The Capstone-RISC-V Instruction Set Reference

Table of Contents

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

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 a 64-bit RISC-V 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. 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.

1.5. 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 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 the 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 `XLENBYTES = 8` bytes.

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 = 4` (sealed), `type = 5` (sealed-return), or `type = 6` (exit). 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..4`	Not applicable when `type = 4` (sealed), `type = 5` (sealed-return) or `type = 6` (exit). The permissions associated with the capability: `0` = no access, `1` = read-only, `2` = read-execute, `3` = read-write, `4` = read-write-execute
`count`	`0..31`	Only applicable when `type = 4` (sealed) or `type = 5` (sealed-return). The number of register values sealed in the region
`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 $\Leftarrow$ defined as follows:

```
<= = { (0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (0, 1), (0, 2), (0, 3), (0, 4),
       (1, 2), (1, 3), (1, 4), (2, 4), (3, 4) }
```

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 revocation capabilities that alias, 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).

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.

`XLEN`-bit integer	Capability
`x0`/`zero`	`c0`/`cnull`
`x1`/`ra`	`c1`/`cra`
`x2`/`sp`	`c2`/`csp`
`x3`/`gp`	`c3`/`cgp`
`x4`/`tp`	`c4`/`ctp`
`x5`/`t0`	`c5`/`ct0`
`x6`/`t1`	`c6`/`ct1`
`x7`/`t2`	`c7`/`ct2`
`x8`/`s0`/`fp`	`c8`/`cs0`/`cfp`
`x9`/`s1`	`c9`/`cs1`
`x10`/`a0`	`c10`/`ca0`
`x11`/`a1`	`c11`/`ca1`
`x12`/`a2`	`c12`/`ca2`
`x13`/`a3`	`c13`/`ca3`
`x14`/`a4`	`c14`/`ca4`
`x15`/`a5`	`c15`/`ca5`
`x16`/`a6`	`c16`/`ca6`
`x17`/`a7`	`c17`/`ca7`
`x18`/`s2`	`c18`/`cs2`
`x19`/`s3`	`c19`/`cs3`

XLEN-bit integer	Capability
x20/s4	c20/cs4
x21/s5	c21/cs5
x22/s6	c22/cs6
x23/s7	c23/cs7
x24/s8	c24/cs8
x25/s9	c25/cs9
x26/s10	c26/cs10
x27/s11	c27/cs11
x28/t3	c28/ct3
x29/t4	c29/ct4
x30/t5	c30/ct5
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).

2.3. Extension to Other Registers

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:

- The valid field of the capability in pc is 0 (invalid).
- The bound of the capability in pc is [base, end), where base and end are the base and end fields of the capability in pc, and the cursor field of the capability in pc is not in the range [base, end-4] (i.e., cursor < base or cursor > end-4).

If no exception is raised, the instruction pointed to by the cursor field of the capability in pc is fetched and executed. The cursor field of the capability in pc is then incremented by 4 (i.e., cursor += 4).

2.4. Added Registers

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

- ceh: the sealed capability for the exception handler.
- cwrld: the currently executed world. 0 = normal world, 1 = secure world.
- normal_pc: the program counter for the normal world before the secure world is entered.

- `normal_sp`: the stack pointer for the normal world before the secure world is entered.
- `switch_reg`: the index of the general-purpose register used when switching worlds.
- `switch_cap`: the capability used to store contexts when switching worlds.

TODO: talk about how to read/write those registers.

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.

2.6. Instruction Set

The Capstone-RISC-V instruction set is based on the RV64G 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 RV64G 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*).

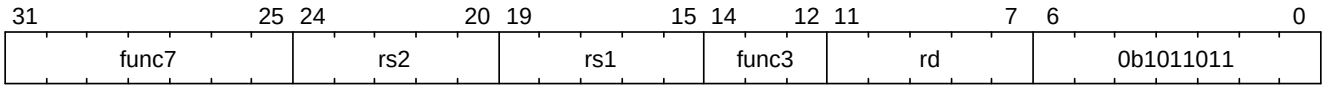

Figure 1. R-type instruction format

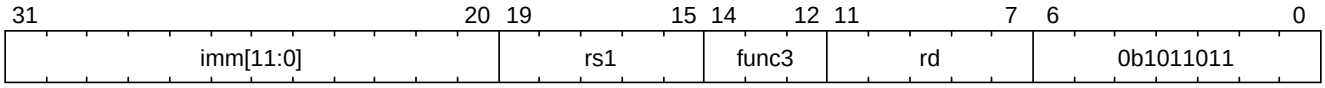

Figure 2. I-type instruction format

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.

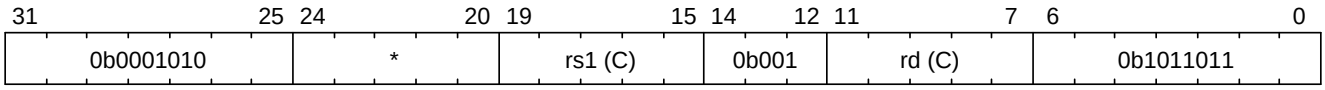

Figure 3. MOVC instruction format

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

- `rs1` does not contain a capability

If no exception is raised: If `rs1` is the same register as `rd`, the instruction is a no-op. If `rs1` is not the same register as `rd`, `rd` will contain the original content of `rs1`, and if the content is not a non-linear capability (i.e., `type != 1`) or an exit capability (i.e., `type != 6`), `rs1` will be set to the content of `cnull`.

3.1.2. Cursor Increment

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

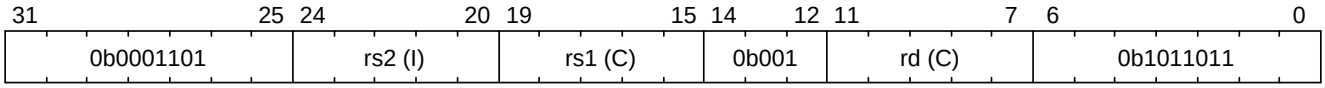

Figure 4. CINCOFFSET instruction format

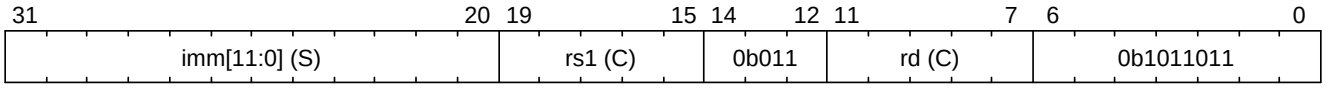

Figure 5. CINCOFFSETIMM instruction format

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

- `rs1` does not contain a capability.
- `rs2` does not contain an integer (for CINCOFFSET).
- The capability in `rs1` does not have `type = 0` (linear) or `type = 1` (non-linear).

If no exception is raised: For CINCOFFSET, the offset is read from `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 equal to an atomic

execution of `MOVC rd, rs1` followed by an increment of the `cursor` field of `rd` by the offset.

3.1.3. Cursor Setter and Getter

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

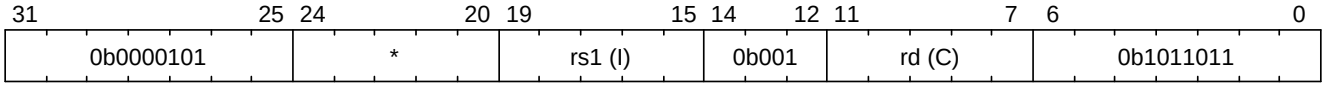

Figure 6. SCC instruction format

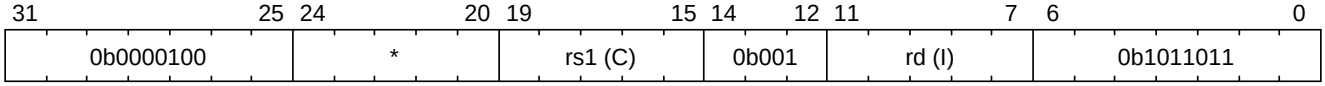

Figure 7. LCC instruction format

For SCC, an exception is raised if any of the following conditions are met:

- `rd` does not contain a capability.
- `rs1` does not contain an integer.
- The capability in `rd` does not have `type = 0` (linear) or `type = 1` (non-linear).

For LCC, an exception is raised if any of the following conditions are met:

- `rs1` does not contain a capability.
- The capability in `rs1` does not have `type = 0` (linear), `type = 1` (non-linear), or `type = 3` (uninitialised).

3.1.4. Bounds Shrinking

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

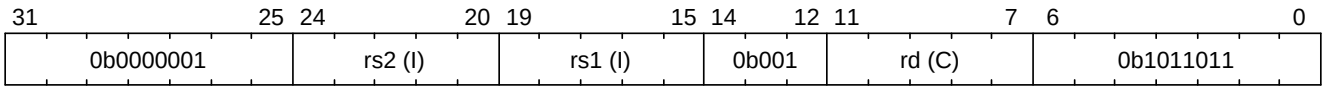

Figure 8. SHRINK instruction format

The instruction reads `rs1` and `rs2` and attempts to set the bounds of the capability in `rd` to `[rs1, rs2)`.

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

- `rd` does not contain a capability.
- The `valid` field of the capability in `rd` is `0` (invalid).
- The `type` field of the capability in `rd` is not `0`, `1`, or `3` (linear, non-linear, or uninitialised).
- `rs1` does not contain an integer.
- `rs2` does not contain an integer.
- `rs1 >= rs2`.

- The original bounds of the capability in `rd` are `[base, end)` and `rs1 < base` or `rs2 > end`.

3.1.5. Bounds Splitting

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

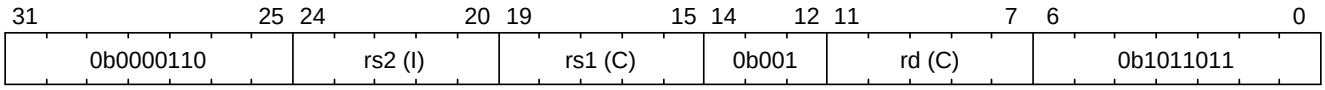

Figure 9. SPLIT instruction format

The instruction reads a capability from `rs1` and an integer from `rs2` and attempts to split the capability into two capabilities, one with bounds `[base, rs2)` and the other with bounds `[rs2, end)`, assuming the original bounds were `[base, end)`.

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

- `rs1` does not contain a capability.
- The `valid` field of the capability in `rs1` is `0` (invalid).
- `rs2` does not contain an integer.
- The `type` field of the capability in `rs1` is neither `0` nor `1` (neither linear nor non-linear).
- The original bounds of the capability in `rs1` are `[base, end)` and `rs2 <= base` or `rs2 >= end`.

If no exception is raised: The capability in `rs1` has its `end` field set to `rs2`. A new capability is created with `base = rs2` and the other fields equal to those of the original capability in `rs1`. The new capability is written to `rd`.

3.1.6. Permission Tightening

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

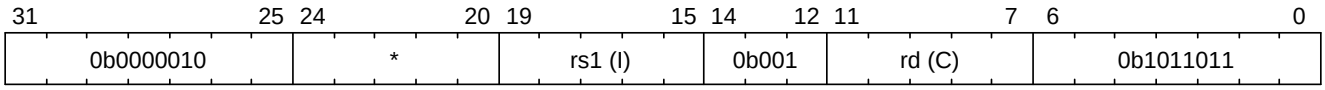

Figure 10. TIGHTEN instruction format

The instruction reads the new permissions from `rs1` and attempts to set the `perms` field of the capability in `rd` to `rs1`.

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

- `rd` does not contain a capability.
- The `valid` field of the capability in `rd` is `0` (invalid).
- The `type` field of the capability in `rd` is not `0`, `1`, or `3` (linear, non-linear, or uninitialised).
- `rs1` does not contain an integer.
- The content of `rs1` is outside the range of `perms`.
- The `perms` field of the capability in `rd` is `p` and `rs1` ⇐ `p` 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.

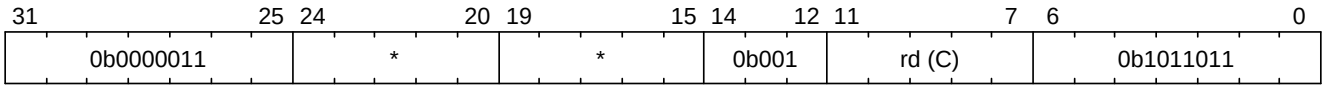

Figure 11. DELIN instruction format

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

- `rd` does not contain a capability.
- The `valid` field of the capability in `rd` is `0` (invalid).
- The `type` field of the capability in `rd` is not `0` (linear).

If no exception is raised: The `type` field of the capability in `rd` 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).

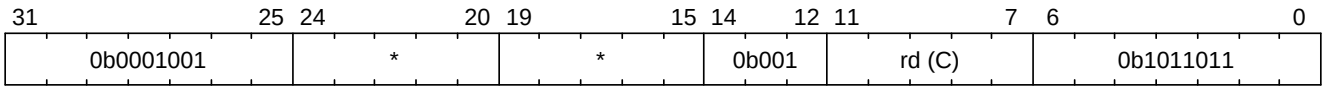

Figure 12. INIT instruction format

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

- `rd` does not contain a capability.
- The `valid` field of the capability in `rd` is `0` (invalid).
- The `type` field of the capability in `rd` is not `3` (uninitialised).
- The `end` field and the `cursor` field of the capability in `rd` are not equal.

If no exception is raised: The `type` field of the capability in `rd` is set to `0` (linear).

3.2.3. Sealing

The SEAL instruction seals a linear capability.

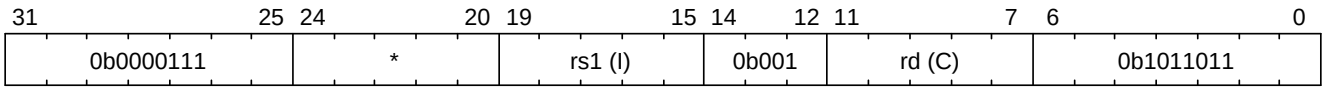

Figure 13. SEAL instruction format

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

- `rd` does not contain a capability.
- The `valid` field of the capability in `rd` is `0` (invalid).
- The `type` field of the capability in `rd` is not `0` (linear).
- The `perms` field of the capability in `rd` is not `3` (read-write) or `4` (read-write-execute).
- The size of the memory region associated with the capability in `rd` is smaller than `CLENBYTES * 32` bytes. That is, `end - base < CLENBYTES * 32`.
- `rs1` does not contain an integer.
- The integer contained in `rs1` is larger than `31`.

If no exception is raised: The `type` field of the capability in `rd` is set to `2` (sealed). The `count` field set to the integer contained in `rs1`.

3.3. Dropping

TODO: check whether dropping is actually necessary.

The DROP instruction invalidates a capability.

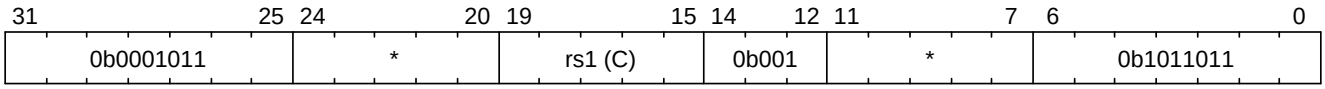

Figure 14. DROP instruction format

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

- `rs1` does not contain a capability.
- The `valid` field of the capability in `rs1` is `0` (invalid).

If no exception is raised: The `valid` field of the capability in `rs1` is set to `0` (invalid).

3.4. Revocation

3.4.1. Revocation Capability Creation

The MREV instruction creates a revocation capability.

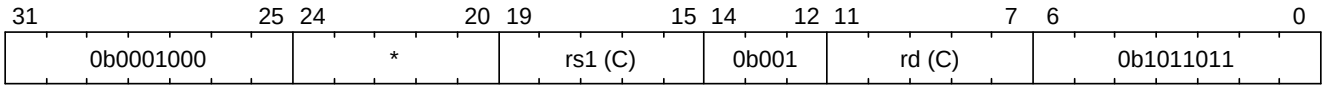

Figure 15. MREV instruction format

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

- `rs1` does not contain a capability.
- The `type` field of the capability in `rs1` is not `0` (linear).
- The `valid` field of the capability in `rs1` is `0` (invalid).

If no exception is raised: A new capability is created in `rd` with the same `base`, `end`, `perms` and `cursor` fields as the capability in `rs1`. The `type` field of the new capability is set to `2` (revocation).

3.4.2. Revocation Operation

The REVOKE instruction revokes a capability.

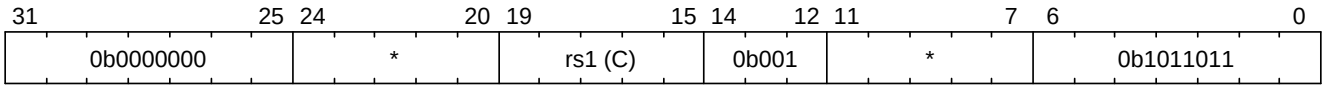

Figure 16. REVOKE instruction format

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

- `rs1` does not contain a capability.
- The `type` field of the capability in `rs1` is not `2` (revocation).
- The `valid` field of the capability in `rs1` is `0` (invalid).

If no exception is raised:

For all capabilities `c` in the system (in either a register or memory location), its `valid` field is set to `0` (invalid) if any of the following conditions are met:

- The type field of `c` is not `2` (revocation), the `valid` field of `c` is `1` (valid), and `c` aliases with `rs1`
- The type field of `c` is `2` (revocation), the `valid` field of `c` is `1` (valid), and `rs1 <t c`

The `type` field of the capability in `rs1` 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`)
- The `perms` field of `c` is not `3` (read-write) or `4` (read-write-execute)

Otherwise, the `type` field of the capability in `rs1` is set to `3` (uninitialised), and its `cursor` field is set to `base`.

4. Memory Access Instructions

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

4.1. Load/Store with Capabilities

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

Load:

The LDD, LDW, LDH, LDB instructions load an integer in the size of doubleword, word, halfword, and byte 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.

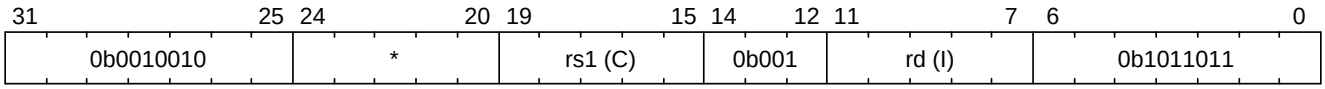

Figure 17. LDD instruction format

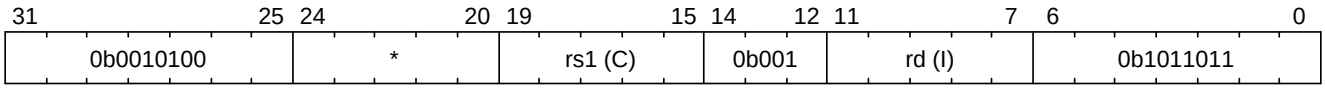

Figure 18. LDW instruction format

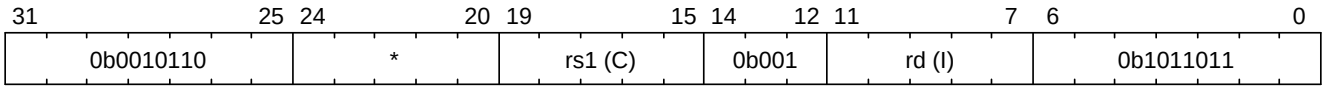

Figure 19. LDH instruction format

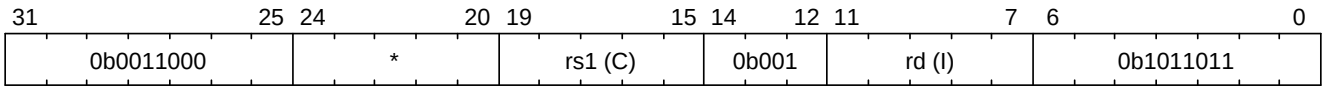

Figure 20. LDB instruction format

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

- `rs1` does not contain a capability.
- The `type` field of the capability in `rs1` is neither `0` (linear) nor `1` (non-linear).
- The `valid` field of the capability in `rs1` is `0` (invalid).
- The `perms` field of the capability in `rs1` is `0` (no access).
- The bound of the capability in `rs1` is `[base, end)`, where `base` and `end` are the `base` and `end` fields of the capability in `rs1`, and the `cursor` field of the capability in `rs1` is not in the range `[base, end-size]` (i.e., `cursor < base` or `cursor > end-size`), where `size` is the size (in bytes) of the integer being loaded.
- The `cursor` field of the capability in `rs1` is not aligned to the size of the integer being loaded.

If no exception is raised: Load the content at the memory location `[cursor, cursor + size)` as an integer, where `cursor` is the `cursor` field of the capability in `rs1` and `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 `rd`.

Store:

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

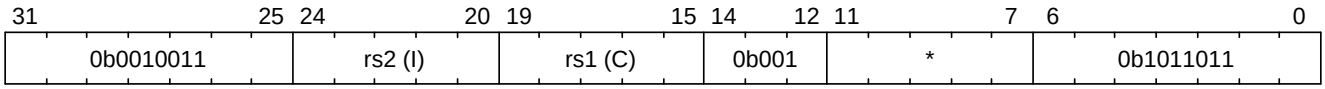

Figure 21. STD instruction format

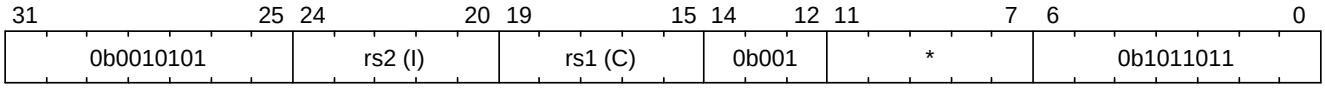

Figure 22. STW instruction format

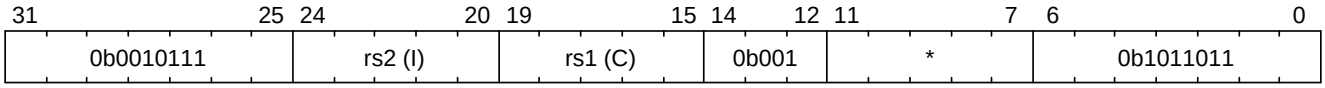

Figure 23. STH instruction format

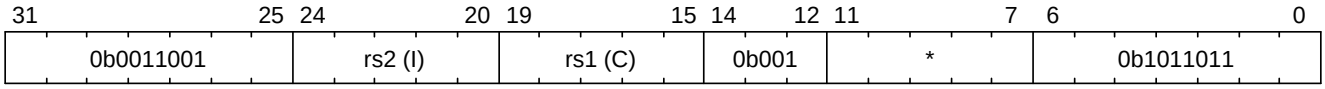

Figure 24. STB instruction format

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

- `rs1` does not contain a capability.
- The `type` field of the capability in `rs1` is not `0`, `1`, or `3` (linear, non-linear, or uninitialized).
- The `valid` field of the capability in `rs1` is `0` (invalid).
- The `perms` field of the capability in `rs1` is not `3` or `4` (read-write or read-write-execute).
- The bound of the capability in `rs1` is `[base, end)`, where `base` and `end` are the `base` and `end` fields of the capability in `rs1`, and the `cursor` field of the capability in `rs1` is not in the range `[base, end-size]` (i.e., `cursor < base` or `cursor > end-size`), where `size` is the size (in bytes) of the integer being loaded.
- The `cursor` field of the capability in `rs1` is not aligned to the size of the scalar value being loaded.
- `rs2` does not contain an integer.

If no exception is raised: Store the integer in `rs2` to the memory location `[cursor, cursor + size)`, where `cursor` is the `cursor` field of the capability in `rs1` and `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). The `cursor` field of the capability in `rs1` is set to `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.

LDC:

The LDC instruction loads a capability from memory.

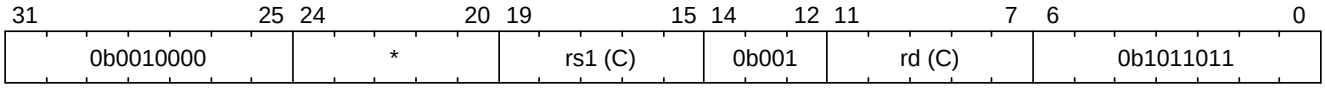

Figure 25. LDC instruction format

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

- `rs1` does not contain a capability.
- The `type` field of the capability in `rs1` is neither `0` (linear) nor `1` (non-linear).
- The `valid` field of the capability in `rs1` is `0` (invalid).
- The `perms` field of the capability in `rs1` is `0` (no access).
- The bound of the capability in `rs1` is `[base, end)`, where `base` and `end` are the `base` and `end` fields of the capability in `rs1`, and the `cursor` field of the capability in `rs1` is not in the range `[base, end-CLENBYTES]` (i.e., `cursor < base` or `cursor > end-CLENBYTES`).
- The `cursor` field of the capability in `rs1` is not aligned to `CLEN` bits.
- The data contained in the memory location `[cursor, cursor + CLENBYTES)`, where `cursor` is the `cursor` field of the capability in `rs1`, is not a capability.
- The capability being loaded is not a non-linear capability (i.e., `type != 1`), and the `perms` field of the capability in `rs1` is not `3` or `4` (read-write or read-write-execute).

If no exception is raised: Load the capability at the memory location `[cursor, cursor + CLENBYTES)`, where `cursor` is the `cursor` field of the capability in `rs1`, into `rd`. If the capability being loaded is a linear capability, the data contained in the memory location `[cursor, cursor + CLENBYTES)` will be set to the content of `cnull`.

STC:

The STC instruction stores a capability to memory.

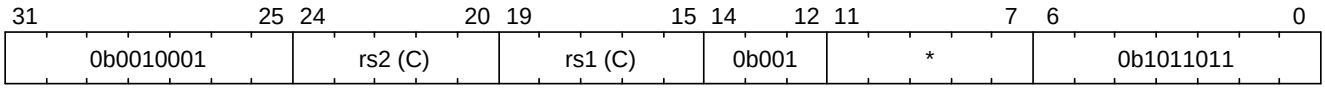

Figure 26. STD instruction format

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

- `rs1` does not contain a capability.
- The `type` field of the capability in `rs1` is not `0`, `1`, or `3` (linear, non-linear, or uninitialized).
- The `valid` field of the capability in `rs1` is `0` (invalid).

- The `perms` field of the capability in `rs1` is not `3` or `4` (read-write or read-write-execute).
- The bound of the capability in `rs1` is `[base, end)`, where `base` and `end` are the `base` and `end` fields of the capability in `rs1`, and the `cursor` field of the capability in `rs1` is not in the range `[base, end-CLENBYTES]` (i.e., `cursor < base` or `cursor > end-CLENBYTES`).
- The `cursor` field of the capability in `rs1` is not aligned to `CLEN` bits.
- `rs2` does not contain a capability.

If no exception is raised: Store the capability in `rs2` to the memory location `[cursor, cursor + CLENBYTES)`, where `cursor` is the `cursor` field of the capability in `rs1`. The `cursor` field of the capability in `rs1` is set to `cursor + CLENBYTES`.

5. Control Flow Instructions

5.1. Jump to Capabilities

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

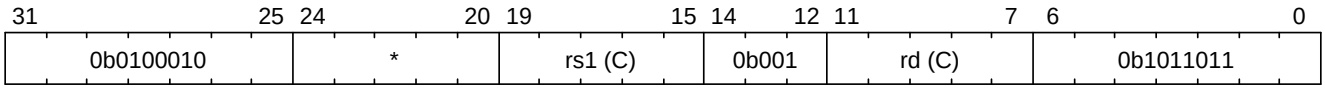

Figure 27. CJAL instruction format

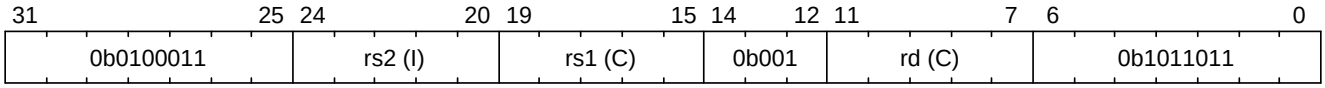

Figure 28. CBNZ instruction format

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

- `cwrld` is `0` (normal world).
- `rs1` does not contain a capability.
- The `type` field of the capability in `rs1` is neither `0` (linear) nor `1` (non-linear).
- The `perms` field of the capability in `rs1` is neither `2` (read-execute) nor `4` (read-write-execute).

If no exception is raised: For CJAL instruction, the program counter (`pc`) is set to the capability in `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`. If the capability is valid, and the `cursor` field of the capability in `pc` is in-bound (i.e., the bound of the capability in `pc` is `[base, end)`, where `base` and `end` are the `base` and `end` fields of the capability in `pc`, and `base <= cursor <= end-4`), then the next instruction will be fetched from its `cursor` field.

The behaviour of CBNZ depends on the content of `rs2`:

- If the content of `rs2` is zero (`0`), the behaviour is the same as for NOP.
- Otherwise, the behaviour is the same as for CJAL.

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.

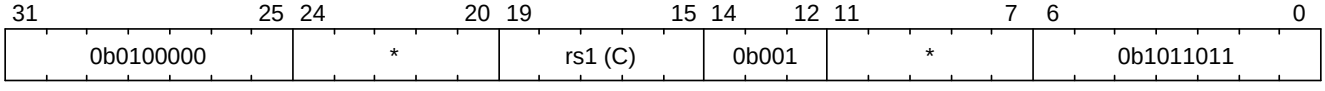

Figure 29. CALL instruction format

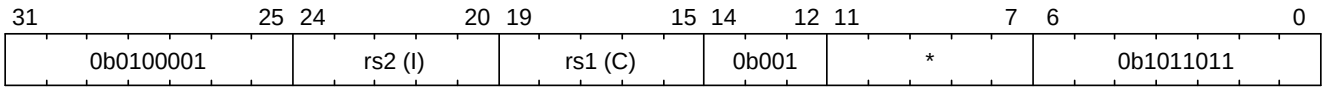

Figure 30. RETURN instruction format

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

- `cwrld` is `0` (normal world).
- `rs1` does not contain a capability.
- The `valid` field of the capability in `rs1` is `0` (invalid).
- The `type` field of the capability in `rs1` is not `4` (sealed).

For RETURN, an exception is raised also when any of the following conditions are met:

- `rs2` does not contain an integer

If no exception is raised:

For CALL:

1. Load the content at the memory location `[base, base + CLENBYTES)`, where `base` is the `base` field of the capability in `rs1`, to the program counter (`pc`).
2. For `i = 1, 2, ⋯, count`, load the content at the memory location `[base + i * CLENBYTES, base + (i + 1) * CLENBYTES)`, where `count` is the `count` field of the capability in `rs1`, to `x[i]` (the `i`-th general-purpose register).
3. Store the old `pc` value to the memory location `[base, base + CLENBYTES)`, and the old `sp` value to the memory location `[base + CLENBYTES, base + 2 * CLENBYTES)`.
4. Set the `type` field of the capability in `rs1` to `5` (sealed-return), and write the capability to the register `cra`.

For RETURN when the content of `rs1` is `0`:

1. Load the content at the memory location `[base, base + CLENBYTES)`, where `base` is the `base` field of the capability in `rs1`, to the program counter (`pc`).
2. For `i = 1, 2, ⋯, 31`, load the content at the memory location `[base + i * CLENBYTES, base + (i + 1) * CLENBYTES)`, to `x[i]` (the `i`-th general-purpose register).
3. Write the old `pc` value with the `cursor` field replaced with the content of `rs2` to the memory location `[base, base + CLENBYTES)`.
4. For `i = 1, 2, ⋯, count`, store the content of `x[i]` (the `i`-th general-purpose register) to the memory location `[base + i * CLENBYTES, base + (i + 1) * CLENBYTES)`, where `count` is the `count` field of the capability in `rs1`.
5. Set the `type` field of the capability in `rs1` to `4` (sealed), and write the capability to the exception handler register `ceh`.

For RETURN when the content of `rs1` is not `0`:

1. Load the content at the memory location `[base, base + CLENBYTES)`, where `base` is the `base` field

of the capability in `rs1`, to the program counter (`pc`).

2. Load the content at the memory location `[base + CLENBYTES, base + 2 * CLENBYTES)`, where `base` is the `base` field of the capability in `rs1`, to the stack pointer (`sp`).
3. Write the old `pc` value with the `cursor` field replaced with the content of `rs2` to the memory location `[base, base + CLENBYTES)`.
4. For `i = 1, 2, ⋯, count`, store the content of `x[i]` (the `i`-th general-purpose register) to the memory location `[base + i * CLENBYTES, base + (i + 1) * CLENBYTES)`, where `count` is the `count` field of the capability in `rs1`.
5. Set the `type` field of the capability in `rs1` to `4` (sealed), and write the capability to the register `x[reg]` where `reg` is the `reg` field of the capability in `rs1`.

5.3. World Switching

TransCapstone-RISC-V is an extended version of Capstone-RISC-V which adds a pair of extra instructions CAPENTER and CAPEXIT 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.

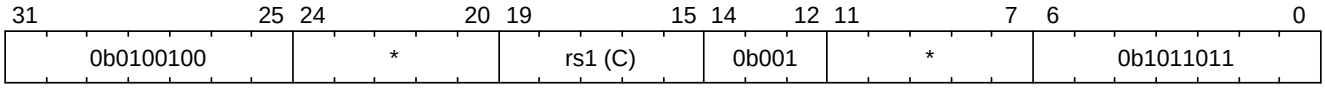

Figure 31. CAPENTER instruction format

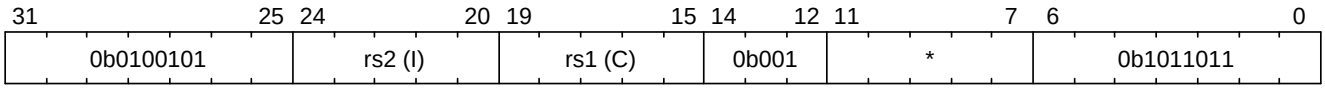

Figure 32. CAPEXIT instruction format

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 those instructions roughly correspond to the CALL and RETURN instructions respectively.

5.3.1. CAPENTER

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

- `cwrld` is `1` (secure world).
- `rs1` does not contain a capability.
- The `valid` field of the capability in `rs1` is `0` (invalid).
- The `type` field of the capability in `rs1` is not `4` (sealed).

If no exception is raised:

1. Load the content at the memory location `[base, base + CLENBYTES)`, where `base` is the `base` field of the capability in `rs1`, to the program counter (`pc`).

2. For `i = 1, 2, ⋯, count`, load the content at the memory location `[base + i * CLENBYTES, base + (i + 1) * CLENBYTES)`, where `count` is the `count` field of the capability in `rs1`, to `x[i]` (the `i`-th general-purpose register).
3. Store the old `pc` value to `normal_pc`, and the old `sp` value to `normal_sp`.
4. Set the `type` field of the capability in `rs1` to `5` (sealed-return), and write the capability to the register `switch_cap`.
5. Write `rs1` to the register `switch_reg`.
6. Create a capability of `type = 6` (exit) in `cra`.

5.3.2. CAPEXIT

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

- `cwrld` is `0` (normal world).
- `rs1` does not contain a capability.
- The `valid` field of the capability in `rs1` is `0` (invalid).
- The `type` field of the capability in `rs1` is not `6` (exit).
- `rs2` does not contain an integer.
- The `valid` field of the capability in `switch_cap` is `0` (invalid).

If no exception is raised:

1. Write the content of `normal_pc` and `normal_sp` to `pc` and `sp` respectively.
2. Write the old `pc` content with the `cursor` field replaced with the content of `rs2` to the memory location `[base, base + CLENBYTES)`, where `base` is the `base` field of the capability in `switch_cap`.
3. For `i = 1, 2, ⋯, count`, store the content of `x[i]` (the `i`-th general-purpose register) to the memory location `[base + i * CLENBYTES, base + (i + 1) * CLENBYTES)`, where `count` is the `count` field of the capability in `switch_cap`.
4. Set the `type` field of `switch_cap` to `4` (sealed) and write it to `x[switch_reg]`.

6. Adjustments to Existing Instructions

6.1. Control Flow Instructions

In RISC-V, 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.

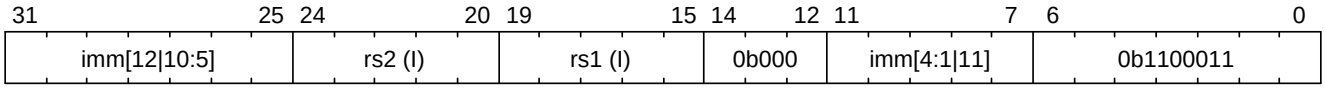

Figure 33. beq instruction format (B-type)

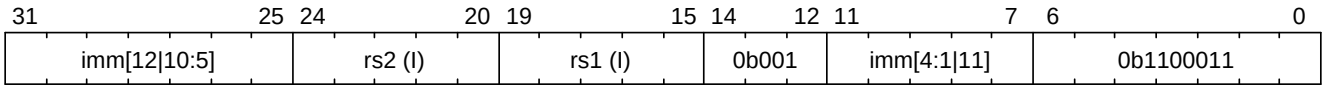

Figure 34. bne instruction format (B-type)

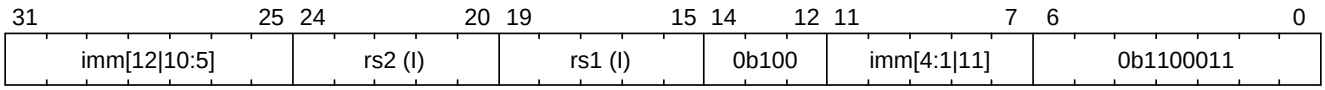

Figure 35. blt instruction format (B-type)

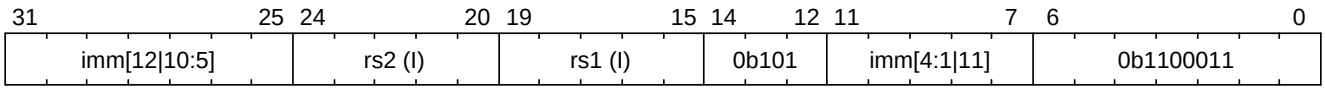

Figure 36. bge instruction format (B-type)

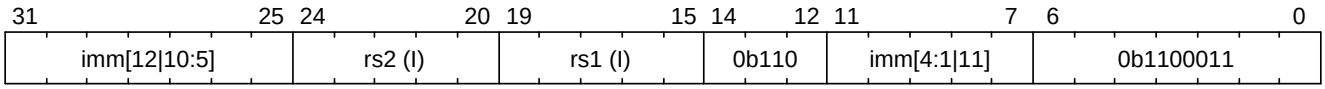

Figure 37. bltu instruction format (B-type)

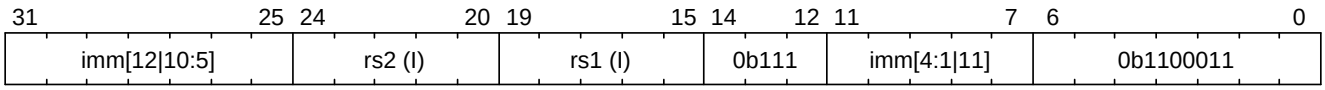

Figure 38. bgeu instruction format (B-type)

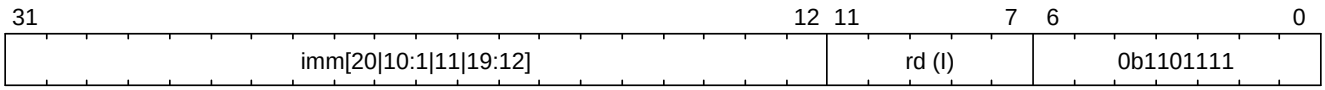

Figure 39. jal instruction format (J-type)

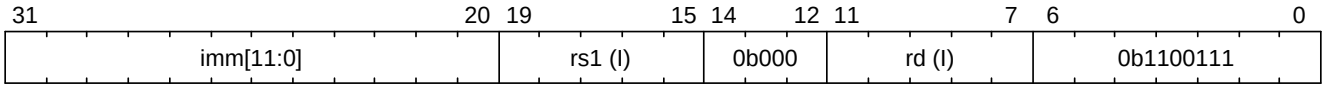

Figure 40. jalr instruction format (I-type)

Adjustments when `cwld` is `0` (normal world):

- An exception is raised if any of its operands is not an integer.

Adjustments when `cwld` is `1` (secure world):

- An exception is raised if any of its operands is not an integer.
- The `cursor` field of the capability in `pc`, instead of `pc` itself, is changed by the instruction.
- If the instruction is `jal` or `jalr`, the `cursor` field of the capability in `pc` (for the next instruction), instead of `pc` itself, is written to `rd`.

7. Interrupts and Exceptions

TODO: add support for nesting

TODO: Note that it is safe to delegate exception handling but not safe to delegate interrupt handling. Hence we need to distinguish between exceptions and interrupts

When an interrupt or exception occurs, the behaviour of a processor core depends on which world it is currently executing in (i.e., the content of the `cwrld` register). If the core is executing in the normal world, the behaviour is entirely inherited from RISC-V. Otherwise, if the core is executing in the secure world,

If the content in `ceh` is a valid sealed capability and the content in `switch_cap` is a valid sealed-return capability:

1. Store the current value of the program counter (`pc`) to the memory location `[switch_cap.base, switch_cap.base + CLENBYTES)`.
2. For `i = 1, 2, ⋯, 31`, store the content of the `i`-th general purpose to the memory location `[switch_cap.base + i * CLENBYTES, switch_cap.base + (i + 1) * CLENBYTES)`.
3. Set the `reg` field of `switch_cap` to `0` (asynchronous),
4. Store the content of `switch_cap` to the memory location `[ceh.base + CLENBYTES, ceh.base + 2 * CLENBYTES)`.
5. Write the content of `ceh` to the register `x[reg]` where `reg` is the original content of the `reg` field of `switch_cap`.
6. Load the program counter `pc` and the stack pointer `sp` from `normal_pc` and `normal_sp` respectively.
7. Scrub the other general-purpose registers.
8. Set the `cwrld` register to `0` (normal world).
9. Trigger exception handling in the normal world.

Otherwise:

1. Write `cnull` to the register `x[reg]` where `reg` is the original content of the `reg` field of `switch_cap`.
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 the exception handling in the 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.

8. Memory Consistency Model

TODO

Appendix A: Debugging Instructions (Non-Normative)

TODO

Appendix B: Instruction Listing

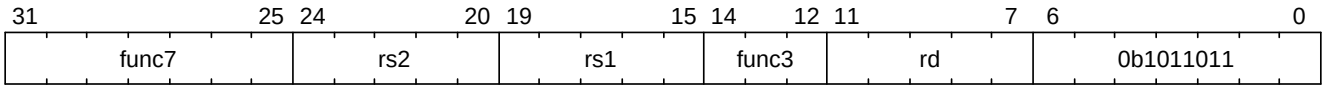

Figure 41. Instruction format: R-type

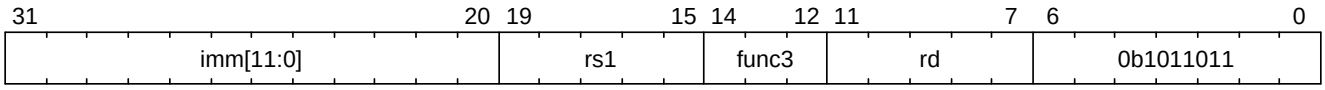

Figure 42. Instruction format: I-type

Table 2. Debugging instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]	World
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	-	*

Table 3. Capability manipulation instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]	World
REVOKE	R	001	0000000	C	-	-	-	*
SHRINK	R	001	0000001	I	I	C	-	*
TIGHTEN	R	001	0000010	I	-	C	-	*
DELIN	R	001	0000011	-	-	C	-	*
LCC	R	001	0000100	C	-	I	-	*
SCC	R	001	0000101	I	-	C	-	*
SPLIT	R	001	0000110	C	I	C	-	*
SEAL	R	001	0000111	I	-	C	-	*
MREV	R	001	0001000	C	-	C	-	*
INIT	R	001	0001001	-	-	C	-	*

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]	World
MOVC	R	001	0001010	C	-	C	-	*
DROP	R	001	0001011	C	-	-	-	*
CAPGET	R	001	0001100	-	-	C	-	N
CINCOFFSET	R	001	0001101	C	I	C	-	*
CINCOFFSETIMM	I	011	-	C	-	C	S	*

Table 4. Memory access instructions

Mnemonic	Format	Func3	Func7	rs1	rs2	rd	imm[11:0]	World
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	-	-	*

Table 5. Control flow instructions

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

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

Appendix C: Assembly Code Examples

TODO

Appendix D: Abstract Binary Interface (Non-Normative)

TODO