16-bit Push/Pop instructions and double-moves

Version 0.3.0, 2025-03-27

Table of Contents

Licensing and Acknowledgements

This document is in the Frozen state.

Change is extremely unlikely. A high threshold will be used, and a change will only occur because of some truly critical issue being identified during the public review cycle. Any other desired or needed changes can be the subject of a follow-on new extension.

Copyright and license information

Acknowledgements

Contributors to version 0.3.0 of the specification (in alphabetical order) include:

- James Ball <jameball@qti.qualcomm.com> (Qualcomm Technologies, Inc.)
- Derek Hower <dhower@qti.qualcomm.com> (Qualcomm Technologies, Inc.)
- Ana Pazos <apazos@qti.qualcomm.com> (Qualcomm Technologies, Inc.)
- Albert Yosher <ayosher@qti.qualcomm.com> (Qualcomm Technologies, Inc.)

We express our gratitude to everyone that contributed to, reviewed or improved this specification through their comments and questions.

Versions

This specification documents version 0.3.0 of Xqccmp.

Version History

0.1.0

State

development

0.2.0

State

development

Changes

- Fix all push and pop instructions description for RV64 (IDL code fix)
- Fix qc.cm.pushfp instruction to avoid rlist == 4, since not saving s0 == fp but updating it, should not occur
- Add CSRs qc.mstkbottomaddr and qc.mstktopaddr to support cache range checking
- Add list of supported custom exceptions - stack alignment check and stack range check
- Add stack exception checks in all push/pop instructions

0.3.0

State

frozen

Changes

- Fix all push and pop instructions IDL code to take in consideration that xlen() returns bits and not bytes
- Declare state of extension as "frozen"

Chapter 1. Conventions

Version requirements are specified as conditions using the following operators:

Operator	Meaning
`~> VERSION`	Accepts any version that is *compatible* with `VERSION`. By default, a version A is compatible with a version B if A's version number is greater than or equal to version B. If that default assumption is ever broken, it will be noted in the list of extension versions.
`= VERSION`	Accepts only version `VERSION`.
`>= VERSION`	Accepts any version greater than or equal to `VERSION`.
`<= VERSION`	Accepts any version less than or equal to `VERSION`.
`> VERSION`	Accepts any version greater than `VERSION`.
`< VERSION`	Accepts any version less than `VERSION`.

Chapter 2. Extension description

The Xqccmp extension is a set of instructions which may be executed as a series of existing 32-bit RISC-V instructions.

This extension reuses some encodings from *c.fsdsp*. Therefore it is *incompatible* with [Zcd], which is included when C and D extensions are both present.

Xqccmp is primarily targeted at embedded class CPUs due to implementation complexity. Additionally, it is not compatible with architecture class profiles.

The Xqccmp extension depends on the [Zca] extension.

The Xqccmp extension using same encodings as Zcmp extension for similar instructions, with 2 major differences:

- Order of load and store of registers is opposite to Zcmp order, but compliant with SW ABI
- Xqccmp on top of Zcmp instructions defines qc.cm.pushfp instruction to manage frame pointer

The PUSH/POP assembly syntax uses several variables, the meaning of which are:

- *reg_list* is a list containing 1 to 13 registers (ra and 0 to 12 s registers)
  - valid values: {ra}, {ra, s0}, {ra, s0-s1}, {ra, s0-s2}, ..., {ra, s0-s8}, {ra, s0-s9}, {ra, s0-s11}
  - note that {ra, s0-s10} is *not* valid, giving 12 lists not 13 for better encoding
- *stack_adj* is the total size of the stack frame.
  - valid values vary with register list length and the specific encoding, see the instruction pages for details.

RV32	RV64	Mnemonic	Instruction
yes	yes	qc.cm.push *{reg_list}, -stack_adj*	[insns-qc_cm_push]
yes	yes	qc.cm.pushfp *{reg_list}, -stack_adj*	[insns-qc_cm_pushfp]
yes	yes	qc.cm.pop *{reg_list}, stack_adj*	[insns-qc_cm_pop]
yes	yes	qc.cm.popret *{reg_list}, stack_adj*	[insns-qc_cm_popret]
yes	yes	qc.cm.popretz *{reg_list}, stack_adj*	[insns-qc_cm_popretz]
yes	yes	qc.cm.mva01s *rs1', rs2'*	[insns-qc_cm_mva01s]
yes	yes	qc.cm.mvsa01 *r1s', r2s'*	[insns-qc_cm_mvsa01]

Chapter 3. Instruction summary

The following 7 instructions are affected by this extension:

RV32	RV64	Mnemonic	Instruction
✓	✓	`qc.cm.mva01s r1s, r2s`	Move two s0-s7 registers into a0-a1
✓	✓	`qc.cm.mvsa01 r1s, r2s`	Move a0-a1 into two registers of s0-s7
✓	✓	`qc.cm.pop reg_list, stack_adj`	Destroy function call stack frame
✓	✓	`qc.cm.popret reg_list, stack_adj`	Destroy function call stack frame and return to `ra`.
✓	✓	`qc.cm.popretz reg_list, stack_adj`	Destroy function call stack frame, move zero to `a0` and return to `ra`.
✓	✓	`qc.cm.push reg_list, -stack_adj`	Create function call stack frame
✓	✓	`qc.cm.pushfp reg_list, -stack_adj`	Create function call stack frame with frame pointer

Chapter 4. Instructions (in alphabetical order)

4.1. qc.cm.mva01s

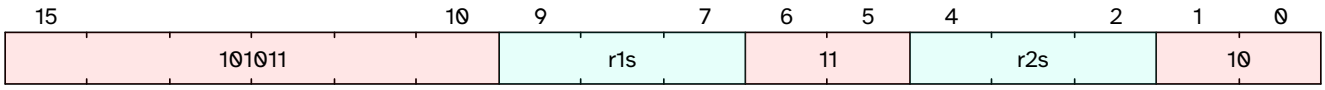
Synopsis

Move two s0-s7 registers into a0-a1

Mnemonic

```
qc.cm.mva01s r1s, r2s
```

Encoding



Description

This instruction moves r1s' into a0 and r2s' into a1. The execution is atomic, so it is not possible to observe state where only one of a0 or a1 have been updated. The encoding uses sreg number specifiers instead of xreg number specifiers to save encoding space. The mapping between them is specified in the pseudo-code below.

Decode Variables

```
Bits<3> r1s = $encoding[9:7];
Bits<3> r2s = $encoding[4:2];
```

Operation

```
if (implemented?(ExtensionName::Zcmp) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg xreg1 = (r1s[2:1] > 0) ? {1, 0, r1s[2:0]} : {0, 1, r1s[2:0]};
XReg xreg2 = (r2s[2:1] > 0) ? {1, 0, r2s[2:0]} : {0, 1, r2s[2:0]};
X[10] = X[xreg1];
X[11] = X[xreg2];
```

Included in

Extension	Version
Xqccmp	~> 0.1.0

4.2. qc.cm.mvsa01

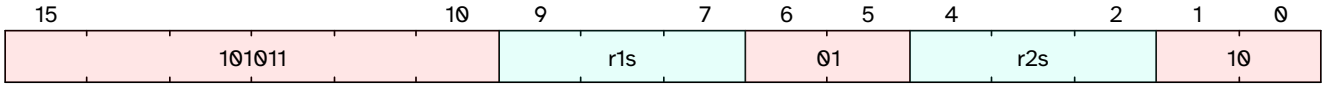
Synopsis

Move a0-a1 into two registers of s0-s7

Mnemonic

```
qc.cm.mvsa01 r1s, r2s
```

Encoding



Description

This instruction moves a0 into r1s' and a1 into r2s'. r1s' and r2s' must be different. The execution is atomic, so it is not possible to observe state where only one of r1s' or r2s' has been updated. The encoding uses sreg number specifiers instead of xreg number specifiers to save encoding space. The mapping between them is specified in the pseudo-code below.

Decode Variables

```
Bits<3> r1s = $encoding[9:7];
Bits<3> r2s = $encoding[4:2];
```

Operation

```
if (implemented?(ExtensionName::Zcmp) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg xreg1 = (r1s[2:1] > 0) ? {1, 0, r1s[2:0]} : {0, 1, r1s[2:0]};
XReg xreg2 = (r2s[2:1] > 0) ? {1, 0, r2s[2:0]} : {0, 1, r2s[2:0]};
X[xreg1] = X[10];
X[xreg2] = X[11];
```

Included in

Extension	Version
Xqccmp	~> 0.1.0

4.3. qc.cm.pop

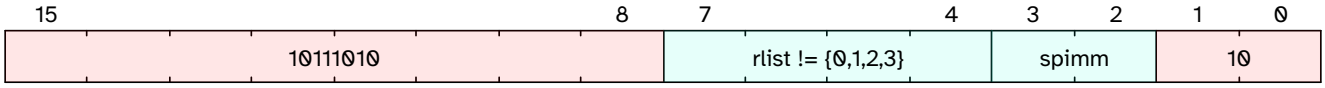
Synopsis

Destroy function call stack frame

Mnemonic

```
qc.cm.pop reg_list, stack_adj
```

Encoding



Description

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame. This instruction pops (loads) the registers in `reg_list` from stack memory, and then adjusts the stack pointer by `stack_adj`.

Restrictions on stack_adj:

- it must be enough to store all of the listed registers
- it must be a multiple of 16 (bytes):
  - for RV32 the allowed values are: 16, 32, 48, 64, 80, 96, 112
  - for RV64 the allowed values are: 16, 32, 48, 64, 80, 96, 112, 128, 144, 160

Decode Variables

```
Bits<4> rlist = $encoding[7:4];
Bits<2> spimm = $encoding[3:2];
```

Operation

```
if (implemented?(ExtensionName::Xqccmp) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address_sp = get_and_validate_stack_pointer(X[2], $encoding);
XReg size = xlen() / 8;
XReg nreg = (rlist == 15) ? 13 : (rlist - 3);
XReg stack_aligned_adj = (nreg * size + 15) & ~0xF;
XReg virtual_address_new_sp = virtual_address_sp + stack_aligned_adj + spimm;
XReg virtual_address_base = virtual_address_new_sp - size;
X[1] = read_memory_xlen(virtual_address_base - 0 * size, $encoding);
if (nreg > 1) {
  X[8] = read_memory_xlen(virtual_address_base - 1 * size, $encoding);
}
if (nreg > 2) {
```

```
  X[9] = read_memory_xlen(virtual_address_base - 2 * size, $encoding);
}
if (nreg > 3) {
  X[18] = read_memory_xlen(virtual_address_base - 3 * size, $encoding);
}
if (nreg > 4) {
  X[19] = read_memory_xlen(virtual_address_base - 4 * size, $encoding);
}
if (nreg > 5) {
  X[20] = read_memory_xlen(virtual_address_base - 5 * size, $encoding);
}
if (nreg > 6) {
  X[21] = read_memory_xlen(virtual_address_base - 6 * size, $encoding);
}
if (nreg > 7) {
  X[22] = read_memory_xlen(virtual_address_base - 7 * size, $encoding);
}
if (nreg > 8) {
  X[23] = read_memory_xlen(virtual_address_base - 8 * size, $encoding);
}
if (nreg > 9) {
  X[24] = read_memory_xlen(virtual_address_base - 9 * size, $encoding);
}
if (nreg > 10) {
  X[25] = read_memory_xlen(virtual_address_base - 10 * size, $encoding);
}
if (nreg > 11) {
  X[26] = read_memory_xlen(virtual_address_base - 11 * size, $encoding);
  X[27] = read_memory_xlen(virtual_address_base - 12 * size, $encoding);
}
X[2] = virtual_address_new_sp;
```

Included in

Extension	Version
Xqccmp	~> 0.1.0

4.4. qc.cm.popret

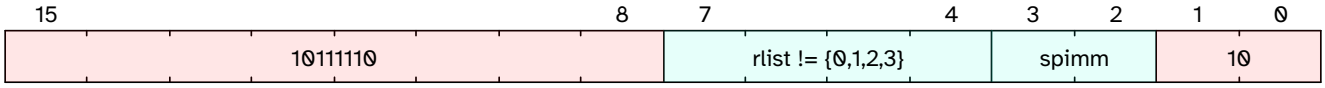
Synopsis

Destroy function call stack frame and return to `ra`.

Mnemonic

```
qc.cm.popret reg_list, stack_adj
```

Encoding



Description

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame, return to `ra`. This instruction pops (loads) the registers in `reg_list` from stack memory, and then adjusts the stack pointer by `stack_adj` and then return to ra.

Restrictions on stack_adj:

- it must be enough to store all of the listed registers
- it must be a multiple of 16 (bytes):
  - for RV32 the allowed values are: 16, 32, 48, 64, 80, 96, 112
  - for RV64 the allowed values are: 16, 32, 48, 64, 80, 96, 112, 128, 144, 160

Decode Variables

```
Bits<4> rlist = $encoding[7:4];
Bits<2> spimm = $encoding[3:2];
```

Operation

```
if (implemented?(ExtensionName::Xqccmp) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address_sp = get_and_validate_stack_pointer(X[2], $encoding);
XReg size = xlen() / 8;
XReg nreg = (rlist == 15) ? 13 : (rlist - 3);
XReg stack_aligned_adj = (nreg * size + 15) & ~0xF;
XReg virtual_address_new_sp = virtual_address_sp + stack_aligned_adj + spimm;
XReg virtual_address_base = virtual_address_new_sp - size;
X[1] = read_memory_xlen(virtual_address_base - 0 * size, $encoding);
if (nreg > 1) {
  X[8] = read_memory_xlen(virtual_address_base - 1 * size, $encoding);
}
if (nreg > 2) {
```

```
  X[9] = read_memory_xlen(virtual_address_base - 2 * size, $encoding);
}
if (nreg > 3) {
  X[18] = read_memory_xlen(virtual_address_base - 3 * size, $encoding);
}
if (nreg > 4) {
  X[19] = read_memory_xlen(virtual_address_base - 4 * size, $encoding);
}
if (nreg > 5) {
  X[20] = read_memory_xlen(virtual_address_base - 5 * size, $encoding);
}
if (nreg > 6) {
  X[21] = read_memory_xlen(virtual_address_base - 6 * size, $encoding);
}
if (nreg > 7) {
  X[22] = read_memory_xlen(virtual_address_base - 7 * size, $encoding);
}
if (nreg > 8) {
  X[23] = read_memory_xlen(virtual_address_base - 8 * size, $encoding);
}
if (nreg > 9) {
  X[24] = read_memory_xlen(virtual_address_base - 9 * size, $encoding);
}
if (nreg > 10) {
  X[25] = read_memory_xlen(virtual_address_base - 10 * size, $encoding);
}
if (nreg > 11) {
  X[26] = read_memory_xlen(virtual_address_base - 11 * size, $encoding);
  X[27] = read_memory_xlen(virtual_address_base - 12 * size, $encoding);
}
X[2] = virtual_address_new_sp;
jump(X[1]);
```

Included in

Extension	Version
Xqccmp	~> 0.1.0

4.5. qc.cm.popretz

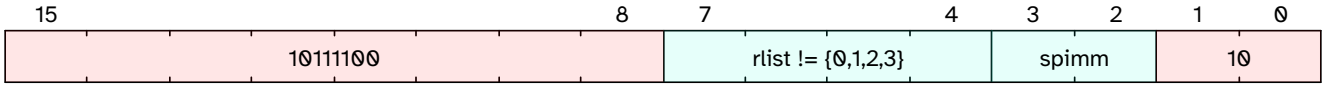
Synopsis

Destroy function call stack frame, move zero to `a0` and return to `ra`.

Mnemonic

```
qc.cm.popretz reg_list, stack_adj
```

Encoding



Description

Destroy stack frame: load `ra` and `0` to 12 saved registers from the stack frame, deallocate the stack frame, move zero to `a0`, return to `ra`. This instruction pops (loads) the registers in `reg_list` from stack memory, and then adjusts the stack pointer by `stack_adj`, move zero to `a0` and then return to `ra`.

Restrictions on stack_adj:

- it must be enough to store all of the listed registers
- it must be a multiple of 16 (bytes):
  - for RV32 the allowed values are: 16, 32, 48, 64, 80, 96, 112
  - for RV64 the allowed values are: 16, 32, 48, 64, 80, 96, 112, 128, 144, 160

Decode Variables

```
Bits<4> rlist = $encoding[7:4];
Bits<2> spimm = $encoding[3:2];
```

Operation

```
if (implemented?(ExtensionName::Xqccmp) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address_sp = get_and_validate_stack_pointer(X[2], $encoding);
XReg size = xlen() / 8;
XReg nreg = (rlist == 15) ? 13 : (rlist - 3);
XReg stack_aligned_adj = (nreg * size + 15) & ~0xF;
XReg virtual_address_new_sp = virtual_address_sp + stack_aligned_adj + spimm;
XReg virtual_address_base = virtual_address_new_sp - size;
X[1] = read_memory_xlen(virtual_address_base - 0 * size, $encoding);
if (nreg > 1) {
  X[8] = read_memory_xlen(virtual_address_base - 1 * size, $encoding);
}
if (nreg > 2) {
```

```
  X[9] = read_memory_xlen(virtual_address_base - 2 * size, $encoding);
}
if (nreg > 3) {
  X[18] = read_memory_xlen(virtual_address_base - 3 * size, $encoding);
}
if (nreg > 4) {
  X[19] = read_memory_xlen(virtual_address_base - 4 * size, $encoding);
}
if (nreg > 5) {
  X[20] = read_memory_xlen(virtual_address_base - 5 * size, $encoding);
}
if (nreg > 6) {
  X[21] = read_memory_xlen(virtual_address_base - 6 * size, $encoding);
}
if (nreg > 7) {
  X[22] = read_memory_xlen(virtual_address_base - 7 * size, $encoding);
}
if (nreg > 8) {
  X[23] = read_memory_xlen(virtual_address_base - 8 * size, $encoding);
}
if (nreg > 9) {
  X[24] = read_memory_xlen(virtual_address_base - 9 * size, $encoding);
}
if (nreg > 10) {
  X[25] = read_memory_xlen(virtual_address_base - 10 * size, $encoding);
}
if (nreg > 11) {
  X[26] = read_memory_xlen(virtual_address_base - 11 * size, $encoding);
  X[27] = read_memory_xlen(virtual_address_base - 12 * size, $encoding);
}
X[2] = virtual_address_new_sp;
X[10] = 0;
jump(X[1]);
```

Included in

Extension	Version
Xqccmp	~> 0.1.0

4.6. qc.cm.push

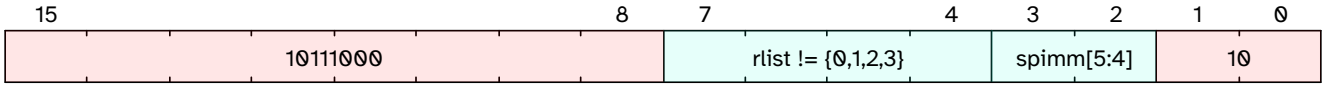
Synopsis

Create function call stack frame

Mnemonic

```
qc.cm.push reg_list, -stack_adj
```

Encoding



Description

Create stack frame: store `ra` and `0` to 12 saved registers to the stack frame, optionally allocate additional stack space. This instruction pushes (stores) the registers in rlist to the memory below the stack pointer, and then creates the stack frame by decrementing the stack pointer by stack_adj, including any additional stack space requested by the value of `spimm`.

Restrictions on stack_adj:

- it must be enough to store all of the listed registers
- it must be a multiple of 16 (bytes):
  - for RV32 the allowed values are: 16, 32, 48, 64, 80, 96, 112
  - for RV64 the allowed values are: 16, 32, 48, 64, 80, 96, 112, 128, 144, 160

Decode Variables

```
Bits<4> rlist = $encoding[7:4];
Bits<6> spimm = {$encoding[3:2], 4'd0};
```

Operation

```
if (implemented?(ExtensionName::Xqccmp) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address_sp = get_and_validate_stack_pointer(X[2], $encoding);
XReg size = xlen() / 8;
XReg nreg = (rlist == 15) ? 13 : (rlist - 3);
XReg stack_aligned_adj = (nreg * size + 15) & ~0xF;
XReg virtual_address_new_sp = virtual_address_sp - stack_aligned_adj - spimm;
XReg virtual_address_base = virtual_address_sp - size;
write_memory_xlen(virtual_address_base - 0 * size, X[1], $encoding);
if (nreg > 1) {
  write_memory_xlen(virtual_address_base - 1 * size, X[8], $encoding);
}
```

```
if (nreg > 2) {
  write_memory_xlen(virtual_address_base - 2 * size, X[9], $encoding);
}
if (nreg > 3) {
  write_memory_xlen(virtual_address_base - 3 * size, X[18], $encoding);
}
if (nreg > 4) {
  write_memory_xlen(virtual_address_base - 4 * size, X[19], $encoding);
}
if (nreg > 5) {
  write_memory_xlen(virtual_address_base - 5 * size, X[20], $encoding);
}
if (nreg > 6) {
  write_memory_xlen(virtual_address_base - 6 * size, X[21], $encoding);
}
if (nreg > 7) {
  write_memory_xlen(virtual_address_base - 7 * size, X[22], $encoding);
}
if (nreg > 8) {
  write_memory_xlen(virtual_address_base - 8 * size, X[23], $encoding);
}
if (nreg > 9) {
  write_memory_xlen(virtual_address_base - 9 * size, X[24], $encoding);
}
if (nreg > 10) {
  write_memory_xlen(virtual_address_base - 10 * size, X[25], $encoding);
}
if (nreg > 11) {
  write_memory_xlen(virtual_address_base - 11 * size, X[26], $encoding);
  write_memory_xlen(virtual_address_base - 12 * size, X[27], $encoding);
}
X[2] = virtual_address_new_sp;
```

Included in

Extension	Version
Xqccmp	~> 0.1.0

4.7. qc.cm.pushfp

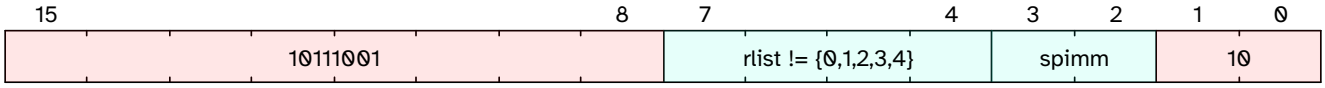
Synopsis

Create function call stack frame with frame pointer

Mnemonic

```
qc.cm.pushfp reg_list, -stack_adj
```

Encoding



Description

Create stack frame: store `ra` and `0` to 12 saved registers to the stack frame, optionally allocate additional stack space. As well, instruction initializes frame pointer (`fp`, aka `x8`) to the Canonical Frame Address (`sp` aka `x2`) on function entry. This instruction pushes (stores) the registers in rlist to the memory below the stack pointer, and then creates the stack frame by decrementing the stack pointer by stack_adj, including any additional stack space requested by the value of `spimm`.

Restrictions on stack_adj:

- it must be enough to store all of the listed registers
- it must be a multiple of 16 (bytes):
  - for RV32 the allowed values are: 16, 32, 48, 64, 80, 96, 112
  - for RV64 the allowed values are: 16, 32, 48, 64, 80, 96, 112, 128, 144, 160

Decode Variables

```
Bits<4> rlist = $encoding[7:4];
Bits<2> spimm = $encoding[3:2];
```

Operation

```
if (implemented?(ExtensionName::Xqccmp) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address_sp = get_and_validate_stack_pointer(X[2], $encoding);
XReg size = xlen() / 8;
XReg nreg = (rlist == 15) ? 13 : (rlist - 3);
XReg stack_aligned_adj = (nreg * size + 15) & ~0xF;
XReg virtual_address_new_sp = virtual_address_sp - stack_aligned_adj - spimm;
XReg virtual_address_base = virtual_address_sp - size;
write_memory_xlen(virtual_address_base - 0 * size, X[1], $encoding);
if (nreg > 1) {
  write_memory_xlen(virtual_address_base - 1 * size, X[8], $encoding);
```

```
}
if (nreg > 2) {
  write_memory_xlen(virtual_address_base - 2 * size, X[9], $encoding);
}
if (nreg > 3) {
  write_memory_xlen(virtual_address_base - 3 * size, X[18], $encoding);
}
if (nreg > 4) {
  write_memory_xlen(virtual_address_base - 4 * size, X[19], $encoding);
}
if (nreg > 5) {
  write_memory_xlen(virtual_address_base - 5 * size, X[20], $encoding);
}
if (nreg > 6) {
  write_memory_xlen(virtual_address_base - 6 * size, X[21], $encoding);
}
if (nreg > 7) {
  write_memory_xlen(virtual_address_base - 7 * size, X[22], $encoding);
}
if (nreg > 8) {
  write_memory_xlen(virtual_address_base - 8 * size, X[23], $encoding);
}
if (nreg > 9) {
  write_memory_xlen(virtual_address_base - 9 * size, X[24], $encoding);
}
if (nreg > 10) {
  write_memory_xlen(virtual_address_base - 10 * size, X[25], $encoding);
}
if (nreg > 11) {
  write_memory_xlen(virtual_address_base - 11 * size, X[26], $encoding);
  write_memory_xlen(virtual_address_base - 12 * size, X[27], $encoding);
}
X[8] = virtual_address_sp;
X[2] = virtual_address_new_sp;
```

Included in

Extension	Version
Xqccmp	~> 0.1.0

Chapter 5. IDL Functions

5.1. abort_current_instruction (builtin)

Abort the current instruction, and start refetching from $pc.

Return Type	void
Arguments	

5.2. access_check

Checks if the physical address paddr is able to access memory, and raises the appropriate exception if not.

Return Type	void
Arguments	Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size, XReg vaddr, MemoryOperation type, ExceptionCode fault_type, PrivilegeMode from_mode

```
if (paddr > 1 << PHYS_ADDR_WIDTH) - access_size {
  raise(fault_type, from_mode, vaddr);
}
if (implemented?(ExtensionName::Smpmp)) {
  if (!pmp_check(paddr[PHYS_ADDR_WIDTH - 1:0], access_size, type)) {
    raise(fault_type, from_mode, vaddr);
  }
}
```

5.3. assert (builtin)

Assert that a condition is true. Failure represents an error in the IDL model.

Return Type	void
Arguments	Boolean test, String message

5.4. cached_translation (generated)

Possibly returns a cached translation result matching vaddr.

CachedTranslationResult contains a Boolean 'valid' field. If valid, 'result' is a usable translation. Otherwise, the cache lookup failed.

Return Type	CachedTranslationResult
Arguments	XReg vaddr, MemoryOperation op

5.5. current_translation_mode

Returns the current first-stage translation mode for an explicit load or store from mode given the machine state (e.g., value of satp or vsatp csr).

Returns SatpMode::Reserved if the setting found in satp or vsatp is invalid.

Return Type	SatpMode
Arguments	PrivilegeMode mode

```
PrivilegeMode effective_mode = effective_ldst_mode();
if (effective_mode == PrivilegeMode::M) {
  return SatpMode::Bare;
}
if (CSR[misa].H == 1'b1) {
  if (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode
::VU) {
    Bits<4> mode_val = CSR[vsatp].MODE;
    if (mode_val == $bits(SatpMode::Sv32)) {
      if (XLEN == 64) {
        if ((effective_mode == PrivilegeMode::VS) && (CSR[hstatus].VSXL !=
$bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
        }
        if ((effective_mode == PrivilegeMode::VU) && (CSR[vsstatus].UXL !=
$bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
        }
      }
      if (!SV32_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv32;
    } else if ((XLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
      if (effective_mode == PrivilegeMode::VS && CSR[hstatus].VSXL != $bits
(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && CSR[vsstatus].UXL != $bits
(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV39_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv39;
    } else if ((XLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
      if (effective_mode == PrivilegeMode::VS && CSR[hstatus].VSXL != $bits
(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
```

```
      }
      if (effective_mode == PrivilegeMode::VU && CSR[vsstatus].UXL != $bits
(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV48_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv48;
    } else if ((XLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
      if (effective_mode == PrivilegeMode::VS && CSR[hstatus].VSXL != $bits
(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && CSR[vsstatus].UXL != $bits
(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV57_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv57;
    } else {
      return SatpMode::Reserved;
    }
  }
} else if (CSR[misa].S == 1'b1) {
  assert(effective_mode == PrivilegeMode::S || effective_mode ==
PrivilegeMode::U, "unexpected priv mode");
  Bits<4> mode_val = CSR[satp].MODE;
  if (mode_val == $bits(SatpMode::Sv32)) {
    if (XLEN == 64) {
      if (effective_mode == PrivilegeMode::S && CSR[mstatus].SXL != $bits
(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::U && CSR[sstatus].UXL != $bits
(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
    }
    if (!implemented?(ExtensionName::Sv32)) {
      return SatpMode::Reserved;
    }
  } else if ((XLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
    if (effective_mode == PrivilegeMode::S && CSR[mstatus].SXL != $bits
(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && CSR[sstatus].UXL != $bits
(XRegWidth::XLEN64)) {
```

```
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv39)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv39;
  } else if ((XLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
    if (effective_mode == PrivilegeMode::S && CSR[mstatus].SXL != $bits
(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && CSR[sstatus].UXL != $bits
(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv48)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv48;
  } else if ((XLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
    if (effective_mode == PrivilegeMode::S && CSR[mstatus].SXL != $bits
(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && CSR[sstatus].UXL != $bits
(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv57)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv57;
  } else {
    return SatpMode::Reserved;
  }
}
```

5.6. effective_ldst_mode

Returns the effective privilege mode for normal explicit loads and stores, taking into account the current actual privilege mode and modifications from mstatus.MPRV.

Return Type	`PrivilegeMode`
Arguments	

```
if (mode() == PrivilegeMode::M) {
  if (CSR[misa].U == 1 && CSR[mstatus].MPRV == 1) {
    if (CSR[mstatus].MPP == 0b00) {
```

```
      if (CSR[misa].H == 1 && mpv() == 0b1) {
        return PrivilegeMode::VU;
      } else {
        return PrivilegeMode::U;
      }
    } else if (CSR[misa].S == 1 && CSR[mstatus].MPP == 0b01) {
      if (CSR[misa].H == 1 && mpv() == 0b1) {
        return PrivilegeMode::VS;
      } else {
        return PrivilegeMode::S;
      }
    }
  }
}
return mode();
```

5.7. exception_handling_mode

Returns the target privilege mode that will handle synchronous exception `exception_code`

Return Type	PrivilegeMode
Arguments	ExceptionCode exception_code

```
if (mode() == PrivilegeMode::M) {
  return PrivilegeMode::M;
} else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::HS) ||
(mode() == PrivilegeMode::U) {
  if (($bits(CSR[medeleg]) & (1 << $bits(exception_code))) != 0) {
    return PrivilegeMode::HS;
  } else {
    return PrivilegeMode::M;
  }
} else {
  assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) ||
(mode() == PrivilegeMode::VU, "Unexpected mode");
  if (($bits(CSR[medeleg]) & (1 << $bits(exception_code))) != 0) {
    if (($bits(CSR[hedeleg]) & (1 << $bits(exception_code))) != 0) {
      return PrivilegeMode::VS;
    } else {
      return PrivilegeMode::HS;
    }
  } else {
    return PrivilegeMode::M;
  }
}
```

5.8. get_and_validate_stack_pointer

Validate and return stack pointer. Stack pointer is validated to fit in range, defined by qc.mstktopaddr and qc.mstkbottomaddr CSRs. In case when stack pointer is out-of-range, raising exception SpOutOfRange. Stack pointer is also validated to be 16-byte aligned. In case when stack pointer is not 16-bytes aligned, raising exception IllegalStackPointer.

Return Type	XReg
Arguments	XReg sp, Bits<INSTR_ENC_SIZE> encoding

```
if (sp < $bits(CSR[qc.mstkbottomaddr])) {
  raise(ExceptionCode::SpOutOfRange, mode(), encoding);
} else if (sp > $bits(CSR[qc.mstktopaddr])) {
  raise(ExceptionCode::SpOutOfRange, mode(), encoding);
} else if ((sp & 0xf) != 0) {
  raise(ExceptionCode::IllegalStackPointer, mode(), encoding);
}
return sp;
```

5.9. ialign

Returns IALIGN, the smallest instruction encoding size, in bits.

Return Type	Bits<6>
Arguments	

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 0x1)) {
  return 16;
} else {
  return 32;
}
```

5.10. implemented? (generated)

Return true if the implementation supports `extension`.

Return Type	Boolean
Arguments	ExtensionName extension

5.11. implemented_version? (generated)

Return true if the implementation supports `extension` meeting 'version_requirement'.

Return Type	Boolean

Arguments	ExtensionName extension, String version_requirement

5.12. in_naturally_aligned_region?

Checks if a length-bit access starting at address lies entirely within an N-bit naturally-aligned region.

Return Type	Boolean
Arguments	XReg address, U32 length

```
XReg Mask = (N / 8) - 1;
return (address & ~Mask) == ((address + length - 1) & ~Mask);
```

5.13. is_naturally_aligned

Checks if value is naturally aligned to N bits.

Return Type	Boolean
Arguments	XReg value

```
return true if (N == 8);
XReg Mask = (N / 8) - 1;
return (value & ~Mask) == value;
```

5.14. jump

Jump to virtual address `target_addr`.

If target address is misaligned, raise a `MisalignedAddress` exception.

Return Type	void
Arguments	XReg target_addr

```
if ((ialign() == 16) && target_addr & 0x1) != 0 {
  raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_addr);
} else if ((ialign() == 32) && (target_addr & 0x3) != 0) {
  raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_addr);
}
$pc = target_addr;
```

5.15. maybe_cache_translation (generated)

Given a translation result, potentially cache the result for later use. This function models a TLB fill operation. A valid implementation does nothing.

Return Type	`void`
Arguments	`XReg vaddr, MemoryOperation op, TranslationResult result`

5.16. misaligned_is_atomic?

Returns true if an access starting at physical_address that is N bits long is atomic.

This function takes into account any Atomicity Granule PMAs, so **it should not be used for load-reserved/store-conditional**, since those PMAs do not apply to those accesses.

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> physical_address`

```
return false if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE == 0);
if (pma_applies?(PmaAttribute::MAG16, physical_address, N) &&
in_naturally_aligned_region?<128>(physical_address, N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG8, physical_address, N) &&
in_naturally_aligned_region?<4>(physical_address, N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG4, physical_address, N) &&
in_naturally_aligned_region?<32>(physical_address, N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG2, physical_address, N) &&
in_naturally_aligned_region?<16>(physical_address, N)) {
  return true;
} else {
  return false;
}
```

5.17. mode

Returns the current active privilege mode.

Return Type	`PrivilegeMode`
Arguments	

```
if (!implemented?(ExtensionName::S) && !implemented?(ExtensionName::U) &&
!implemented?(ExtensionName::H)) {
  return PrivilegeMode::M;
} else {
```

```
  return current_mode;
}
```

5.18. mtval_for

Given an exception code and a **legal** non-zero value for mtval, returns the value to be written in mtval considering implementation options

Return Type	XReg
Arguments	ExceptionCode exception_code, XReg tval

```
if (exception_code == ExceptionCode::Breakpoint) {
  return REPORT_VA_IN_MTVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
  return REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
  return tval;
} else {
  return 0;
}
```

5.19. mtval_readonly?

Returns whether or not CSR[mtval] is read-only based on implementation options

Return Type	Boolean
Arguments	

```
return !(REPORT_VA_IN_MTVAL_ON_BREAKPOINT ||
REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED ||
REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED ||
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED ||
REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT ||
REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT ||
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT ||
REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT ||
REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT ||
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT ||
REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION ||
REPORT_CAUSE_IN_MTVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_MTVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
```

5.20. notify_mode_change (builtin)

Called whenever the privilege mode changes. Downstream tools can use this to hook events.

Return Type	`void`
Arguments	`PrivilegeMode new_mode, PrivilegeMode old_mode`

5.21. pma_applies? (builtin)

Checks if *attr* is applied to the entire physical address region between [paddr, paddr + len) based on static PMA attributes.

Return Type	`Boolean`
Arguments	`PmaAttribute attr, Bits<PHYS_ADDR_WIDTH> paddr, U32 len`

5.22. raise

Raise synchronous exception number `exception_code`.

The exception may be imprecise, and will cause execution to enter an unpredictable state, if PRECISE_SYNCHRONOUS_EXCEPTIONS is false.

Otherwise, the exception will be precise.

Return Type	`void`
Arguments	`ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval`

```
if (!PRECISE_SYNCHRONOUS_EXCEPTIONS) {
```

```
  unpredictable("Imprecise synchronous exception");
} else {
  raise_precise(exception_code, from_mode, tval);
}
```

5.23. raise_precise

Raise synchronous exception number `exception_code`.

Return Type	void
Arguments	ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval

```
PrivilegeMode handling_mode = exception_handling_mode(exception_code);
if (handling_mode == PrivilegeMode::M) {
  CSR[mepc].PC = $pc;
  if (!mtval_readonly?()) {
    CSR[mtval].VALUE = mtval_for(exception_code, tval);
  }
  $pc = {CSR[mtvec].BASE, 2'b00};
  CSR[mcause].INT = 1'b0;
  CSR[mcause].CODE = $bits(exception_code);
  if (CSR[misa].H == 1) {
    CSR[mtval2].VALUE = 0;
    CSR[mtinst].VALUE = 0;
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      if (XLEN == 32) {
        CSR[mstatush].MPV = 1;
      } else {
        CSR[mstatus].MPV = 1;
      }
    } else {
      if (XLEN == 32) {
        CSR[mstatush].MPV = 0;
      } else {
        CSR[mstatus].MPV = 0;
      }
    }
  }
  CSR[mstatus].MPP = $bits(from_mode);
} else if (CSR[misa].S == 1 && (handling_mode == PrivilegeMode::S)) {
  CSR[sepc].PC = $pc;
  if (!stval_readonly?()) {
    CSR[stval].VALUE = stval_for(exception_code, tval);
  }
  $pc = {CSR[stvec].BASE, 2'b00};
  CSR[scause].INT = 1'b0;
  CSR[scause].CODE = $bits(exception_code);
```

```
  CSR[mstatus].SPP = $bits(from_mode)[0];
  if (CSR[misa].H == 1) {
    CSR[htval].VALUE = 0;
    CSR[htinst].VALUE = 0;
    CSR[hstatus].SPV = $bits(from_mode)[2];
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      CSR[hstatus].SPV = 1;
      if (exception_code == ExceptionCode::Breakpoint) && (
REPORT_VA_IN_STVAL_ON_BREAKPOINT || exception_code == ExceptionCode
::LoadAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED ||
exception_code == ExceptionCode::StoreAmoAddressMisaligned) && (
REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED || exception_code == ExceptionCode
::InstructionAddressMisaligned) && (
REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED || exception_code ==
ExceptionCode::LoadAccessFault) && (REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT ||
exception_code == ExceptionCode::StoreAmoAccessFault) && (
REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT || exception_code ==
ExceptionCode::InstructionAccessFault) && (
REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT || exception_code ==
ExceptionCode::LoadPageFault) && (REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT ||
exception_code == ExceptionCode::StoreAmoPageFault) && (
REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT || exception_code == ExceptionCode
::InstructionPageFault) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT) {
        CSR[hstatus].GVA = 1;
      } else {
        CSR[hstatus].GVA = 0;
      }
      CSR[hstatus].SPVP = $bits(from_mode)[0];
    } else {
      CSR[hstatus].SPV = 0;
      CSR[hstatus].GVA = 0;
    }
  }
} else if (CSR[misa].H == 1 && (handling_mode == PrivilegeMode::VS)) {
  CSR[vsepc].PC = $pc;
  if (!vstval_readonly?()) {
    CSR[vstval].VALUE = vstval_for(exception_code, tval);
  }
  $pc = {CSR[vstvec].BASE, 2'b00};
  CSR[vscause].INT = 1'b0;
  CSR[vscause].CODE = $bits(exception_code);
  CSR[vsstatus].SPP = $bits(from_mode)[0];
}
set_mode(handling_mode);
abort_current_instruction();
```

5.24. read_memory

Read from virtual memory.

Return Type	Bits<LEN>
Arguments	XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
  return read_memory_aligned<LEN>(virtual_address, encoding);
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
  assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't
mix low-priority misaligned exceptions with large atomicity granule");
  physical_address = (CSR[misa].S == 1) ? translate(virtual_address,
MemoryOperation::Read, effective_ldst_mode(), encoding).paddr :
virtual_address;
  if (misaligned_is_atomic?<LEN>(physical_address)) {
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Read,
ExceptionCode::LoadAccessFault, effective_ldst_mode());
    return read_physical_memory<LEN>(physical_address);
  }
}
if (!MISALIGNED_LDST) {
  if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
    physical_address = (CSR[misa].S == 1) ? translate(virtual_address,
MemoryOperation::Read, effective_ldst_mode(), encoding).paddr :
virtual_address;
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Read,
ExceptionCode::LoadAccessFault, effective_ldst_mode());
  }
  raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(),
virtual_address);
} else {
  if (MISALIGNED_SPLIT_STRATEGY == "by_byte") {
    Bits<LEN> result = 0;
    for (U32 i = 0; i <= (LEN / 8); i++) {
      result = result | (read_memory_aligned<8>(virtual_address + i, encoding)
<< (8 * i));
    }
    return result;
  } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any
way, leading to unpredictable behavior when any part of the misaligned access
causes an exception");
  }
}
```

5.25. read_memory_aligned

Read from virtual memory using a known aligned address.

Return Type	`Bits<LEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
TranslationResult result;
if (CSR[misa].S == 1) {
  result = translate(virtual_address, MemoryOperation::Read,
effective_ldst_mode(), encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, LEN, virtual_address, MemoryOperation::Read,
ExceptionCode::LoadAccessFault, effective_ldst_mode());
return read_physical_memory<LEN>(result.paddr);
```

5.26. read_memory_xlen

Read XLEN bits from memory

Return Type	`Bits<XLEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
if (xlen() == 32) {
  return read_memory<32>(virtual_address, encoding);
} else {
  return read_memory<64>(virtual_address, encoding);
}
```

5.27. read_physical_memory

Read from physical memory.

Return Type	`Bits<len>`
Arguments	`XReg paddr`

```
if (len == 8) {
  return read_physical_memory_8(paddr);
} else if (len == 16) {
  return read_physical_memory_16(paddr);
} else if (len == 32) {
```

```
  return read_physical_memory_32(paddr);
} else if (len == 64) {
  return read_physical_memory_64(paddr);
} else {
  assert(false, "Invalid len");
}
```

5.28. read_physical_memory_32 (builtin)

Read four bytes from physical memory.

Return Type	`Bits<32>`
Arguments	`XReg paddr`

5.29. read_physical_memory_8 (builtin)

Read a byte from physical memory.

Return Type	`Bits<8>`
Arguments	`XReg paddr`

5.30. refresh_pending_interrupts

refreshes the calculation of a pending interrupt

needs to be called after any state update that could change a pending interrupt. This includes: - CSR[mip] - CSR[mie] - CSR[mstatus].MIE - CSR[mstatus].SIE - CSR[vsstatus].SIE - CSR[mideleg] - CSR[sideleg] - CSR[hideleg] - CSR[hvip] - CSR[hgeip] - CSR[hgeie] - mode changes

Return Type	`void`
Arguments	

```
Bits<XLEN> pending_ints = CSR[mip].sw_read() & $bits(CSR[mie]);
if (pending_ints == 0) {
  pending_and_enabled_interrupts = 0;
  return ;
}
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") ||
(implemented_version?(ExtensionName::S, "> 1.9.1") && implemented_version?
(ExtensionName::Sm, "> 1.9.1"));
Bits<XLEN> mmode_enabled_ints = mode() == PrivilegeMode::M) && (CSR[mstatus
].MIE == 1'b0 ? 0 : ($bits(CSR[mie]) & (HAS_MIDELEG ? ~$bits(CSR[mideleg]) :
~XLEN'0));
Bits<XLEN> mmode_pending_and_enabled = pending_ints & mmode_enabled_ints;
if (mmode_pending_and_enabled != 0) {
  pending_and_enabled_interrupts = mmode_pending_and_enabled;
```

```
  return ;
}
if (CSR[misa].S == 1'b1) {
  Bits<XLEN> smode_enabled_ints = mode() == PrivilegeMode::M) || (CSR[mstatus
].SIE == 1'b0 ? 0 : $bits(CSR[mie]) & ($bits(CSR[mideleg]));
  Bits<XLEN> smode_pending_and_enabled = pending_ints & smode_enabled_ints;
  if (smode_pending_and_enabled != 0) {
    pending_and_enabled_interrupts = smode_pending_and_enabled;
    return ;
  }
}
pending_and_enabled_interrupts = 0;
```

5.31. set_mode

Set the current privilege mode to `new_mode`

Return Type	void
Arguments	PrivilegeMode new_mode

```
if (new_mode != current_mode) {
  notify_mode_change(new_mode, current_mode);
  current_mode = new_mode;
  refresh_pending_interrupts();
}
```

5.32. translate

Translate a virtual address for operation type op that appears to execute at effective_mode.

The translation will depend on the effective privilege mode.

May raise a Page Fault or Access Fault.

The final physical address is **not** access checked (for PMP, PMA, etc., violations). (though intermediate page table reads will be)

Return Type	TranslationResult
Arguments	XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding

```
Boolean cached_translation_valid;
CachedTranslationResult cached_translation_result;
cached_translation_result = cached_translation(vaddr, op);
if (cached_translation_result.valid) {
```

```
  return cached_translation_result.result;
}
TranslationResult result;
if (effective_mode == PrivilegeMode::M) {
  result.paddr = vaddr;
  return result;
}
SatpMode translation_mode = current_translation_mode(effective_mode);
if (translation_mode == SatpMode::Reserved) {
  if (op == MemoryOperation::Read) {
    raise(ExceptionCode::LoadPageFault, effective_mode, vaddr);
  } else if (op == MemoryOperation::Write || op == MemoryOperation
::ReadModifyWrite) {
    raise(ExceptionCode::StoreAmoPageFault, effective_mode, vaddr);
  } else {
    assert(op == MemoryOperation::Fetch, "Unexpected memory operation");
    raise(ExceptionCode::InstructionPageFault, effective_mode, vaddr);
  }
}
if (translation_mode == SatpMode::Sv32) {
  result = stage1_page_walk<32, 34, 32, 2>(vaddr, op, effective_mode,
encoding);
} else if (translation_mode == SatpMode::Sv39) {
  result = stage1_page_walk<39, 56, 64, 3>(vaddr, op, effective_mode,
encoding);
} else if (translation_mode == SatpMode::Sv48) {
  result = stage1_page_walk<48, 56, 64, 4>(vaddr, op, effective_mode,
encoding);
} else if (translation_mode == SatpMode::Sv57) {
  result = stage1_page_walk<57, 56, 64, 5>(vaddr, op, effective_mode,
encoding);
} else {
  assert(false, "Unexpected SatpMode");
}
maybe_cache_translation(vaddr, op, result);
return result;
```

5.33. unpredictable (builtin)

Indicate that the hart has reached a state that is unpredictable because the RISC-V spec allows multiple behaviors. Generally, this will be a fatal condition to any emulation, since it is unclear what to do next.

The single argument *why* is a string describing why the hart entered an unpredictable state.

Return Type	`void`
Arguments	`String why`

5.34. write_memory

Write to virtual memory

Return Type	void
Arguments	XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
  write_memory_aligned<LEN>(virtual_address, value, encoding);
  return ;
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
  assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't
mix low-priority misaligned exceptions with large atomicity granule");
  physical_address = (CSR[misa].S == 1) ? translate(virtual_address,
MemoryOperation::Write, effective_ldst_mode(), encoding).paddr :
virtual_address;
  if (misaligned_is_atomic?<LEN>(physical_address)) {
    access_check(physical_address, LEN, virtual_address, MemoryOperation::
Write, ExceptionCode::StoreAmoAccessFault, effective_ldst_mode());
    write_physical_memory<LEN>(physical_address, value);
    return ;
  }
}
if (!MISALIGNED_LDST) {
  if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
    physical_address = (CSR[misa].S == 1) ? translate(virtual_address,
MemoryOperation::Write, effective_ldst_mode(), encoding).paddr :
virtual_address;
    access_check(physical_address, LEN, virtual_address, MemoryOperation::
Write, ExceptionCode::StoreAmoAccessFault, effective_ldst_mode());
  }
  raise(ExceptionCode::StoreAmoAddressMisaligned, effective_ldst_mode(),
virtual_address);
} else {
  if (MISALIGNED_SPLIT_STRATEGY == "by_byte") {
    for (U32 i = 0; i <= (LEN / 8); i++) {
      write_memory_aligned<8>(virtual_address + i, (value >> (8 * i))[7:0],
encoding);
    }
  } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any
way, leading to unpredictable behavior when any part of the misaligned access
causes an exception");
  }
}
```

5.35. write_memory_aligned

Write to virtual memory using a known aligned address.

Return Type	void
Arguments	XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding

```
XReg physical_address;
physical_address = (CSR[misa].S == 1) ? translate(virtual_address,
MemoryOperation::Write, effective_ldst_mode(), encoding).paddr :
virtual_address;
access_check(physical_address, LEN, virtual_address, MemoryOperation::Write,
ExceptionCode::StoreAmoAccessFault, effective_ldst_mode());
write_physical_memory<LEN>(physical_address, value);
```

5.36. write_memory_xlen

Read XLEN bits from memory

Return Type	void
Arguments	XReg virtual_address, Bits<XLEN> value, Bits<INSTR_ENC_SIZE> encoding

```
if (xlen() == 32) {
  return write_memory<32>(virtual_address, value, encoding);
} else {
  return write_memory<64>(virtual_address, value, encoding);
}
```

5.37. write_physical_memory

Write to physical memory.

Return Type	void
Arguments	XReg paddr, Bits<len> value

```
if (len == 8) {
  write_physical_memory_8(paddr, value);
} else if (len == 16) {
  write_physical_memory_16(paddr, value);
} else if (len == 32) {
  write_physical_memory_32(paddr, value);
} else if (len == 64) {
  write_physical_memory_64(paddr, value);
```

```
} else {
  assert(false, "Invalid len");
}
```

5.38. write_physical_memory_32 (builtin)

Write four bytes to physical memory.

Return Type	void
Arguments	XReg paddr, Bits<32> value

5.39. write_physical_memory_8 (builtin)

Write a byte to physical memory.

Return Type	void
Arguments	XReg paddr, Bits<8> value

5.40. xlen

Returns the effective XLEN for the current privilege mode.

Return Type	Bits<8>
Arguments	

```
if (XLEN == 32) {
  return 32;
} else {
  if (mode() == PrivilegeMode::M) {
    if (CSR[misa].MXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (CSR[misa].MXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    }
  } else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::S) {
    if (CSR[mstatus].SXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (CSR[mstatus].SXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    }
  } else if (implemented?(ExtensionName::U) && mode() == PrivilegeMode::U) {
    if (CSR[mstatus].UXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (CSR[mstatus].UXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    }
```

```
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) {
    if (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    }
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VU) {
    if (CSR[vsstatus].UXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (CSR[vsstatus].UXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    }
  }
}
```