The Capstone-RISC-V Instruction Set Reference

Table of Contents

1. Introduction 5
- 1.1. Goals 5
- 1.2. Major Design Elements 5
- 1.3. Capstone-RISC-V ISA Overview 6
- 1.4. Assembly Mnemonics 6
- 1.5. Notations 6
- 1.6. Bibliography 7

2. Programming Model 8
- 2.1. Capabilities 8
- 2.2. Variants 10
- 2.3. Extension to General-Purpose Registers 10
- 2.4. Extension to Other Registers 11
- 2.5. Added Registers 12
- 2.6. Extension to Memory 13
- 2.7. Instruction Set 13

3. Capability Manipulation Instructions 14
- 3.1. Cursor, Bounds, and Permissions Manipulation 14
- 3.2. Type Manipulation 16
- 3.3. Dropping 18
- 3.4. Revocation 18

4. Memory Access Instructions 20
- 4.1. Load/Store with Capabilities 20
- 4.2. Load/Store Capabilities 21
- 4.3. *TransCapstone* Added Instructions 23

5. Control Flow Instructions 25
- 5.1. Jump to Capabilities 25
- 5.2. Domain Crossing 25
- 5.3. A World Switching Extension for *TransCapstone* 28

6. Control State Instructions 31
- 6.1. Capability CSR (CCSR) Manipulation 31

7. Adjustments to Existing Instructions 32
- 7.1. Control Flow Instructions 32
- 7.2. Memory Access Instructions 33

8. Interrupts and Exceptions 35

8.1. Exception and Exit Codes . . . 35
8.2. Pure Capstone . . . 36
8.3. TransCapstone . . . 37
9. Memory Consistency Model . . . 40
Appendix A: Debugging Instructions (Non-Normative) . . . 41
A.1. World Switching . . . 41
A.2. Exception Handling . . . 41
Appendix B: Instruction Listing . . . 43
Appendix C: Assembly Code Examples . . . 46
Appendix D: Abstract Binary Interface (Non-Normative) . . . 47

Contributors to this document include (in alphabetical order): Mingkai Li, 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 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 RV64G.

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
`async`	`0..1`	Only applicable when `type = 4` (sealed) or `type = 5` (sealed-return). Whether the capability is sealed asynchronously: `0` = synchronously, `1` = asynchronously
`reg`	`0..31`	Only applicable when `type = 5` (sealed-return). The index of the general-purpose register to restore the capability to

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

```
<=p = { (0, 0), (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 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).

2.2. Variants

Capstone currently supports two variants of the ISA, i.e., *Pure Capstone* and *TransCapstone*. While *Pure Capstone* is a pure capability-based ISA, *TransCapstone* is a hybrid ISA that supports both capabilities and traditional virtual memory. In *TransCapstone*, the memory is divided into two parts, i.e., the secure memory and the untrusted memory. The range of the secure memory is defined as `[SBASE, SEND)`, where `SBASE` and `SEND` are required to be aligned to `CLEN` bits. These two variants share most of the parts of the ISA, and separate descriptions are provided for the parts that are different.

2.3. 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`

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

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

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

2.4. Extension to Other Registers

2.4.1. 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:

- `x[pc].valid` is `0` (invalid).
- `x[pc].type` is neither `0` (linear) nor `1` (non-linear).
- `x[pc].cursor` is not aligned to `4`.
- `x[pc].perms` is not executable (i.e., `2 <=p x[pc].perms` does not hold).
- `x[pc].cursor` is not in the range `[pc.base, pc.end-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`).

2.5. Added Registers

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

Table 2. Additional Registers in Capstone-RISC-V ISA

Capstone Variant	**Additional Registers**		
Pure Capstone	**Mnemonic**	**CCSR encoding**	**Description**
	`ceh`	`0x000`	The sealed capability for the exception handler
	`cih`	`0x001`	The sealed capability for the interrupt handler
TransCapstone	**Mnemonic**	**CCSR encoding**	**Description**
	`ceh`	`0x000`	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
	`exit_reg`	-	The index of the general-purpose register for receiving the exit code when exiting the secure world

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 state registers (CCSRs)*. Under their respective constraints, CCSRs can be manipulated using CCSR manipulation instructions.

The manipulation constraints for each CCSR are indicated below.

Table 3. Manipulation Constraints for CCSRs

Mnemonic	Read	Write
`ceh`	No constraint	No constraint
`cih`	The original content must be an invalid capability (`valid = 0`)	The original content must be an invalid capability (`valid = 0`)

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.

2.6. 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.

2.7. 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*).

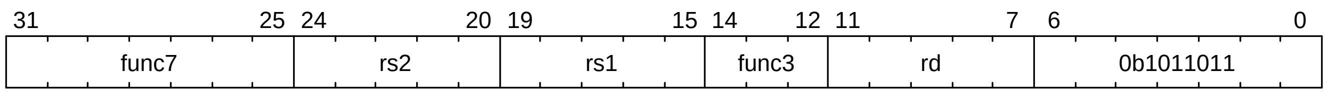

Figure 1. R-type instruction format

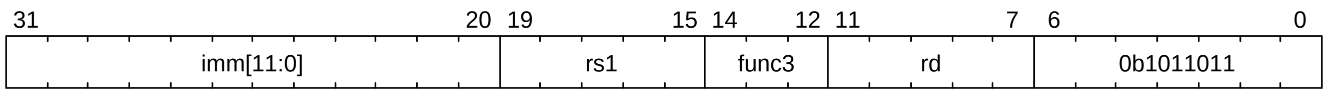

Figure 2. I-type instruction format

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.

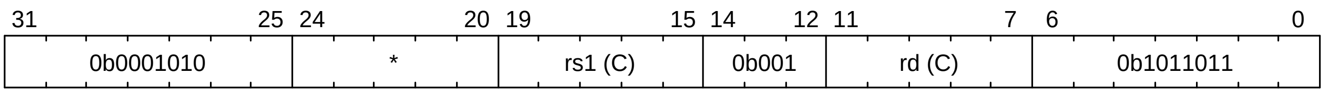

Figure 3. MOVC instruction format

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

- `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).

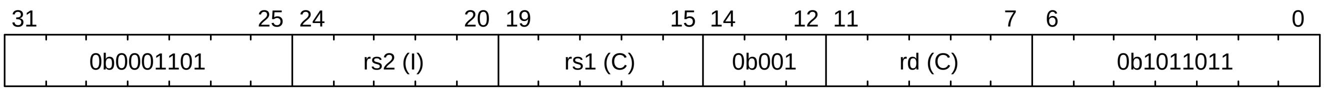

Figure 4. CINCOFFSET instruction format

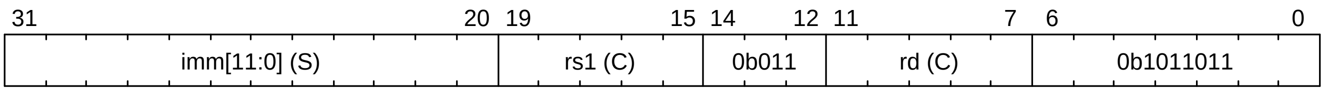

Figure 5. CINCOFFSETIMM instruction format

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

- `x[rs1]` is not a capability.
- `x[rs2]` is not an integer (for CINCOFFSET).
- `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 equal to an atomic execution of `MOVC rd, rs1` followed by an increment of `x[rd].cursor` 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.

31 25	24 20	19 15	14 12	11 7	6 0
0b0000101	*	rs1 (I)	0b001	rd (C)	0b1011011

Figure 6. SCC instruction format

31 25	24 20	19 15	14 12	11 7	6 0
0b0000100	*	rs1 (C)	0b001	rd (I)	0b1011011

Figure 7. LCC instruction format

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

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

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

31 25	24 20	19 15	14 12	11 7	6 0
0b0000001	rs2 (I)	rs1 (I)	0b001	rd (C)	0b1011011

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:

- `x[rd]` is not a capability.
- `x[rd].valid` is `0` (invalid).
- `x[rd].type` is not `0`, `1`, or `3` (linear, non-linear, or uninitialised).
- `x[rs1]` is not an integer.
- `x[rs2]` is not an integer.
- `x[rs1] >= x[rs2]`.
- `x[rs1] < x[rs2].base` or `x[rs2] > x[rs1].end`.

3.1.5. Bounds Splitting

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

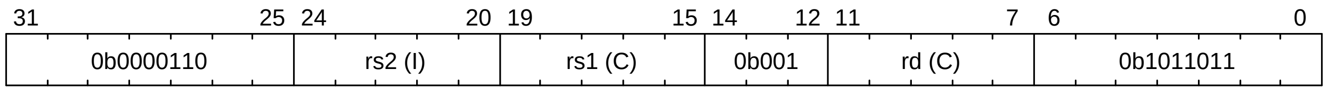

Figure 9. SPLIT instruction format

The instruction 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:

- `x[rs1]` is not a capability.
- `x[rs1].valid` is `0` (invalid).
- `x[rs1].type` is neither `0` nor `1` (neither linear nor non-linear).
- `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]`.

3.1.6. Permission Tightening

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

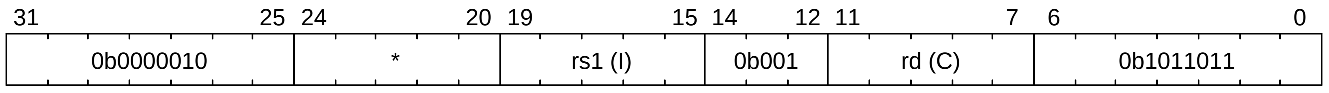

Figure 10. TIGHTEN instruction format

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

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

- `x[rd]` is not a capability.
- `x[rd].valid` is `0` (invalid).
- `x[rd].type` is not `0`, `1`, or `3` (linear, non-linear, or uninitialised).
- `x[rs1]` is not an integer.
- `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.

31 25	24 20	19 15	14 12	11 7	6 0
0b0000011	*	*	0b001	rd (C)	0b1011011

Figure 11. DELIN instruction format

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

- `x[rd]` is not a capability.
- `x[rd].valid` is `0` (invalid).
- `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).

31 25	24 20	19 15	14 12	11 7	6 0
0b0001001	*	*	0b001	rd (C)	0b1011011

Figure 12. INIT instruction format

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

- `x[rd]` is not a capability.
- `x[rd].valid` is `0` (invalid).
- `x[rd].type` is not `3` (uninitialised).
- `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.

31 25	24 20	19 15	14 12	11 7	6 0
0b0000111	*	*	0b001	rd (C)	0b1011011

Figure 13. SEAL instruction format

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

- `x[rd]` is not a capability.
- `x[rd].valid` is `0` (invalid).
- `x[rd].type` is not `0` (linear).

- `x[rd].perms` is not `3` (read-write) or `4` (read-write-execute).
- The size of the memory region associated with `x[rd]` is smaller than `CLENBYTES * 33` bytes. That is, `x[rd].end - x[rd].base < CLENBYTES * 33`.

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.

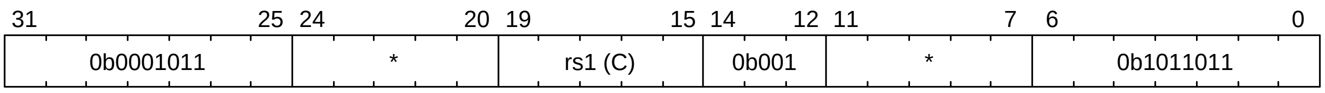

Figure 14. DROP instruction format

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

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

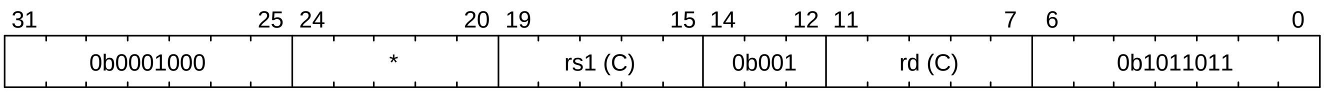

Figure 15. MREV instruction format

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

- `x[rs1]` is not a capability.
- `x[rs1].type` is not `0` (linear).
- `x[rs1].valid` is `0` (invalid).

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.

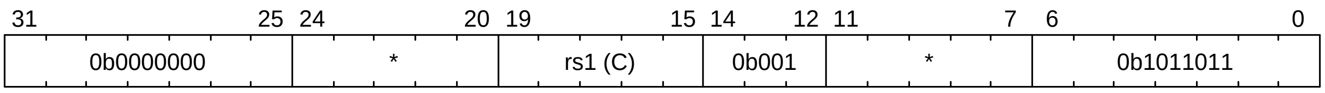

Figure 16. REVOKE instruction format

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

- `x[rs1]` is not a capability.
- `x[rs1].type` is not `2` (revocation).
- `x[rs1].valid` is `0` (invalid).

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`).
- `c.perms` is not `3` (read-write) or `4` (read-write-execute).

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.

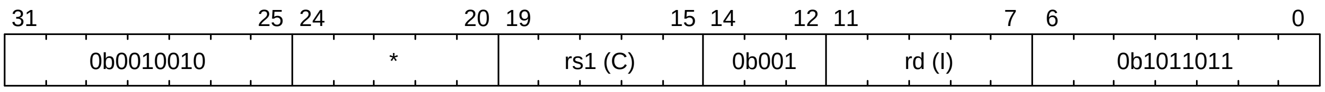

Figure 17. LDD instruction format

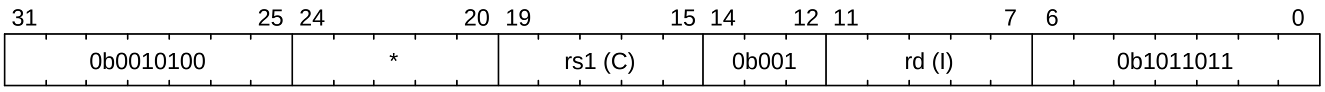

Figure 18. LDW instruction format

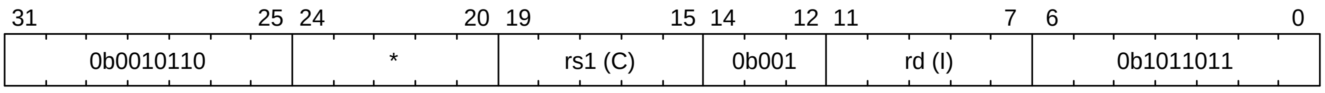

Figure 19. LDH instruction format

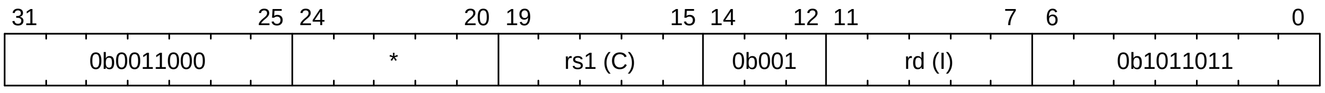

Figure 20. LDB instruction format

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

- `x[rs1]` is not a capability.
- `x[rs1].type` is neither `0` (linear) nor `1` (non-linear).
- `x[rs1].valid` is `0` (invalid).
- `x[rs1].perms` is `0` (no access).
- `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].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.

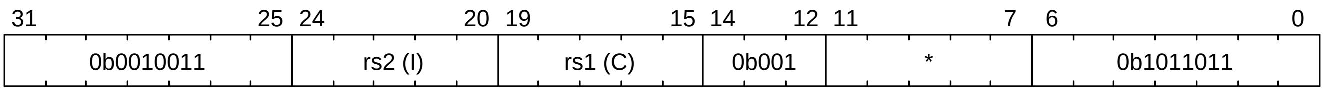

Figure 21. STD instruction format

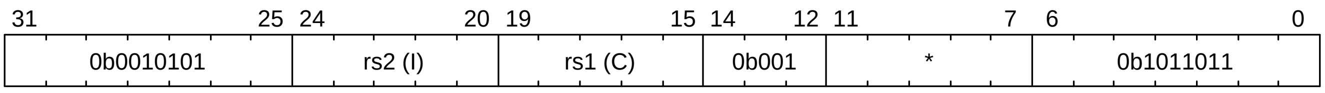

Figure 22. STW instruction format

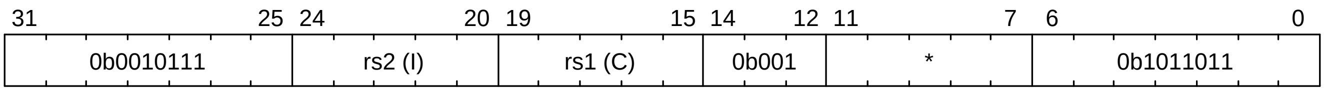

Figure 23. STH instruction format

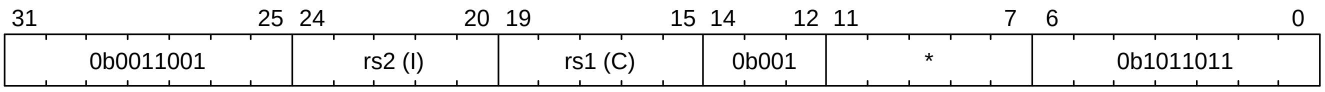

Figure 24. STB instruction format

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

- `x[rs1]` is not a capability.
- `x[rs1].type` is not `0`, `1`, or `3` (linear, non-linear, or uninitialized).
- `x[rs1].valid` is `0` (invalid).
- `x[rs1].perms` is not `3` or `4` (read-write or read-write-execute).
- `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].cursor` is not aligned to the size of the scalar value being loaded.
- `x[rs2]` is not an integer.

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.

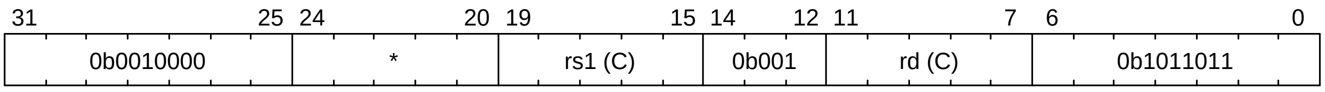

Figure 25. LDC instruction format

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

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

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.

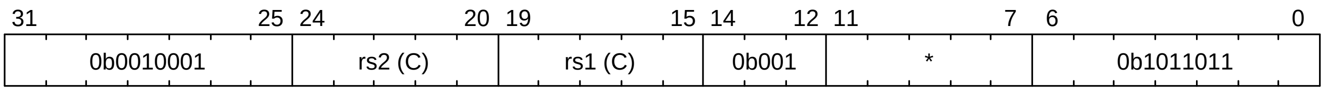

Figure 26. STC instruction format

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

- `x[rs1]` is not a capability.
- `x[rs1].type` is not `0`, `1`, or `3` (linear, non-linear, or uninitialized).
- `x[rs1].valid` is `0` (invalid).
- `x[rs1].perms` is not `3` or `4` (read-write or read-write-execute).
- `x[rs1].cursor` is not in the range `[x[rs1].base, x[rs1].end-CLENBYTES]`.
- `x[rs1].cursor` is not aligned to `CLEN` bits.
- `x[rs2]` is not a capability.

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.

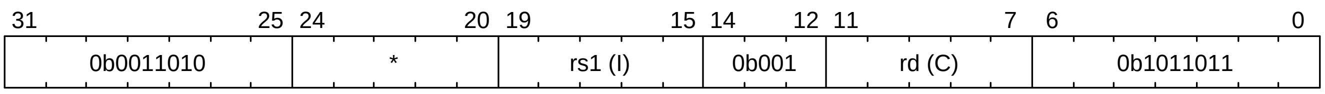

Figure 27. LDCR instruction format

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

- `x[rs1]` is not an integer.
- `x[rs1]` is not aligned to `CLEN` bits.
- `x[rs1]` is in the range `[SBASE, SEND)`.
- 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`), 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.

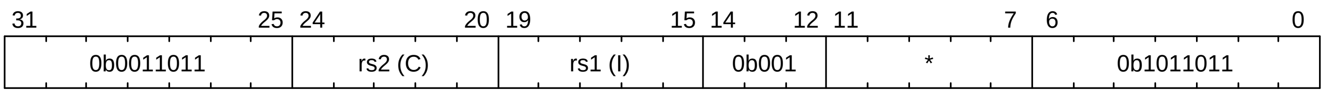

Figure 28. STCR instruction format

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

- `x[rs1]` is not an integer.
- `x[rs1]` is not aligned to `CLEN` bits.
- `x[rs1]` is in the range `[SBASE, SEND)`.
- `x[rs2]` is not a capability.

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`), `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.

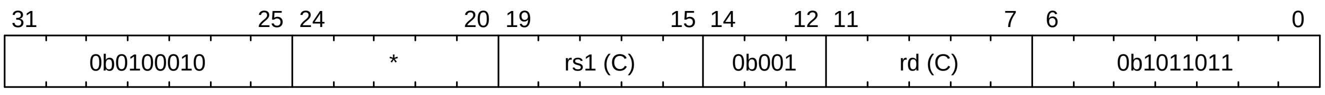

Figure 29. CJALR instruction format

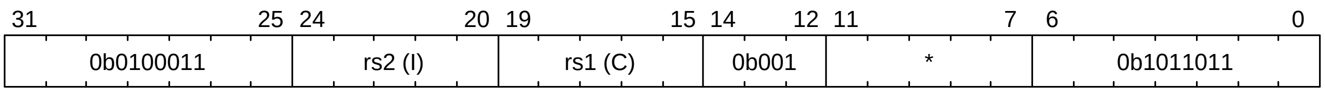

Figure 30. CBNZ instruction format

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

- *Pure Capstone*
  - `x[rs1]` is not a capability.
  - `x[rs1].type` is neither `0` (linear) nor `1` (non-linear).
  - `x[rs1].perms` is neither `2` (read-execute) nor `4` (read-write-execute).
- *TransCapstone*
  - `cwrld` is `0` (normal world).
  - Any of the conditions for *Pure Capstone* are met.

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*.

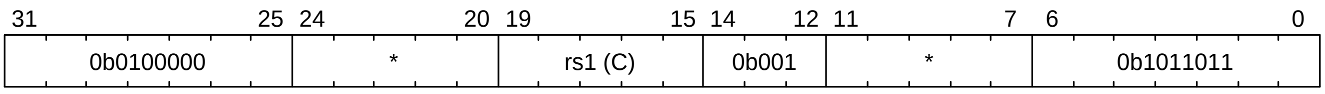

Figure 31. CALL instruction format

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

- *Pure Capstone*
  - `x[rs1]` is not a capability.
  - `x[rs1].valid` is `0` (invalid).
  - `x[rs1].type` is not `4` (sealed).
  - `x[rs1].async` is `1` (asynchronous).
- *TransCapstone*
  - `cwrld` is `0` (normal world).
  - Any of the conditions for *Pure Capstone* are met.

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].reg` to `rd`, set `x[rs1].async` to `0` (synchronous), and write the resulting `x[rs1]` to the register `cra`.

5.2.2. RETURN

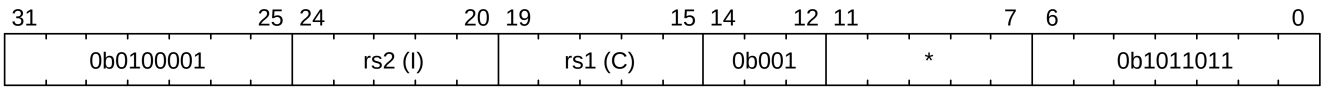

Figure 32. RETURN instruction format

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

- *Pure Capstone*
  - `x[rs1]` is not a capability.
  - `x[rs1].valid` is `0` (invalid).
  - `x[rs1].type` is not `5` (sealed-return).
  - `x[rs2]` is not an integer.

- *TransCapstone*
    - `cwrld` is `0` (normal world).
    - Any of the conditions for *Pure Capstone* are met.

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`, `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` (asynchronous):

1. Load the content at the memory location `[x[rs1].base, x[rs1].base + CLENBYTES)` to the program counter (`pc`).
2. For `i = 1, 2, ⋯, 31`, load the content at the memory location `[x[rs1].base + i * CLENBYTES, x[rs1].base + (i + 1) * CLENBYTES)`, to `x[i]` (the `i`-th general-purpose register).
3. 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)`.
4. For `i = 1, 2, ⋯, 31`, store the content of `x[i]` (the `i`-th general-purpose register) to the memory location `[x[rs1].base + i * CLENBYTES, x[rs1].base + (i + 1) * CLENBYTES)`. When `i = rs1`, store the content of `cnull` instead to `[x[rs1].base + i * CLENBYTES, x[rs1].base + (i + 1) * CLENBYTES)`.
5. Set the `x[rs1].type` to `4` (sealed), and write the resulting `x[rs1]` to the register `ceh`.

Note

When the `async` field of a sealed-return capability is `1` (asynchronous), some memory accesses are granted by this capability. The following table shows the memory accesses granted by sealed and sealed-return capabilities in different scenarios.

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

Capability type	`async`	**Read**	**Write**	**Execute**
Sealed	`0`	No	No	No
Sealed	`1`	No	No	No

Capability type	`async`	**Read**	**Write**	**Execute**
Sealed-return	`0`	No	No	No
Sealed-return	`1`	`cursor in [base, end)`	`cursor in [base, end)`	No

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

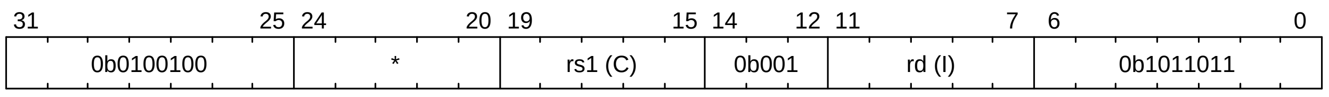

Figure 33. CAPENTER instruction format

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

- `cwrld` is `1` (secure world).
- `x[rs1]` is not a capability.
- `x[rs1].valid` is `0` (invalid).
- `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 `5` (sealed-return), `x[rs1].async` to `0` (synchronous), and write the resulting `x[rs1]` to `switch_cap`.

6. Write `rs1` to `switch_reg`.
7. Create a capability of `type = 6` (exit) in `cra`.
8. Write `rd` to `exit_reg`.
9. Set `cwrld` to `1` (secure world).

When `x[rs1].async = 1` (asynchronous):

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

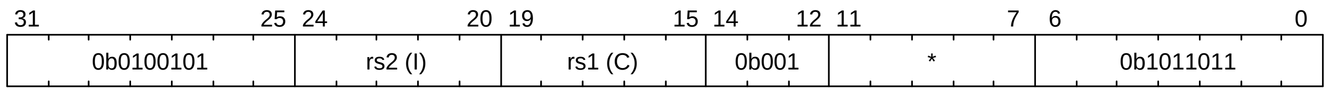

Figure 34. CAPEXIT instruction format

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

- `cwrld` is `0` (normal world).
- `x[rs1]` is not a capability.
- `x[rs1].valid` is `0` (invalid).
- `x[rs1].type` is not `6` (exit).
- `x[rs2]` is not an integer.
- `switch_cap` is not a capability.
- `switch_cap.valid` is `0` (invalid).
- `switch_cap.type` is not `4` (sealed-return).

- `switch_cap.async` is `1` (asynchronous).

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 `[switch_cap.base, switch_cap.base + CLENBYTES)`.
3. Write the former value of `ceh` and `csp` to the memory location `[switch_cap.base + CLENBYTES, switch_cap.base + 2 * CLENBYTES)` and `[switch_cap.base + 2 * CLENBYTES, switch_cap.base + 3 * CLENBYTES)` respectively.
4. Set `switch_cap.type` to `4` (sealed), `switch_cap.async` to `0` (synchronous), and write the resulting `switch_cap` to `x[switch_reg]`.
5. Set `x[exit_reg]` to `0` (normal exit).
6. Set `cwrld` to `0` (normal world).

6. Control State Instructions

6.1. Capability CSR (CCSR) Manipulation

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

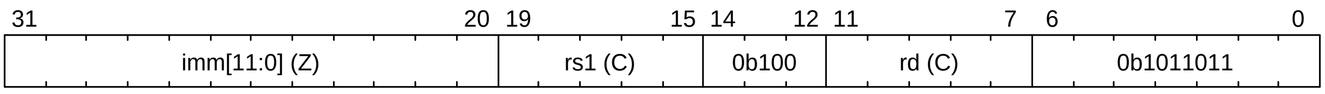

Figure 35. CCSRRW instruction format

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

- The immediate value `imm` does not correspond to the encoding of a valid capability CSR.
- `x[rs1]` is not a capability.

If no exception is raised:

1. If the read constraint is satisfied, the content of the capability CSR specified by the immediate value `imm` is written to the register `rd`. Otherwise, the content of the register `rd` is set to the content of `cnull`.
2. If the write constraint is satisfied, `x[rs1]` is written to the capability CSR specified by the immediate value `imm`. If the former `x[rs1]` is not a non-linear capability (i.e., `type != 1`), it will be set to the content of `cnull`. Otherwise, the original content of the capability CSR is preserved.

7. Adjustments to Existing Instructions

For most existing instructions in the RISC-V ISA, the adjustments are straightforward. Their behaviour is unchanged, and an 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 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.

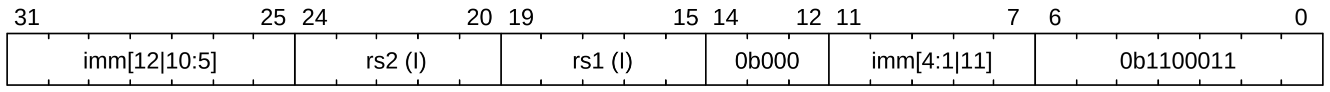

Figure 36. beq instruction format (B-type)

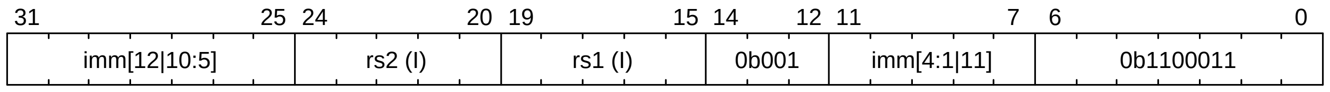

Figure 37. bne instruction format (B-type)

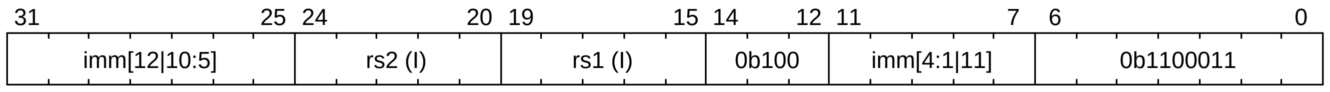

Figure 38. blt instruction format (B-type)

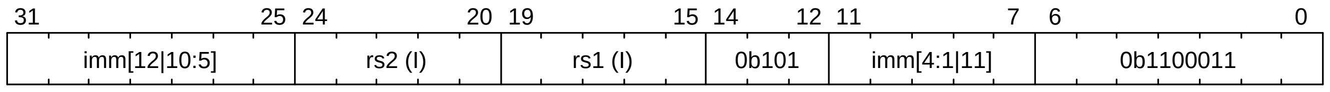

Figure 39. bge instruction format (B-type)

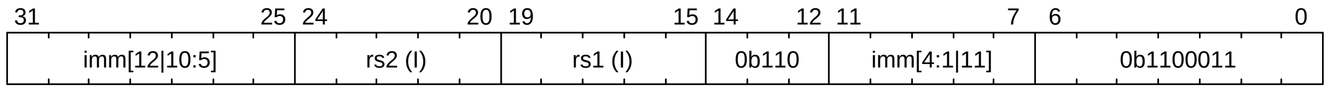

Figure 40. bltu instruction format (B-type)

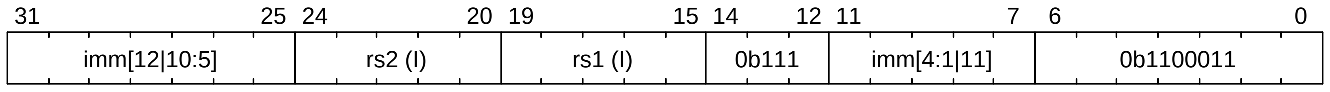

Figure 41. bgeu instruction format (B-type)

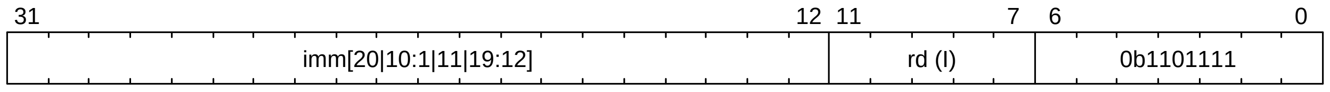

Figure 42. jal instruction format (J-type)

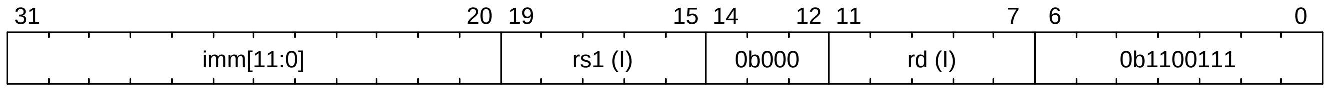

Figure 43. jalr instruction format (I-type)

The following adjustments are made to these instructions:

- *Pure Capstone*
  - An exception is raised if `x[rs1]`, `x[rs2]` or `x[rd]` is a capability.
  - `x[pc].cursor`, instead of `pc` itself, is changed by the instruction.
  - If the instruction is `jal` or `jalr`, `x[pc].cursor`, which contains the address of the next instruction, is written to `x[rd]`.
- *TransCapstone*
  - An exception is raised if `x[rs1]`, `x[rs2]` or `x[rd]` contains a capability.
  - If `cwld` is `1` (secure world), `x[pc].cursor`, instead of `pc` itself, is changed by the instruction.
  - If `cwld` 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 RISC-V, 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.

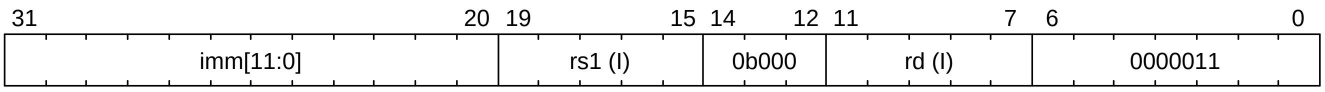

Figure 44. lb instruction format (I-type)

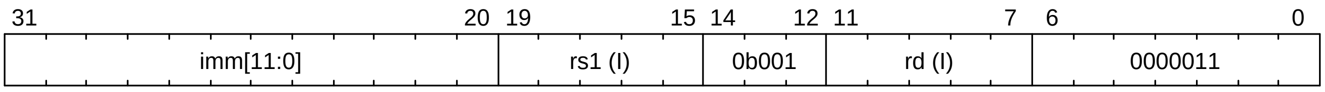

Figure 45. lh instruction format (I-type)

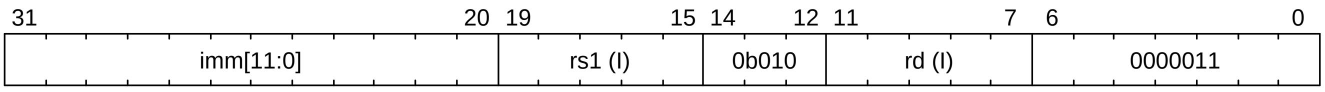

Figure 46. lw instruction format (I-type)

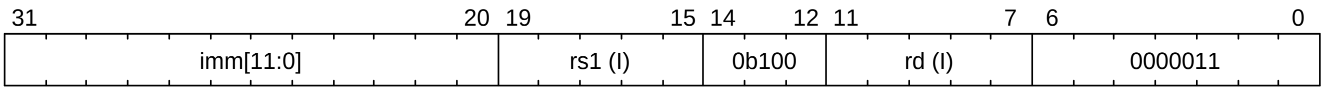

Figure 47. lbu instruction format (I-type)

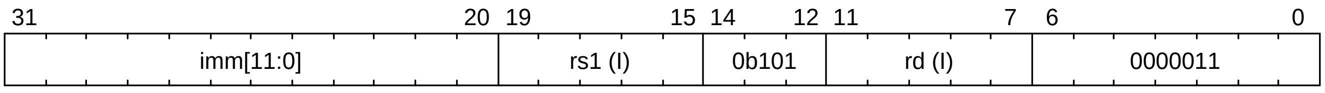

Figure 48. lhu instruction format (I-type)

31–20	19–15	14–12	11–7	6–0
imm[11:0]	rs1 (I)	0b110	rd (I)	0000011

Figure 49. lwu instruction format (I-type)

31–25	24–20	19–15	14–12	11–7	6–0
imm[11:5]	rs2 (I)	rs1 (I)	0b000	imm[4:0]	0100011

Figure 50. sb instruction format (S-type)

31–25	24–20	19–15	14–12	11–7	6–0
imm[11:5]	rs2 (I)	rs1 (I)	0b001	imm[4:0]	0100011

Figure 51. sh instruction format (S-type)

31–25	24–20	19–15	14–12	11–7	6–0
imm[11:5]	rs2 (I)	rs1 (I)	0b010	imm[4:0]	0100011

Figure 52. sw instruction format (S-type)

31–25	24–20	19–15	14–12	11–7	6–0
imm[11:5]	rs2 (I)	rs1 (I)	0b011	imm[4:0]	0100011

Figure 53. sd instruction format (S-type)

The following adjustments are made to these instructions:

- *Pure Capstone*
  - An exception is raised if any of these instructions is executed.
- *TransCapstone*
  - An exception is raised if any of these instructions is executed when `cwld` is `1` (secure world).
  - An exception is raised if `x[rs1]`, `x[rs2]` or `x[rd]` contains a capability.
  - An 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 RV64G, where applicable, for ease of implementation and interoperability.

Table 5. 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	8	1
Invalid capability	9	1

Note

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

8.2. 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 "panics" means here

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

8.2.1. Handling of Interrupts

TODO: need to specify how to record cause of the interrupt

TODO: interrupt masking

TODO: record the pending 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. For `i = 1, 2, ⋯, 31`, load the content of `x[i]` from memory location `[cih.base + i * CLENBYTES, cih.base + (i + 1) * CLENBYTES)`.
3. Store the original program counter `pc` to the memory location `[cih.base + CLENBYTES, cih.base + 2 * CLENBYTES)`.
4. For `i = 1, 2, ⋯, 31`, store the *original* content of `x[i]` to memory location `[cih.base + i * CLENBYTES, cih.base + (i + 1) * CLENBYTES)`.
5. Set `cih.type` to `5` (sealed-return), `cih.reg` to `0` (asynchronous), and `cih.async` to `1` (asynchronous).

6. Write `cih` to the register `c1`.
7. Write the exception code to the register `x10`.

8.2.2. Handling of Exceptions

The CPU core panics 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 capability (`type != 4`).

Otherwise:

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

8.3. TransCapstone

TransCapstone retains the same interrupt and exception handling mechanims for the normal world as in RV64G.

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 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.

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

8.3.2. 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. For `i = 1, 2, ⋯, 31`, load the content of `x[i]` from memory location `[ceh.base + i * CLENBYTES, ceh.base + (i + 1) * CLENBYTES)`.
3. Store the original program counter `pc` to the memory location `[ceh.base + CLENBYTES, ceh.base + 2 * CLENBYTES)`.
4. For `i = 1, 2, ⋯, 31`, store the *original* content of `x[i]` to memory location `[ceh.base + i * CLENBYTES, ceh.base + (i + 1) * CLENBYTES)`.
5. Set the `ceh.type` to `5` (sealed-return), and `ceh.async` to `0` (asynchronous).
6. Write the content of `ceh` to the register `c1`.
7. Write the exception code to the register `x10`.

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

Otherwise:

If the content in `switch_reg` 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. 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 `switch_cap.async` to `1` (asynchronous).
4. Write the content of `switch_cap` to `x[switch_reg]`.
5. Load the program counter `pc` and the stack pointer `sp` from `normal_pc` and `normal_sp` respectively.
6. Write the exit code to `x[exit_reg]`.
7. 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 `x[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.

31 25	24 20	19 15	14 12	11 7	6 0
0b0001100	*	rs1 (I)	0b000	*	0b1011011

Figure 54. SETWORLD instruction format

31 25	24 20	19 15	14 12	11 7	6 0
0b0001101	*	rs1 (I)	0b000	*	0b1011011

Figure 55. 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.

31 25	24 20	19 15	14 12	11 7	6 0
0b0001110	*	rs1 (C)	0b000	*	0b1011011

Figure 56. SETEH instruction format

31 25	24 20	19 15	14 12	11 7	6 0
0b0001111	*	rs1 (I)	0b000	*	0b1011011

Figure 57. 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

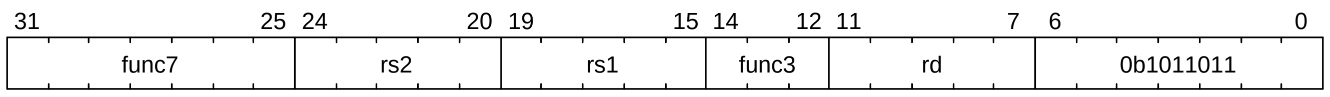

Figure 58. Instruction format: R-type

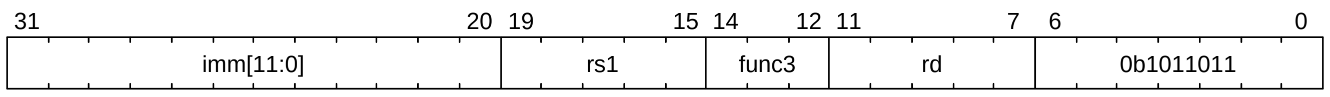

Figure 59. Instruction format: I-type

Table 6. 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 7. 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	R	`001`	`0000100`	C	-	I	-	*	*
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	-	-	-	*	*
CAPGET	R	`001`	`0001100`	-	-	C	-	N	*
CINCOFFSET	R	`001`	`0001101`	C	I	C	-	*	*
CINCOFFSETIMM	I	`011`	-	C	-	C	S	*	*

Table 8. 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 9. 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 10. Control state 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