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 RV64I 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.
- 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-RISC-V ISA Variant

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

In *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. Correspondingly, *TranCapstone* allows softwares to run in either of the 2 *worlds*, i.e., the *normal world* and the *secure world*.

World	MMU	Capabilities
Normal World	Yes	Yes
Secure World	-	Yes

- 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* is limited to memory accesses through capabilities and provides the security guarantees of Capstone.

TransCapstone provides an isolation between the 2 worlds, which prevents the *secure world* from being compromised by the *normal world*. A world switching mechanism is added in *TransCapstone* to allow a secure switching between the 2 worlds. The *secure world* of *TransCapstone* is mostly the same as *Pure Capstone*, and separate descriptions are provided for the parts that are different.

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 RV64I.

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, `6` = exit
`cursor`	`0..2^XLEN-1`	Not applicable when `type = 2` (revocation), `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, 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 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	-	-
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 3. 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

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

When pc contains a capability, some of the fields of the capability are checked before each instruction fetch. An exception is raised when any of the following conditions are met (in priority order):

- 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, the instruction pointed to by `pc.cursor` is fetched and executed. The `pc.cursor` is then incremented by `4` (i.e., `pc.cursor += 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	Allowed Instructions
`[0, 2^XLEN)`	Capabilities	LDD/LDW/LDH/LDB, STD/STW/STH/STB, LDC and STC

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.

Memory Region	Address Space	Access Method	Allowed Instructions
Normal Memory	`[0, SBASE) U [SEND, 2^XLEN)`	MMU	RV64I load/store instructions, LDCR and STCR
Secure Memory	`[SBASE, SEND)`	Capabilities	LDD/LDW/LDH/LDB, STD/STW/STH/STB, LDC and STC

2.5. Added Registers

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

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

<table>
<tr><th>Capstone Variant</th><th>Additional Registers</th></tr>
<tr><td>Pure Capstone</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>cih</td><td>0x001</td><td>-</td><td>The sealed capability for the interrupt 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>0x010</td><td>-</td><td>The initial capability covering the entire address space of the memory</td></tr>
<tr><td>cis</td><td>-</td><td>0x000</td><td>The interrupt status register</td></tr>
<tr><td>tval</td><td>-</td><td>0x001</td><td>The exception data (trap value) register</td></tr>
<tr><td>cause</td><td>-</td><td>0x002</td><td>The exception cause register</td></tr>
</table>
TODO: we might reuse the encoding space for some RV64I CSRs.
</td></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>0x010</td><td>-</td><td>The initial capability covering the entire address space of the secure memory</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>0x001</td><td>The exception data (trap value) register</td></tr>
<tr><td>cause</td><td>-</td><td>0x002</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 5. 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
`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 6. 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 initialize the system. In the initialisation phase of the system, the `cinit` CCSR is set to an initial capability as shown in the table below.

Table 7. Initial Capability of `cinit`

Variant	`type`	`cursor`	`base`	`end`	`perms`	`valid`
Pure Capstone	`1` (linear)	`0`	`0`	`2^XLEN`	`4` (read-write-execute)	`1` (valid)
TransCapstone	`1` (linear)	`SBASE`	`SBASE`	`SEND`	`4` (read-write-execute)	`1` (valid)

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. Any attempt to write

`cinit` or read it for the second time will be silently ignored.

2.6. Instruction Set

The Capstone-RISC-V instruction set is based on the RV64I 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 RV64I 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*).

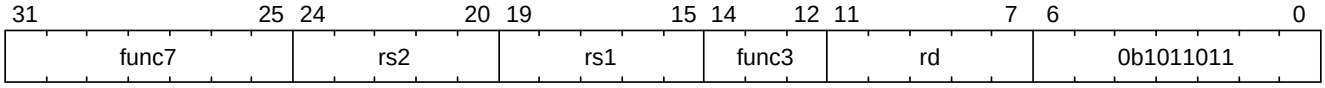

Figure 1. R-type instruction format

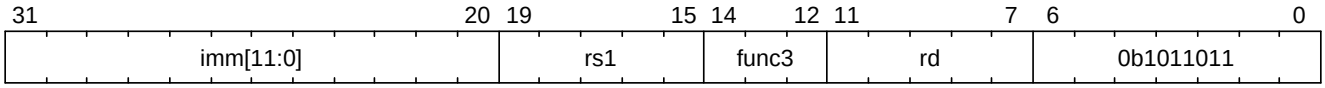

Figure 2. I-type instruction format

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.

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.

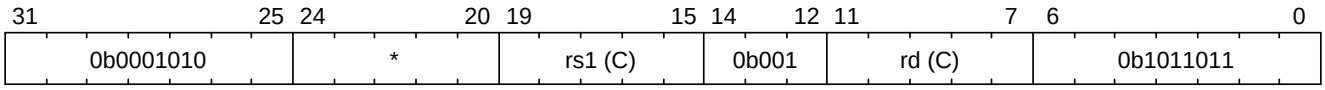

Figure 3. MOVC instruction format

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, write `x[rs1]` to `x[rd]`, and if `x[rs1]` is not a non-linear capability (i.e., `type != 1`) or an exit capability (i.e., `type != 6`), 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).

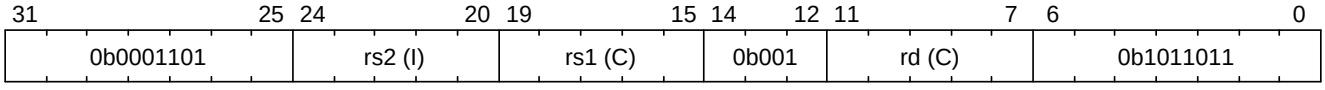

Figure 4. CINCOFFSET instruction format

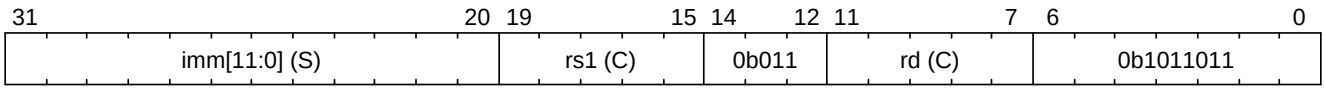

Figure 5. CINCOFFSETIMM instruction format

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

- Unexpected operand type (24)
  - `x[rs1]` is not a capability.
  - `x[rs2]` is not an integer (for CINCOFFSET).
- Unexpected capability type (26)

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

If no exception is raised: For CINCOFFSET, the offset is read from `x[rs2]`. For CINCOFFSETIMM, the offset is the 12-bit sign-extended immediate field `imm`. If the offset is `0`, the instructions are semantically equivalent to `MOVC rd, rs1`. Otherwise, the instructions are equivalent to an atomic execution of `MOVC rd, rs1` followed by an increment of `x[rd].cursor` by the offset.

3.1.3. Cursor Setter

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

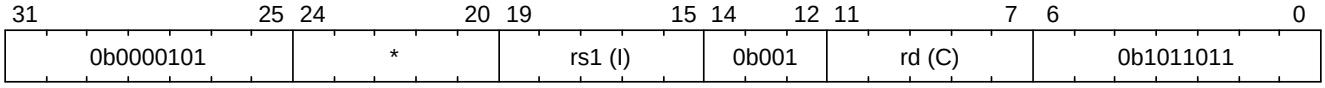

Figure 6. SCC instruction format

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

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

3.1.4. Field Getter

The `cursor` field of a capability can also be directly set and read with the SCC and LCC instructions respectively.

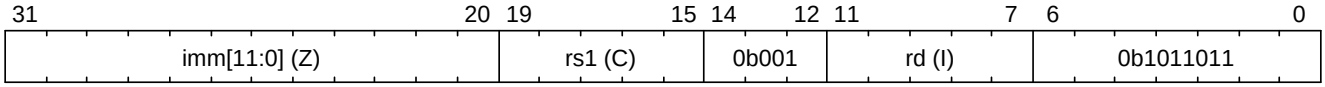

Figure 7. LCC instruction format

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

- Unexpected operand type (24)
  - `x[rs1]` is not a capability.
- Illegal operand value (29)
  - The immediate value `imm` is greater than `6`.
  - The immediate value `imm` is `0` and `x[rs1]` does not have `type = 0` (linear), `type = 1` (non-linear), or `type = 3` (uninitialised).
  - The immediate value `imm` is `2` and `x[rs1]` has `type = 6` (exit).
  - The immediate value `imm` is `3` and `x[rs1]` has `type = 4` (sealed), `type = 5` (sealed-return), or `type = 6` (exit).
  - The immediate value `imm` is `4` and `x[rs1]` has `type = 4` (sealed), `type = 5` (sealed-return), or

`type = 6` (exit).

- The immediate value `imm` is `5` and `x[rs1]` does not have `type = 4` (sealed) or `type = 5` (sealed-return).
- The immediate value `imm` is `6` and `x[rs1]` does not have `type = 5` (sealed-return).

If no exception is raised: Depending on the immediate value `imm`, the instruction write different fields of `x[rs1]` to `x[rd]` according to the following table:

`imm`	**Field read**
`0`	`cursor`
`1`	`type`
`2`	`base`
`3`	`end`
`4`	`perms`
`5`	`async`
`6`	`reg`

3.1.5. Bounds Shrinking

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

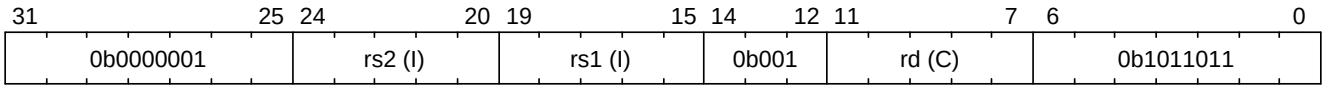

Figure 8. SHRINK instruction format

The instruction attempts to set the bounds of the capability `x[rd]` to `[x[rs1], x[rs2])`.

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.
- Illegal operand value (29)
  - `x[rd].type` is not `0`, `1`, or `3` (linear, non-linear, or uninitialised).
  - `x[rs1] >= x[rs2]`.
  - `x[rs1] < x[rd].base` or `x[rs2] > x[rd].end`.

3.1.6. Bounds Splitting

The SPLIT instruction can split a capability into two by splitting the bounds.

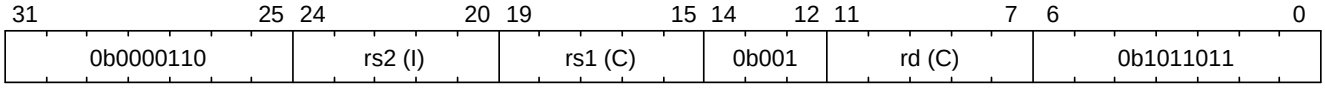

Figure 9. SPLIT instruction format

The instruction 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)`.

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` nor `1` (neither linear nor non-linear).
- Illegal operand value (29)
  - `x[rs2]` is not an integer.
  - `x[rs2] <= x[rs1].base` or `x[rs2] >= x[rs1].end`.

If no exception is raised: Set `x[rs1].end` to `x[rs2]`. A new capability is created with `base = x[rs2]` and the other fields equal to those of the original `x[rs1]`. The new capability is written to `x[rd]`.

3.1.7. Permission Tightening

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

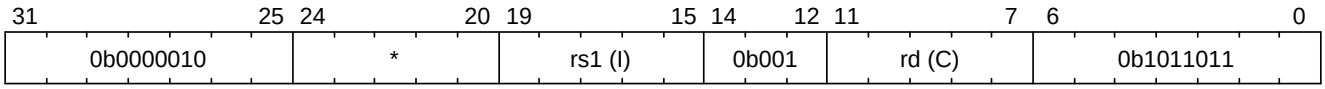

Figure 10. TIGHTEN instruction format

The instruction attempts to set `x[rd].perms` to `x[rs1]`.

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.
- Unexpected capability type (26)
  - `x[rd].type` is not `0`, `1`, or `3` (linear, non-linear, or uninitialised).
- Illegal operand value (29)
  - `x[rs1]` is outside the range of `perms`.
  - `x[rs1] <=p x[rd].perms` does not hold.

3.2. Type Manipulation

Some instructions affect the `type` field of a capability.

3.2.1. Delinearisation

The DELIN instruction delinearises a linear capability.

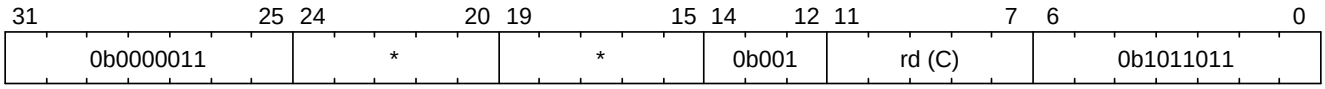

Figure 11. DELIN instruction format

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: `x[rd].type` is set 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).

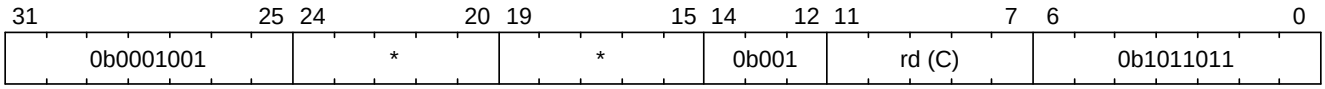

Figure 12. INIT instruction format

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

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

If no exception is raised: `x[rd].type` is set to `0` (linear).

3.2.3. Sealing

The SEAL instruction seals a linear capability.

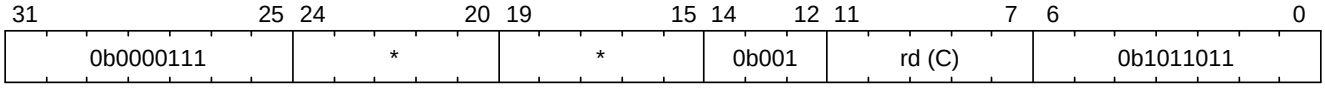

Figure 13. SEAL instruction format

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

- Unexpected operand type (24)
  - `x[rd]` is not a capability.
- Unexpected capability type (26)
  - `x[rd].type` is not `0` (linear).
- Insufficient capability permissions (27)
  - `6 <=p x[rd].perms` does not hold.
- Capability out of bound (28)
  - The size of the memory region associated with `x[rd]` is smaller than `CLENBYTES * 34` bytes. That is, `x[rd].end - x[rd].base < CLENBYTES * 34`.

If no exception is raised: `x[rd].type` is set to `2` (sealed), and `x[rd].async` is set to `0` (synchronous).

3.3. Dropping

TODO: check whether dropping is actually necessary.

The DROP instruction invalidates a capability.

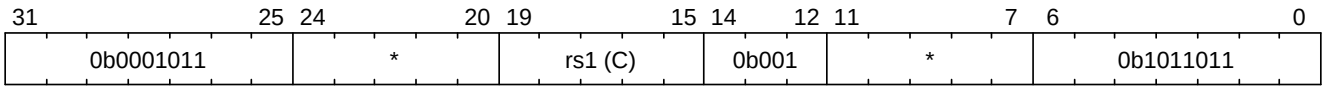

Figure 14. DROP instruction format

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: `x[rs1].valid` is set to `0` (invalid).

3.4. Revocation

3.4.1. Revocation Capability Creation

The MREV instruction creates a revocation capability.

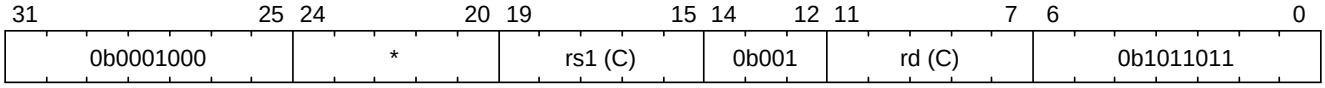

Figure 15. MREV instruction format

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: A new capability is created in `x[rd]` with the same `base`, `end`, `perms` and `cursor` fields as `x[rs1]`. The `type` field of the new capability is set to `2` (revocation).

3.4.2. Revocation Operation

The REVOKE instruction revokes a capability.

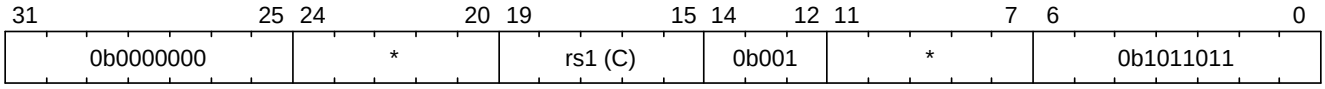

Figure 16. REVOKE instruction format

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:

For all capabilities `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`.

`x[rs1].type` is set to `0` (linear) if any of the following conditions are met for each invalidated `c`:

- The type of `c` is non-linear (i.e., `c.type != 1`)
- `2 <=p c.perms` does not hold

Otherwise, `x[rs1].type` is set to `3` (uninitialised), and `x[rs1].cursor` is set to `x[rs1].base`.

4. Memory Access Instructions

Capstone provides instructions to load from and store to memory regions using capabilities as well as instructions to load and store capabilities.

4.1. Load/Store with Capabilities

Capstone offers a set of instructions for loading and storing integers of various sizes using capabilities.

4.1.1. Load

The LDD, LDW, LDH, LDB instructions load an integer in the size of doubleword, word, halfword, and bste respectively. In Capstone, a doubleword is defined as `XLENBYTES` bytes, a word, halfword, and byte are defined as `XLENBYTES/2`, `XLENBYTES/4`, and `XLENBYTES/8` bytes respectively.

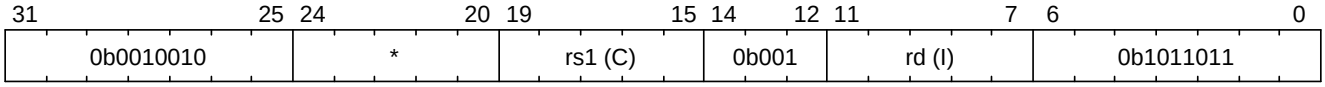

Figure 17. LDD instruction format

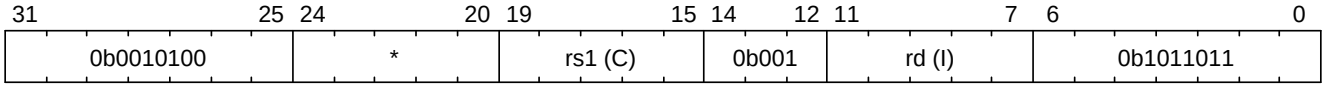

Figure 18. LDW instruction format

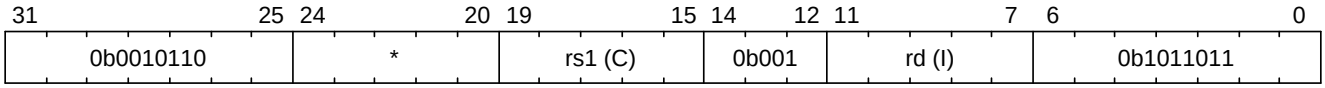

Figure 19. LDH instruction format

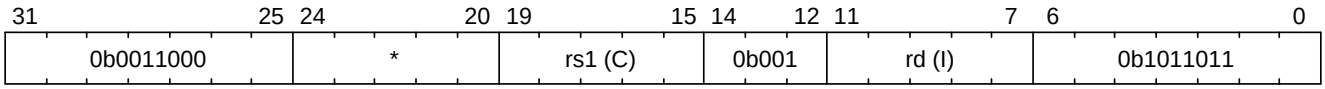

Figure 20. LDB instruction format

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` (lienar) or `1` (non-linear) and x[rs1].cursor` is not in the range `[x[rs1].base, x[rs1].end-size]`, where `size` is the size (in bytes) of the integer being loaded.
- `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 - size]`, where `size` is the size (in bytes) of the integer being loaded.

- Load address misaligned (4)
  - `x[rs1].cursor` is not aligned to the size of the integer being loaded.

If no exception is raised: Load the content at the memory location `[x[rs1].cursor, x[rs1].cursor + size)` as an integer, where `size` is the size of the integer (i.e., `XLENBYTES`, `XLENBYTES/2`, `XLENBYTES/4`, or `XLENBYTES/8` bytes for LDD, LDW, LDH, and LDB respectively), to `x[rd]`.

4.1.2. Store

The STD, STW, STH, STB instructions store an integer in the size of doubleword, word, halfword, and byte respectively.

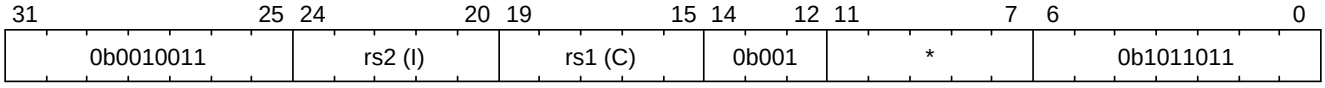

Figure 21. STD instruction format

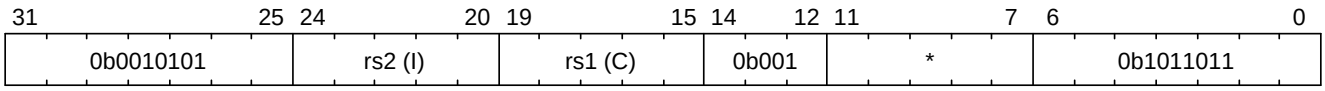

Figure 22. STW instruction format

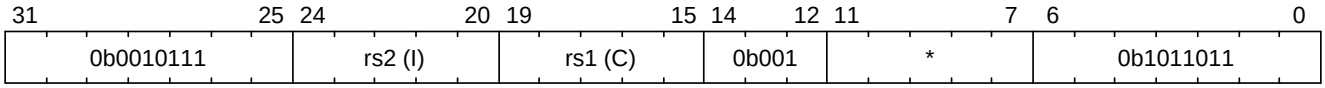

Figure 23. STH instruction format

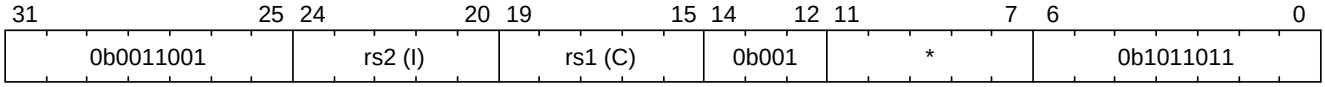

Figure 24. STB instruction format

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 not `0`, `1`, `3`, `5`, or `6` (linear, non-linear, uninitialized, sealed-return, or exit).
- Insufficient capability permissions (27)
  - `x[rs1].type` is `0`, `1`, or `3` 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` is not in the range `[x[rs1].base, x[rs1].end-size]`, where `size` is the size (in bytes) of the integer being stored.
  - `x[rs1].type` is `5` or `6` and `x[rs1].cursor` is not in the range `[x[rs1].base + 34 * CLENBYTES, x[rs1].base + 34 * CLENBYTES -size]`, where `size` is the size (in bytes) of the integer being stored.
- Store/AMO address misaligned (6)
  - `x[rs1].cursor` is not aligned to the size of the scalar value being loaded.

If no exception is raised: Store the integer in `x[rs2]` to the memory location `[x[rs1].cursor, x[rs1].cursor + size)`, where `size` is the size of the integer (i.e., `XLENBYTES`, `XLENBYTES/2`, `XLENBYTES/4`, or `XLENBYTES/8` bytes for STD, STW, STH, and STB respectively). `x[rs1].cursor` is set to `x[rs1].cursor + size`. The data contained in the `CLEN`-bit aligned memory location `[cbase, cend)`, which alias with memory location `[cursor, cursor + size)` (i.e., `cbase = cursor & ~(CLENBYTES - 1)` and `cend = cbase + CLENBYTES`), will be interpreted as an integer type.

4.2. Load/Store Capabilities

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

4.2.1. Load Capabilities

The LDC instruction loads a capability from memory.

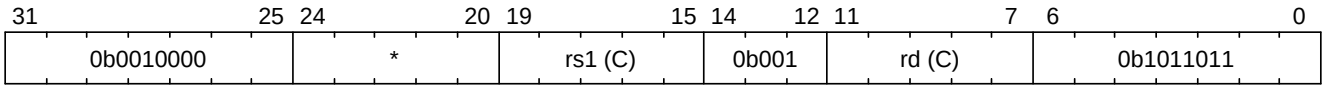

Figure 25. LDC instruction format

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` is not aligned to `CLEN` bits.
- Unexpected operand type (24) (TODO)
  - The data contained in the memory location `[x[rs1].cursor, x[rs1].cursor + 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, x[rs1].cursor + 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, x[rs1].cursor + CLENBYTES)` will be set to the content of `cnull`.

4.2.2. Store Capabilities

The STC instruction stores a capability to memory.

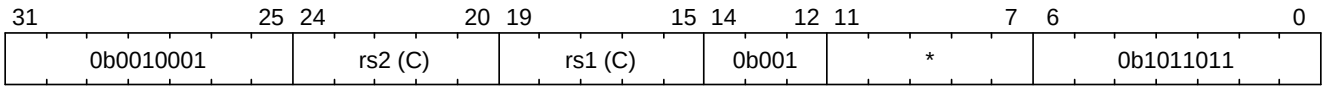

Figure 26. STC instruction format

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).
- 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` is not in the range `[x[rs1].base, x[rs1].end - CLENBYTES]`.
  - `x[rs1].type` is `5` or `6` and `x[rs1].cursor` is not in the range `[x[rs1].base, x[rs1].base + 34 * CLENBYTES - CLENBYTES]`.
- Store/AMO address misaligned (6)
  - `x[rs1].cursor` is not aligned to `CLEN` bits.

If no exception is raised: Store `x[rs2]` to the memory location `[x[rs1].cursor, x[rs1].cursor +`

CLENBYTES). 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.

4.3. *TransCapstone* Added Instructions

In *TransCapstone*, besides the LDC and STC instructions, two additional instructions (i.e., LDCR and STCR) are added to load and store capabilities from/to the normal memory using raw addresses. These 2 instructions are only available in *TransCapstone* and an exception will be raised if they are executed in *Pure Capstone*.

4.3.1. Load with Raw Addresses

The LDCR instruction loads a capability from the normal memory using raw addresses.

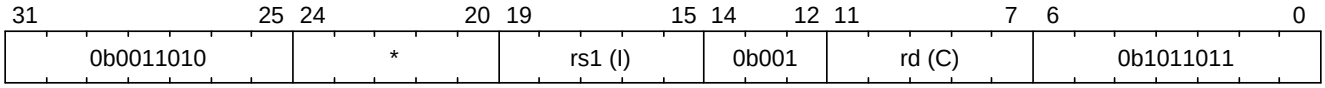

Figure 27. LDCR instruction format

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

If no exception is raised: Load the capability at the memory location [x[rs1], x[rs1] + CLENBYTES) into rd. If the capability being loaded is a non-linear capability (i.e. type != 1) or an exit capability (i.e., type != 6), the data contained in the memory location [x[rs1], x[rs1] + CLENBYTES) will be set to the content of cnull.

4.3.2. Store with Raw Addresses

The STCR instruction stores a capability to the normal memory using raw addresses.

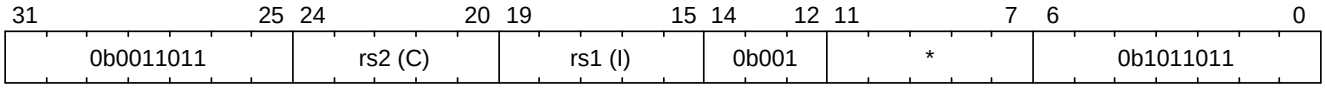

Figure 28. STCR instruction format

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]` is not aligned to `CLEN` bits.
- Store/AMO access fault (7)
    - `x[rs1]` is in the range `[SBASE, SEND)`.

If no exception is raised: Store `x[rs2]` to the memory location `[x[rs1], x[rs1] + CLENBYTES)`. If `x[rs2]` is not a non-linear capability (i.e., `type != 1`) or an exit capability (i.e., `type != 6`), `x[rs2]` will be set to the content of `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.

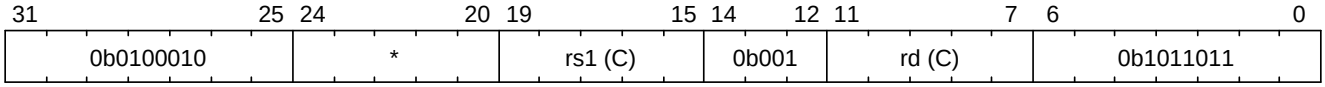

Figure 29. CJALR instruction format

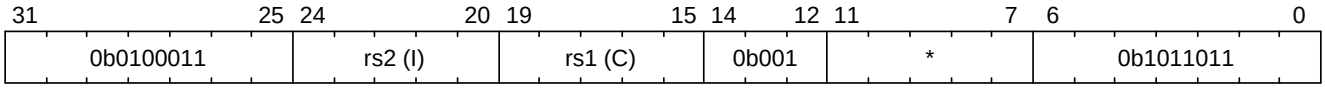

Figure 30. CBNZ instruction format

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.
  - Unexpected capability type (26)
    - `x[rs1].type` is neither `0` (linear) nor `1` (non-linear).
  - Insufficient capability permissions (27)
    - `1 <=p x[rs1].perms` does not hold.

If no exception is raised:

- CJAL: Set the program counter (`pc`) to `x[rs1]`. Meanwhile, the existing capability in `pc`, with its `cursor` field replaced by the address of the next instruction, is written to the register `rd`.
- CBNZ: If `x[rs2]` is zero (`0`), the behaviour is the same as for NOP. Otherwise, set the program counter (`pc`) to `x[rs1]`.

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*.

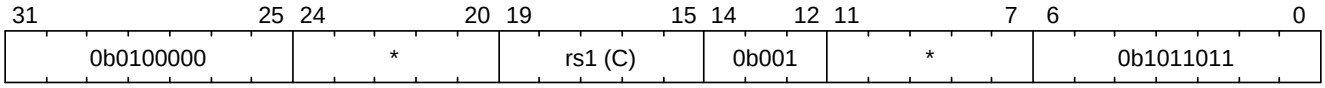

Figure 31. CALL instruction format

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. Load the content at the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)` to the program counter (`pc`).
2. Load the content at the memory location `[x[rs1].base + CLENBYTES, x[rs1].base + 2 * CLENBYTES)` to `ceh`.
3. Load the content at the memory location `[x[rs1].base + 2 * CLENBYTES, x[rs1].base + 3 * CLENBYTES)` to `csp`.
4. Store the former `pc`, `ceh` and `csp` values to the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)`, `[x[rs1].base + CLENBYTES, x[rs1].base + 2 * CLENBYTES)` and `[x[rs1].base + 2 * CLENBYTES, x[rs1].base + 3 * CLENBYTES)` respectively.
5. Set `x[rs1].type` to `5` (sealed-return), `x[rs1].cursor` to `x[rs1].base`, x[rs1].reg` to `rd`, set `x[rs1].async` to `0` (synchronous), and write the resulting `x[rs1]` to the register `cra`.

5.2.2. RETURN

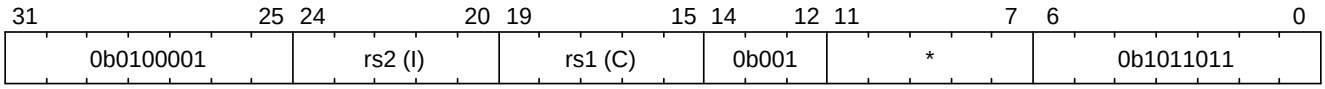

Figure 32. RETURN instruction format

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)
    - `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 `5` (sealed-return).

If no exception is raised:

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

1. Load the content at the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)` to the program counter (`pc`).
2. Load the content at the memory location `[x[rs1].base + CLENBYTES, x[rs1].base + 2 * CLENBYTES)` to `ceh`.
3. Load the content at the memory location `[x[rs1].base + 2 * CLENBYTES, x[rs1].base + 3 * CLENBYTES)` to `csp`.
4. Store the former `pc` with `cursor` replaced with `x[rs2]`, `ceh` and `csp` values to the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)`, `[x[rs1].base + CLENBYTES, x[rs1].base + 2 * CLENBYTES)` and `[x[rs1].base + 2 * CLENBYTES, x[rs1].base + 3 * CLENBYTES)` respectively.
5. Set `x[rs1].type` to `4` (sealed), and write the capability to the register `x[x[rs1].reg]`.

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

1. Load the content at the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)` to the program counter (`pc`).
2. Load the content at the memory location `[x[rs1].base + CLENBYTES, x[rs1].base + 2 * CLENBYTES)` to `ceh`.
3. Load the content at the memory location `[x[rs1].base + 2 * CLENBYTES, x[rs1].base + 3 * CLENBYTES)` to `csp`.
4. For `i = 1, 2, ⋯, 31`, load the content at the memory location `[x[rs1].base + (i + 2) * CLENBYTES, x[rs1].base + (i + 3) * CLENBYTES)`, to `x[i]` (the `i`-th general-purpose register).
5. Write the former value of `pc`, with the `cursor` field replaced by `x[rs2]`, to the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)`.
6. Store the former alue of `ceh` to the memory location `[x[rs1].base + CLENBYTES, x[rs1].base + 2 * CLENBYTES)`.

7. Store the former value of `csp` to the memory location `[x[rs1].base + 2 * CLENBYTES, x[rs1].base + 3 * CLENBYTES)`.
8. Set the `x[rs1].type` to `4` (sealed). If `x[rs1].async = 1`, write the resulting `x[rs1]` to the register `ceh`. Otherwise (`x[rs1].async = 2`), write the resulting `x[rs1]` to the register `cih`.

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. The CAPENTER instruction causes an entry into the secure world from the normal world, and the CAPEXIT instruction causes an exit from the secure world into the normal world.

The CAPENTER instruction can only be used in the normal world, whereas the CAPEXIT instruction can only be used in the secure world. In addition, the CAPEXIT instruction can only be used when an exit capability is provided. Attempting to use those instructions in the wrong world or without the required capability will cause an exception. The behaviours of these 2 instructions roughly correspond to the CALL and RETURN instructions respectively.

5.3.1. CAPENTER

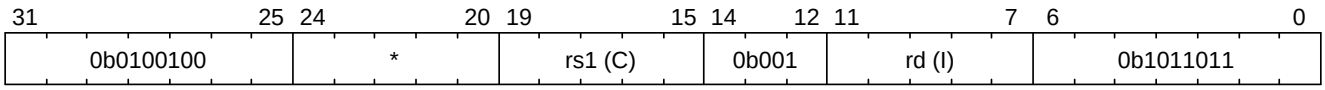

Figure 33. CAPENTER instruction format

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. Load the content at the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)` to the program counter (`pc`).
2. Load the content at the memory location `[x[rs1].base + CLENBYTES, x[rs1].base + 2 * CLENBYTES)` to `ceh`.
3. Load the content at the memory location `[x[rs1].base + 2 * CLENBYTES, x[rs1].base + 3 * CLENBYTES)` to `csp`.

4. Store the former value of `pc` and `sp` to `normal_pc` and `normal_sp` respectively.
5. Set `x[rs1].type` to `6` (exit), `x[rs1].cursor` to `x[rs1].base`, and write the resulting `x[rs1]` to `cra`.
6. Write `rs1` to `switch_reg`.
7. Write `rd` to `exit_reg`.
8. Set `cwrld` to `1` (secure world).

Otherwise:

1. Load the content at the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)` to the program counter (`pc`).
2. Load the content at the memory location `[x[rs1].base + CLENBYTES, x[rs1].base + 2 * CLENBYTES)` to `ceh`.
3. Load the content at the memory location `[x[rs1].base + 2 * CLENBYTES, x[rs1].base + 3 * CLENBYTES)` to `csp`.
4. For `i = 1, 2, ⋯, 31`, load the content at the memory location `[x[rs1].base + (i + 2) * CLENBYTES, x[rs1].base + (i + 3) * CLENBYTES)`, to `x[i]` (the `i`-th general-purpose register).
5. Store the former value of `pc` and `sp` to `normal_pc` and `normal_sp` respectively.
6. Set `x[rs1].type` to `5` (sealed-return), `x[rs1].cursor` to `x[rs1].base`, `x[rs1].async` to `0` (synchronous), and write the resulting `x[rs1]` to `switch_cap`.
7. Write `rs1` to `switch_reg`.
8. Write `rd` to `exit_reg`.
9. Set `cwrld` to `1` (secure world).

> **Note**
>
> 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

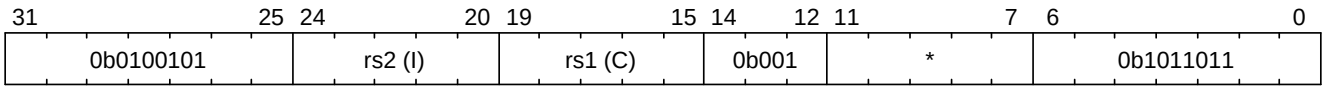

Figure 34. CAPEXIT instruction format

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

6. Control and Status Instructions

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

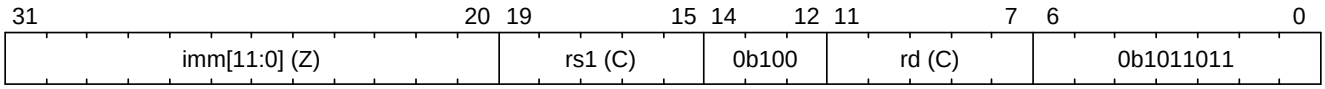

Figure 35. CCSRRW instruction format

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 the RV64I ISA, 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. For control flow instructions and memory access instructions, however, the behaviour is slightly changed to be capability-aware.

7.1. Control Flow Instructions

In RV64I, a set of instructions are used to control the flow of execution. These instructions include conditional branch instructions (i.e., `beq`, `bne`, `blt`, `bge`, `bltu`, and `bgeu`), and unconditional jump instructions (i.e., `jal` and `jalr`). In Capstone, adjustments are made to these instructions to support capability-aware execution.

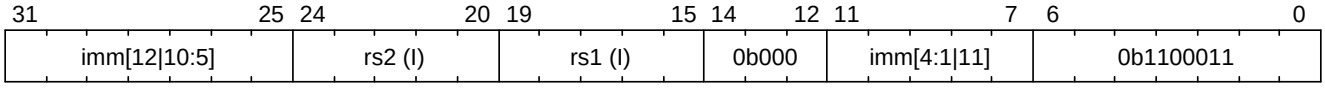

Figure 36. beq instruction format (B-type)

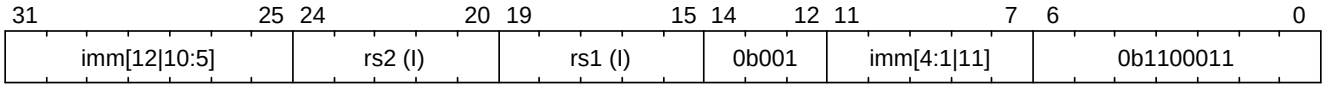

Figure 37. bne instruction format (B-type)

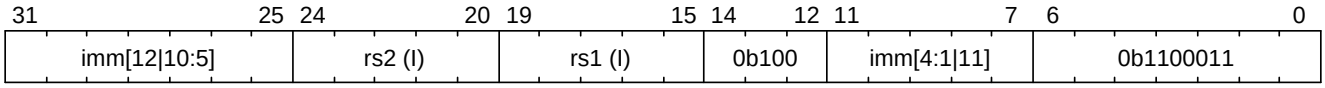

Figure 38. blt instruction format (B-type)

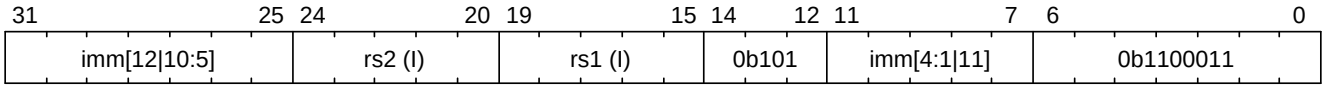

Figure 39. bge instruction format (B-type)

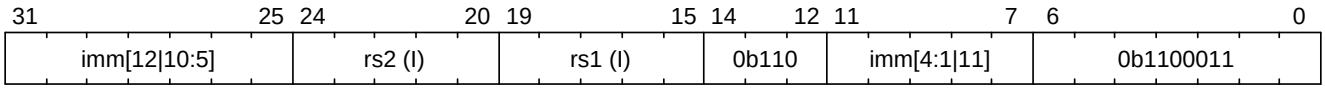

Figure 40. bltu instruction format (B-type)

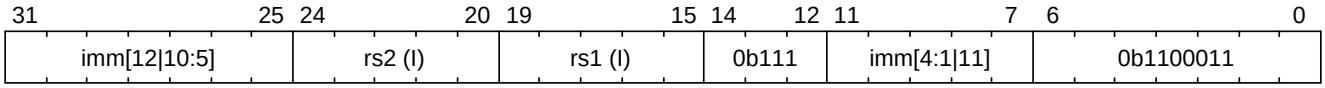

Figure 41. bgeu instruction format (B-type)

Figure 42. jal instruction format (J-type)

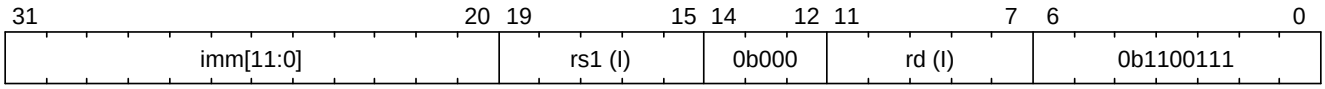

Figure 43. jalr instruction format (I-type)

The following adjustments are made to these instructions:

- *Pure Capstone*
  - An “unexpected operand type (24)” exception is raised if `x[rs1]`, `x[rs2]` or `x[rd]` is a capability.
  - `pc.cursor`, instead of `pc` itself, is changed by the instruction.
  - If the instruction is `jal` or `jalr`, `pc.cursor`, which contains the address of the next instruction, is written to `x[rd]`.
- *TransCapstone*
  - An “unexpected operand type (24)” exception is raised if `x[rs1]`, `x[rs2]` or `x[rd]` contains a capability.
  - If `cwrld` is `1` (secure world), `pc.cursor`, instead of `pc` itself, is changed by the instruction.
  - If `cwrld` is `1` (secure world) and the instruction is `jal` or `jalr`, `pc.cursor` (i.e., the address of the next instruction), is written to `x[rd]`.

7.2. Memory Access Instructions

In RV64I, memory access instructions include load instructions (i.e., `lb`, `lh`, `lw`, `lbu`, `lhu`, `lwu`, `ld`, and `fld`), and store instructions (i.e., `sb`, `sh`, `sw`, `sd`, and `fsd`). As the Capstone-RISC-V ISA extends each of the 32 general-purpose registers, instructions that take these registers as operands are also extended. These instructions (i.e., `lb`, `lh`, `lw`, `lbu`, `lhu`, `lwu`, `ld`, `sb`, `sh`, `sw`, and `sd`) take an integer as a raw address, and load or store a value from or to this address. In Capstone, adjustments are made to these instructions to support capability-aware memory access.

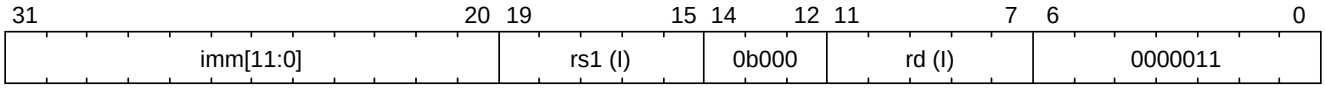

Figure 44. lb instruction format (I-type)

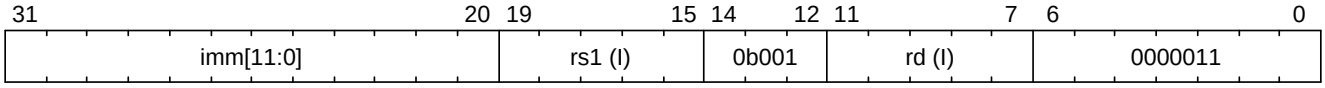

Figure 45. lh instruction format (I-type)

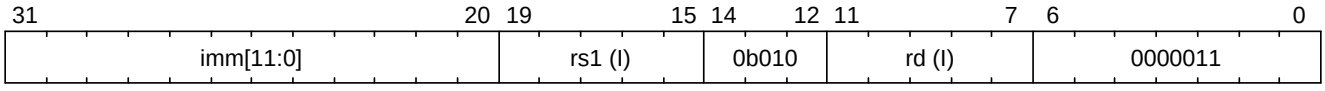

Figure 46. lw instruction format (I-type)

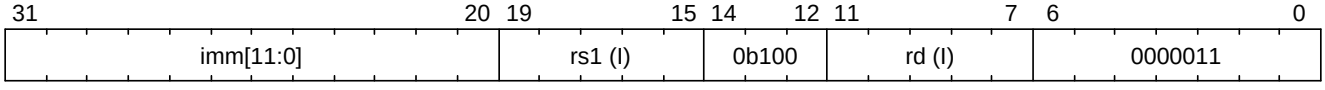

Figure 47. lbu instruction format (I-type)

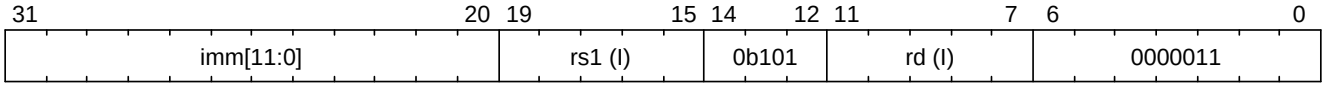

Figure 48. lhu instruction format (I-type)

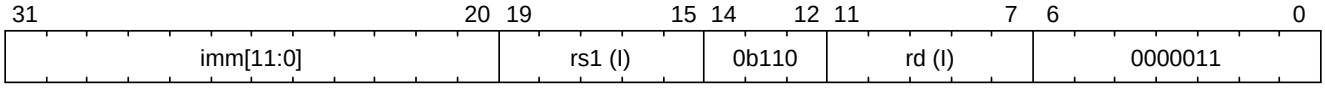

Figure 49. lwu instruction format (I-type)

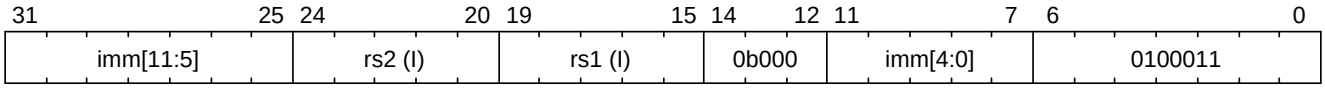

Figure 50. sb instruction format (S-type)

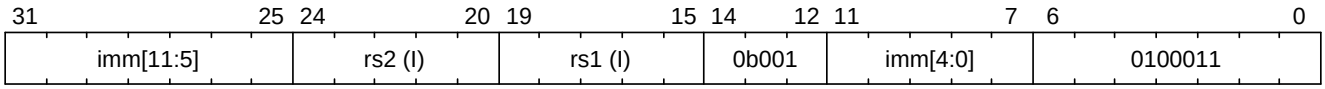

Figure 51. sh instruction format (S-type)

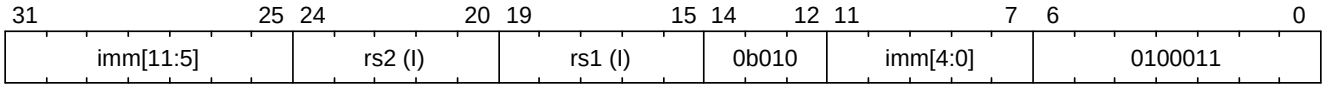

Figure 52. sw instruction format (S-type)

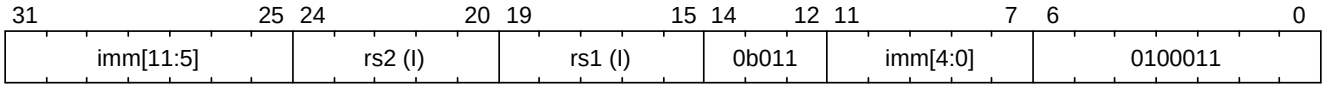

Figure 53. sd instruction format (S-type)

The following adjustments are made to these instructions:

- *Pure Capstone*
  - An “illegal instruction (2)” exception is raised if any of these instructions is executed.
- *TransCapstone*
  - An “illegal instruction (2)” exception is raised if any of these instructions is executed when `cwrld` is `1` (secure world).
  - An “unexpected operand type (24)” exception is raised if `x[rs1]`, `x[rs2]` or `x[rd]` contains a capability.
  - Depending on whether the instruction is a load or a store, a “load access fault (5)” or “store/AMO access fault (7)” exception is raised if the address to be accessed is within the range `(SBASE-size, SEND)` (i.e. `addr = x[rs1] + sext(imm)` and `SBASE-size < addr < SEND`), where `size` is the size (in bytes) of the integer to be loaded or stored.

8. Interrupts and Exceptions

TODO: add support for nesting

8.1. Exception and Exit Codes

Note

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 some form of exception codes:

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 to pass to the exception handler domain will be in the register `a1`, and the `rd` operand of CAPENTER will be the exit code the user process receives.

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 RV64I, 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
Unexpected capability type	26	1

Exception	Exception code	TransCapstone exit code
Insufficient capability permissions	27	1
Capability out of bound	28	1
Illegal operand value	29	1
Unhandleable exception	30	N/A in TransCapstone

For interrupts, the same encodings as in RV64I are used.

Note

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 RV64I. 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 or world boundaries (i.e., when `ceh` is a sealed capability or when the normal world ends up handling the exception), the exception data 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 RV64I, 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	The instruction itself (or the lowest XLEN bits if it is wider than XLEN)
Invalid capability	The instruction itself (or the lowest XLEN bits if it is wider than XLEN)
Unhandleable exception	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, or 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 handler domain as an argument. When exception handling is complete, the exception 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.

TODO: specify what happens if any of the involved memory accesses fails

8.3.1. Interrupt Status

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

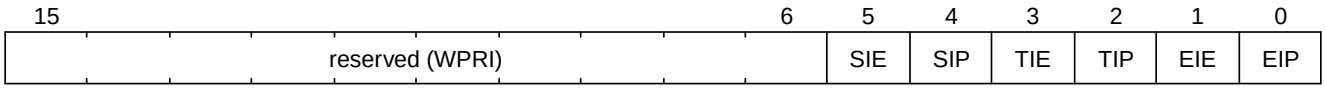

Figure 54. `cis` CSR layout

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 RV64I.

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

Note

We can require that `cih` does not 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

In this way, the `cih` register acts as a global interrupt-enable flag.

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. Load the program counter `pc` from memory location `[cih.base, cih.base + CLENBYTES)`.
2. Load `ceh` from memory location `[cih.base + CLENBYTES, cih.base + 2 * CLENBYTES]`.
3. Load `csp` from memory location `[cih.base + 2 * CLENBYTES, cih.base + 3 * CLENBYTES]`.
4. For `i = 1, 2, ⋯, 31`, load the content of `x[i]` from memory location `[cih.base + (i + 2) * CLENBYTES, cih.base + (i + 3) * CLENBYTES)`.
5. Store the original program counter `pc` to the memory location `[cih.base, cih.base + CLENBYTES)`.
6. Store the original `ceh` to the memory location `[cih.base + CLENBYTES, cih.base + 2 * CLENBYTES)`.
7. Store the original `csp` to the memory location `[cih.base + 2 * CLENBYTES, cih.base + 3 * CLENBYTES)`.
8. For `i = 1, 2, ⋯, 31`, store the *original* content of `x[i]` to memory location `[cih.base + (i + 2) * CLENBYTES, cih.base + (i + 3) * CLENBYTES)`.
9. Set `cih.type` to `5` (sealed-return), `x[rs1].cursor` to `x[rs1].base`, `cih.reg` to `0`, and `cih.async` to `2` (upon interrupt).
10. Write `cih` to the register `c1`.
11. Write the exception code to the register `x10`.
12. Write `cnull` to the register `cih`.

8.3.4. Handling of Exceptions

> **Note**
>
> 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 `26` (unhandleable exception) 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 `ceh.type = 4`:

1. Load the program counter `pc` from memory location `[ceh.base, ceh.base + CLENBYTES)`.
2. Load new `ceh` from memory location `[ceh.base + CLENBYTES, ceh.base + 2 * CLENBYTES)`.
3. Load new `csp` from memory location `[ceh.base + 2 * CLENBYTES, ceh.base + 3 * CLENBYTES)`.
4. For `i = 1, 2, ⋯, 31`, load the content of `x[i]` from memory location `[ceh.base + (i + 2) * CLENBYTES, ceh.base + (i + 3) * CLENBYTES)`.
5. Store the original program counter `pc` to the memory location `[ceh.base, ceh.base + CLENBYTES)`.
6. Store the original `ceh` to the memory location `[ceh.base + CLENBYTES, ceh.base + 2 * CLENBYTES)`.
7. Store the original `csp` to the memory location `[ceh.base + 2 * CLENBYTES, ceh.base + 3 * CLENBYTES)`.
8. For `i = 1, 2, ⋯, 31`, store the *original* content of `x[i]` to memory location `[ceh.base + (i + 2) * CLENBYTES, ceh.base + (i + 3) * CLENBYTES)`.
9. Set the original `ceh.type` to `5` (sealed-return), `x[rs1].cursor` to `x[rs1].base`, `ceh.reg` to `0`, and `ceh.async` to `1` (upon exception).
10. Write the modified content of original `ceh` to the register `c1`.
11. Write the exception code to the register `x10`.

If `ceh.type = 0` or `1`:

1. Write `pc` to `epc`.
2. Write `ceh` to `pc`.
3. If `ceh.type != 1` and `ceh.type != 6`, write `cnull` to `ceh`.
4. Write the exception code to `cause`
5. Write extra exception data to `tval`.

> **Note**
>
> The `csp` register is designed to hold data that the in-domain exception handler can utilize. 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 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 mechanims for the normal world as in RV64I.

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.

For interrupts, in order to prevent denial-of-service attacks by the secure world, the processor core needs to 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 other words, the secure world will simply resume execution from where it was interrupted after the interrupt is handled by the normal-world OS.

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 exeption (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 and follows the normal-world interrupt handling process thereafter.

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 the current `ceh` to the memory location `[switch_cap.base + CLENBYTES, switch_cap.base + 2 * CLENBYTES)`.
3. Store the current `csp` to the memory location `[switch_cap.base + 2 * CLENBYTES,`

`switch_cap.base + 3 * CLENBYTES)`.

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

Otherwise:

1. Write the content of `cnull` to `x[switch_reg]`.
2. Load the program counter `pc` and the stack pointer `sp` from `normal_pc` and `normal_sp` respectively.
3. Scrub the other general-purpose registers.
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 (`ceh` is not valid), 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. Load the program counter `pc` from memory location `[ceh.base, ceh.base + CLENBYTES)`.
2. Load new `ceh` from memory location `[ceh.base + CLENBYTES, ceh.base + 2 * CLENBYTES)`.
3. Load new `csp` from memory location `[ceh.base + 2 * CLENBYTES, ceh.base + 3 * CLENBYTES)`.
4. For `i = 1, 2, ⋯, 31`, load the content of `x[i]` from memory location `[ceh.base + (i + 2) * CLENBYTES, ceh.base + (i + 3) * CLENBYTES)`.
5. Store the original program counter `pc` to the memory location `[ceh.base, ceh.base + CLENBYTES)`.
6. Store the original `csp` to the memory location `[ceh.base + 2 * CLENBYTES, ceh.base + 3 * CLENBYTES)`.
7. For `i = 1, 2, ⋯, 31`, store the *original* content of `x[i]` to memory location `[ceh.base + (i + 2) * CLENBYTES, ceh.base + (i + 3) * CLENBYTES)`.
8. Set the `ceh.type` to `5` (sealed-return), `x[rs1].cursor` to `x[rs1].base`, and `ceh.async` to `1` (upon exception).

9. Write the content of `ceh` to the register `c1`.
10. Write the exception code to the register `x10`.

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`.
3. If `ceh.type != 1` and `ceh.type != 6`, write `cnull` to `ceh`.
4. Write the exception code to `cause`
5. 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)`.
3. Store `csp` to the memory location `[switch_cap.base + 2 * CLENBYTES, switch_cap.base + 3 * CLENBYTES)`.
4. For `i = 1, 2, ⋯, 31`, store the content of the `i`-th general purpose to the memory location `[switch_cap.base + (i + 2) * CLENBYTES, switch_cap.base + (i + 3) * CLENBYTES)`.
5. Set `switch_cap.async` to `1` (upon exception).
6. Write the content of `switch_cap` to `x[switch_reg]`.
7. Load the program counter `pc` and the stack pointer `sp` from `normal_pc` and `normal_sp` respectively.
8. Write the exit code to `exit_reg`.
9. Set the `cwrld` register to `0` (normal world).

Otherwise:

1. Write the content of `cnull` to `x[switch_reg]`.
2. Load the program counter `pc` and the stack pointer `sp` from `normal_pc` and `normal_sp` respectively.
3. Write the exit code to `exit_reg`.
4. Set the `cwrld` register to `0` (normal world).

Note

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.

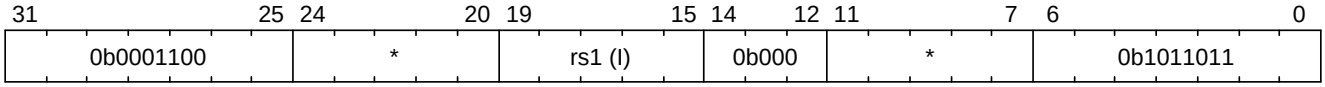

Figure 55. SETWORLD instruction format

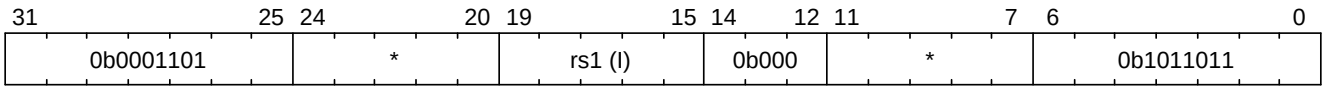

Figure 56. ONPARTITION instruction format

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.

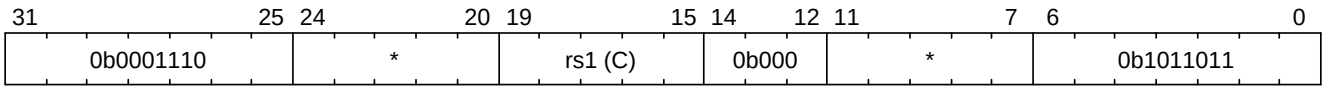

Figure 57. SETEH instruction format

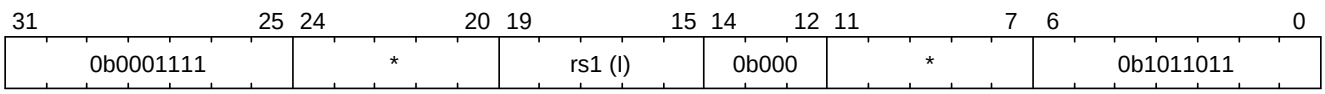

Figure 58. ONNORMALEH instruction format

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

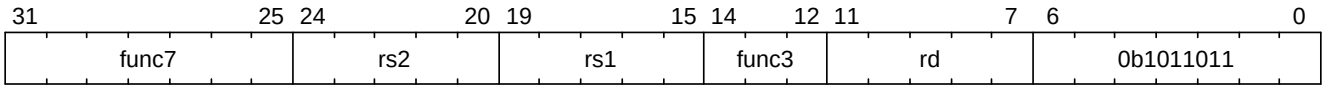

Figure 59. Instruction format: R-type

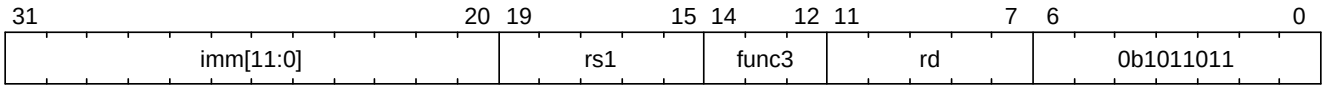

Figure 60. Instruction format: I-type

Table 10. 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

Table 11. 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	I	-	C	-	*	*
DELIN	R	001	0000011	-	-	C	-	*	*
LCC	I	001	0000100	C	-	I	Z	*	*
SCC	R	001	0000101	I	-	C	-	*	*

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]	World	Variant
SPLIT	R	001	0000110	C	I	C	-	*	*
SEAL	R	001	0000111	-	-	C	-	*	*
MREV	R	001	0001000	C	-	C	-	*	*
INIT	R	001	0001001	-	-	C	-	*	*
MOVC	R	001	0001010	C	-	C	-	*	*
DROP	R	001	0001011	C	-	-	-	*	*
CINCOFFSET	R	001	0001100	C	I	C	-	*	*
CINCOFFSETIMM	I	011	-	C	-	C	S	*	*

Table 12. Memory access instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]	World	Variant
LDC	R	001	0010000	C	-	C	-	*	*
STC	R	001	0010001	C	C	-	-	*	*
LDD	R	001	0010010	C	-	I	-	*	*
STD	R	001	0010011	C	I	-	-	*	*
LDW	R	001	0010100	C	-	I	-	*	*
STW	R	001	0010101	C	I	-	-	*	*
LDH	R	001	0010110	C	-	I	-	*	*
STH	R	001	0010111	C	I	-	-	*	*
LDB	R	001	0011000	C	-	I	-	*	*
STB	R	001	0011001	C	I	-	-	*	*
LDCR	R	001	0011010	I	-	C	-	N	T
STCR	R	001	0011011	I	C	-	-	N	T

Table 13. Control flow instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]	World	Variant
CALL	R	001	0100000	C	-	-	-	S	*
RETURN	R	001	0100001	C	I	-	-	S	*
CJALR	R	001	0100010	C	-	C	-	S	*
CBNZ	R	001	0100011	C	I	-	-	S	*
CAPENTER	R	001	0100100	C	-	I	-	N	T
CAPEXIT	R	001	0100101	C	I	-	-	S	T

Table 14. Control and status instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]	World	Variant
CCSRRW	I	100	-	C	-	C	Z	*	*

Note

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 C: Assembly Code Examples

TODO

Appendix D: Abstract Binary Interface (Non-Normative)

TODO