switch_cap.async to 1 (upon exception).
7. Write the content of switch_cap to x[switch_reg], and cnull to switch_cap.
8. Scrub the other general-purpose registers (i.e., write zero to x[i] where i != 2 and i != switch_reg).
9. Write the exit code to x[exit_reg].
10. Set the cwrld register to 0 (normal world).

Otherwise:

1. Write normal_pc + 4 and normal_sp to pc and 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

Appendix A: Instruction Listing

A.1. Capstone Instructions

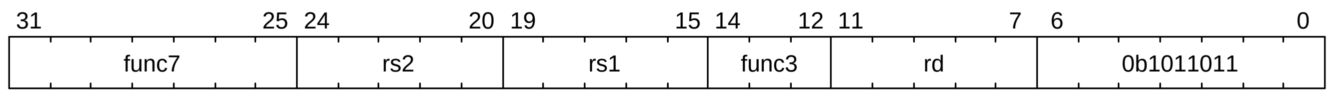

Figure 50. Instruction format: R-type

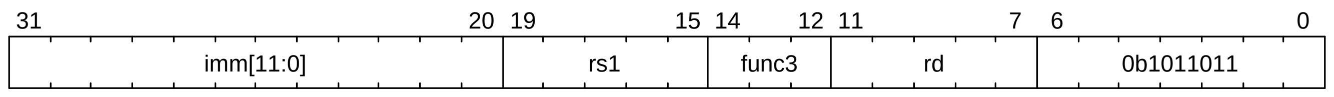

Figure 51. Instruction format: I-type

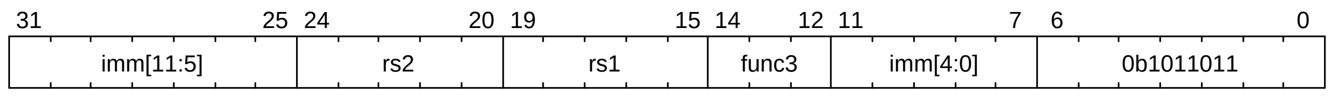

Figure 52. Instruction format: S-type

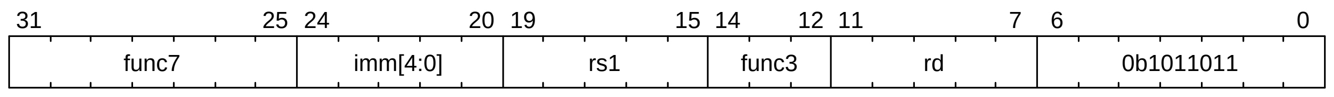

Figure 53. Instruction format: RI-type

Table 10. Capability manipulation instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm [4:0]	imm[11:0]	World	Variant
REVOKE	R	`001`	`0000000`	C	-	-	-	-	*	*
SHRINK	R	`001`	`0000001`	I	I	C	-	-	*	*
TIGHTEN	RI	`001`	`0000010`	C	-	C	Z	-	*	*
DELIN	R	`001`	`0000011`	-	-	C	-	-	*	*
LCC	RI	`001`	`0000100`	C	-	I	Z	-	*	*
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 11. 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
	S	1	100	-	C	C	-	S	N	T
	S	-	100	-	C	C	-	S	S	T
	S	-	100	-	C	C	-	S	-	P

Table 12. Control flow instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]	World	Variant
CALL	R	001	0100000	C	-	C	-	S	T
	R	001	0100000	C	-	C	-	-	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
CBNZ	I	110	-	I	-	C	S	S	T
	I	110	-	I	-	C	S	-	P
CAPENTER	R	001	0100010	C	-	I	-	N	T
CAPEXIT	R	001	0100011	C	I	-	-	S	T

Table 13. Control and status instructions

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

A.2. Extended RV64IZicsr Memory Access Instructions

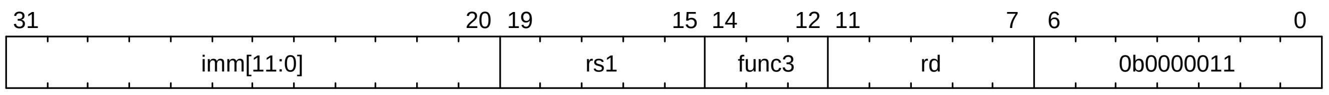

Figure 54. Instruction format: I-type

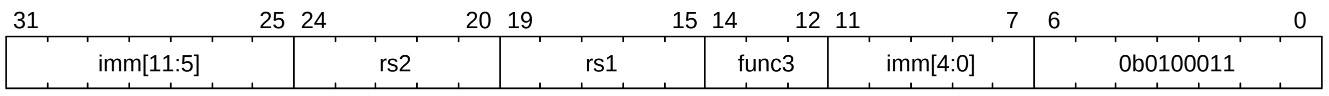

Figure 55. Instruction format: S-type

Table 14. 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
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 15. 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

Mnemonic	Format	emode	Func3	Func7	rs1	rs2	rd	imm[11:0]	World	Variant
	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
-	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 B: Comparison with Other Capability-Based ISA Extensions to RISC-V

Similar to Capstone-RISC-V Academic Version, CHERI-RISC-V [1] and CHERIoT [2] are also capability-based ISA extension to RISC-V, both derived from the CHERI architecture. CHERI-RISC-V is designed for general-purpose computing, whereas CHERIoT builds on RV32E and specialises in low-cost embedded systems such as IoT devices.

We discuss the commonalities and differences between Capstone-RISC-V Academic Version, CHERI-RISC-V, and CHERIoT in this appendix, in the hope to shed light on how to allow Capstone-RISC-V Academic Version to coexist with the other two ISA extensions in the RISC-V ecosystem.

B.1. Commonalities

Capstone-RISC-V Academic Version, CHERI-RISC-V, and CHERIoT all use architectural capabilities to allow capabilities to be stored in either registers or memory, with hardware-enforced provenance and monotonicity guarantees as well as bounds checks on capability dereferences. As a result, some of the instructions in the three ISAs have obvious and direct correspondence, as summarised in the following table.

Table 16. Correspondence between Capstone-RISC-V Academic Version, CHERI-RISC-V, and CHERIoT instructions

Capstone-RISC-V Academic Version instruction(s)	CHERI-RISC-V instruction(s)	CHERIoT instruction(s)
DROP	CClearTag	CClearTag
CJALR	CJALR	CJALR
CALL	CInvoke	-
SEAL	CSealEntry	-
CIncOffset	CIncOffset	CIncAddr
CIncOffsetImm	CIncOffsetImm	CIncAddrImm
LCC	CGetAddr, CGetBase, CGetType, CGetPerm	CGetAddr, CGetBase, CGetTop, CGetType, CGetPerm
SCC	CSetAddr	CSetAddr
TIGHTEN	CAndPerm	CAndPerm
SHRINK	CSetBounds, CSetBoundsExact	CSetBounds, CSetBoundsExact
MOVC	CMove	CMove
LDC	LC.CAP, LC.DDC, CLC	CLC
STC	SC.CAP, LC.DDC, CSC	CSC
L[BHWD]	L[BHWD][U].CAP	L[BHWD][U]
S[BHWD]	S[BHWD][U].CAP	S[BHWD][U]

Capstone-RISC-V Academic Version instruction(s)	CHERI-RISC-V instruction(s)	CHERIoT instruction(s)
CCSRRW	CSpecialRW	CSpecialRW

Most of the shared instructions are the ones for capability manipulations, as a result of having similar capability fields across the three ISA extensions. The basic use of capabilities, namely, explicit capability-based memory accesses, is also common in all three ISA extensions.

B.2. Differences

The differences stem from the different sets of extra features and capability types supported by the ISA extensions. For example, Capstone-RISC-V Academic Version supports linear capabilities and revocation through revocation capabilities that are found in neither CHERI-RISC-V nor CHERIoT. Moreover, CHERIoT does not support hybrid-mode memory accesses that use raw addresses in place of explicit capabilities, or domain switches that involve atomic swapping of sealed execution contexts, and hence lacks the relevant instructions.

While Capstone-RISC-V Academic Version and CHERI-RISC-V both have hybrid mode support, they adopt different models, with Capstone-RISC-V Academic Version (more specifically, *TransCapstone*) using a two-world model that aligns with its high-level goal of isolating pure capability code from privileged legacy code. Sealed capabilities in Capstone-RISC-V Academic Version are also different from those in CHERI-RISC-V and CHERIoT. Capstone-RISC-V Academic Version uses sealed capabilities exclusively for protecting domain execution contexts, allowing unsealing only upon domain switching, whereas the other two ISA extensions find more generic use for them and allow software to unseal them explicitly through an instruction.

The feature sets of the three ISA extensions are summarised in the table below.

Table 17. Feature sets of Capstone-RISC-V Academic Version, CHERI-RISC-V, and CHERIoT

Feature	Capstone-RISC-V Academic Version	CHERI-RISC-V	CHERIoT
Linear capabilities	Y	-	-
Revocation	Revocation capabilities with tracked derivation	Local capabilities	Local capabilities, revocation bits bound to object memory locations, local capabilities
Capability load	Anyone can load capabilities	`Permit_Load_Capability` required	`Permit_Load_Capability` required
Capability store	Anyone can store capabilities	`Permit_Store_Capability` or `Permit_Store_Local_Capability` required	`Permit_Store_Capability` or `Permit_Store_Local_Capability` required
Memory zeroing	Uninitialised capabilities	-	-

Feature	Capstone-RISC-V Academic Version	CHERI-RISC-V	CHERIoT
Software-defined fields	-	Y	Y
Hybrid mode	Separate normal and secure worlds, with MMU for integer address accesses in normal world	Default data capability for integer address accesses	-
Explicit sealing	Anyone can seal	`Permit_Seal` required	`Permit_Seal` required
Implicit sealing upon domain switching	Y	-	-
Explicit unsealing	-	Matching `otype` and `Permit_Unseal` required	Matching `otype` and `Permit_Unseal` required
Implicit unsealing upon domain switching	Anyone can perform domain switching	Matching `otype` and `Permit_CInvoke` sealed entry capabilities for code and data required	-

Bibliography

- [1] Robert N M Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A Theodore Markettos, Simon W Moore, Steven J Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8).
- [2] Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Wesley Filardo, Kunyan Liu, Robert M Norton, Yucong Tao, Robert N M Watson, and Hongyan Xia. CHERIoT: Rethinking security for low-cost embedded systems.

Appendix C: Assembly Code Examples

Appendix D: Abstract Binary Interface (Non-Normative)