Qualcomm Custom Compressed Instruction Lookup Table

Version 0.2.0, 2026-05-24

Table of Contents

Licensing and Acknowledgements

This document is in the Development state.

Change should be expected

Copyright and license information

This document was created using the RISC-V Unified Database at commit 1a8254db3c94e97b09984d44fa2e4af7379af94e.

Acknowledgements

Contributors to version 0.1.0 of the Xqccmi specification (in alphabetical order) include:

- Derek Hower <dhower@qti.qualcomm.com> (Qualcomm Technologies, Inc.)
- Albert Yosher <ayosher@qti.qualcomm.com> (Qualcomm Technologies, Inc.)
- Gil Zukerman <gzukerma@qti.qualcomm.com> (Qualcomm Technologies, Inc.)

Contributors to version 0.2.0 of the Xqccmi specification (in alphabetical order) include:

- Derek Hower <dhower@qti.qualcomm.com> (Qualcomm Technologies, Inc.)
- Albert Yosher <ayosher@qti.qualcomm.com> (Qualcomm Technologies, Inc.)
- Gil Zukerman <gzukerma@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 the following extension versions:

- Xqccmi@0.1.0
- Xqccmi@0.2.0

Version History

		Xqccmi
0.2.0	**State**	development
	Changes	• Precise trap semantics: each sub-instruction in an ILUT entry can now trap independently. Traps are taken immediately at the faulting sub-instruction rather than being deferred. • Resume mechanism: bit 0 of mepc/mnepc encodes which sub-instruction caused the trap. On trap return, qc.cm.ilut re-enters with PC bit 0 reflecting this value; if set, the first sub-instruction is skipped and only the second is re-executed. • mepc/mnepc bit 0 is readable and writable in this architecture. Software may set bit 0 to 1 to force resumption from the second sub-instruction. • Expanded ILUT instruction restrictions to all PC-referencing instructions: added register-indirect jumps (jalr, c.jr, c.jalr) and return instructions (cm.popret, cm.popretz, qc.cm.popret, qc.cm.popretz) to the forbidden list. • Spec language audit: removed all implementation-implying language; descriptions now define architectural behavior only.
0.1.0	**State**	development
	Changes	• Initial version. Custom instruction lookup table extension with dual-instruction packing and 64-bit double entries. Supports 16-bit, 32-bit, and 48-bit instructions in the ILUT. Introduces qc.itba and qc.itdec CSRs.

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

2.1. Xqccmi: Qualcomm Custom Compressed Instruction Lookup Table

The Xqccmi extension adds the `qc.cm.ilut` instruction, a 16-bit compressed instruction that fetches and executes one or two packed instructions from an Instruction Lookup Table (ILUT) stored in memory. The extension also defines two new user-level CSRs: `qc.itba` (ILUT Base Address) and `qc.itdec` (ILUT Double Entry Count).

Xqccmi uses the same encoding space as the `c.fld` instruction defined by the Zcd extension (bits[1:0]=00, bits[15:13]=001). Therefore, Xqccmi and Zcd are mutually exclusive and cannot be implemented together.

Instruction Lookup Table (ILUT)

`qc.cm.ilut` is a 16-bit instruction that takes an 11-bit immediate index (range 0..2047) into the ILUT. The ILUT base address is held in the `qc.itba` CSR (address 0x800). The instruction architecturally behaves as the sub-instruction(s) stored at the addressed ILUT entry, executed in order.

ILUT memory is treated as instruction memory. It requires execute (X) permission and does NOT require read ® permission. A `fence.i` instruction must be executed after any write to ILUT memory to guarantee that subsequent `qc.cm.ilut` executions observe the updated contents.

qc.itba CSR (Address 0x800)

The `qc.itba` CSR holds the base address of the ILUT.

Bits	Name	Access	Description
31:6	base	WARL (RW)	64-byte aligned base address of the ILUT. Bits[5:0] of the physical base address are implicitly zero.
5:0	mode	Read-only 0	Reserved for future use. Always reads as zero; writes are ignored.

`qc.itba` is a user-level read/write CSR. Software must save and restore `qc.itba` on context switches.

qc.itdec CSR (Address 0x801) — Double Entry Count

The `qc.itdec` CSR specifies how many of the leading ILUT entries are 64-bit double entries. All remaining entries are 32-bit.

Bits	Name	Access	Description
13:3	dec	WARL (RW)	11-bit double entry count. Legal range: 0..2047. Reset value: 0.
31:14	—	Read-only 0	Reserved. Always reads as zero; writes are ignored.

Bits	Name	Access	Description
2:0	—	Read-only 0	Reserved. Always reads as zero; writes are ignored.

`qc.itdec` is a user-level read/write CSR. The default reset value of 0 selects backward-compatible mode in which all entries are 32-bit.

Table Layout

The first DEC logical entries (indices 0 through DEC−1) are 64-bit double entries. The remaining entries (indices DEC through 2047) are 32-bit entries.

The byte offset from the ILUT base address for logical index i is:

- If i < DEC: offset = $i \times 8$
- If $i \geq$ DEC: offset = DEC × 8 + (i − DEC) × 4

The maximum table size is DEC×8 + (2048−DEC)×4 bytes. This ranges from approximately 8 KB (DEC=0, all 32-bit entries) up to approximately 16 KB (DEC=2047, nearly all 64-bit entries). No alignment constraint beyond the 64-byte base alignment imposed by `qc.itba` is required for individual entries.

Dual-Instruction Packing

Each ILUT entry may contain one or two instructions packed together. Instruction sizes are determined by the standard RISC-V instruction-length encoding (bits[1:0] of each sub-instruction).

32-bit Entry Valid Combinations

Combination	Description
One 32-bit instruction	bits[1:0] == 2'b11; the entire 32-bit entry is a single RVI instruction.
Two 16-bit instructions	bits[1:0] != 2'b11; the first 16-bit RVC instruction occupies bits[15:0] and the second occupies bits[31:16].

64-bit Entry Valid Combinations

Combination	Description
16 + 32 bits	First instruction is 16-bit (bits[1:0] != 2'b11); second instruction is 32-bit.
16 + 48 bits	First instruction is 16-bit; second instruction is 48-bit.
32 + 16 bits	First instruction is 32-bit (bits[1:0] == 2'b11); second instruction is 16-bit occupying bits[47:32].
32 + 32 bits	Both instructions are 32-bit.
Single 48-bit instruction	A single 48-bit instruction occupying bits[47:0]; bits[63:48] must be padded with `c.nop` (16'h0001).

A single 32-bit instruction placed in a 64-bit entry is valid as the 32+16 case with `c.nop` (16'h0001)

padding in bits[63:48].

Unused bits in any entry MUST be padded with `c.nop` (16'h0001). Instruction combinations that do not fit within the entry size cause an illegal instruction exception. Instruction sizes are determined by the standard RISC-V instruction-length encoding (bits[1:0] of each sub-instruction).

Instruction Restrictions

Any instruction that references PC as an input or output is NOT permitted in the ILUT. If any such instruction is present in the entry, an IllegalInstruction exception is raised before any sub-instruction in the entry takes architectural effect.

Forbidden instructions fall into three categories:

- **PC-relative instructions** (read PC as input):
  - `auipc`
  - All branch instructions (`beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`, `c.beqz`, `c.bnez`, etc.)
  - PC-relative jump instructions (`jal`, `c.j`, `c.jal`)
- **Register-indirect jumps** (write PC as output):
  - `jalr`
  - `c.jr`
  - `c.jalr`
- **Return instructions** (write PC as output):
  - `cm.popret`
  - `cm.popretz`
  - `qc.cm.popret`
  - `qc.cm.popretz`

The following push/pop instructions are explicitly **allowed** in ILUT entries because they do not reference PC: `cm.push`, `cm.pop`, `qc.cm.push`, `qc.cm.pushfp`, `qc.cm.pop`.

16-bit compressed instructions (bits[1:0] != 2'b11) are otherwise valid in ILUT entries.

Execution Model

The PC points to the `qc.cm.ilut` instruction throughout the execution of all sub-instructions contained within the fetched ILUT entry. Sub-instructions execute sequentially and independently:

- The architectural state changes produced by the first sub-instruction are visible to the second sub-instruction and to any subsequent trap handler.
- Each sub-instruction can trap independently; traps are taken precisely at the sub-instruction that caused them, not deferred.
- After the final sub-instruction completes, the PC advances past `qc.cm.ilut` normally.

Exception Handling

On any trap (exception, interrupt, NMI, or double-trap) occurring during the execution of a sub-instruction fetched from the ILUT:

- mepc (or mnepc for a double-trap) is set to the PC of `qc.cm.ilut`.
- Bit 0 of mepc/mnepc encodes which sub-instruction was executing when the trap occurred:
  - Bit 0 = 0: the trap occurred during the first (or only) sub-instruction.
  - Bit 0 = 1: the trap occurred during the second sub-instruction (the architectural effects of the first sub-instruction are already visible).

On trap return (`mret`, `qc.c.mret`, `qc.c.mnret`, `qc.c.mileaveret`), $pc is restored directly from mepc/mnepc including bit 0:

- If bit 0 = 0: `qc.cm.ilut` re-executes from the beginning, running both sub-instructions.
- If bit 0 = 1: `qc.cm.ilut` skips the first sub-instruction and executes only the second.

Bit 0 of mepc/mnepc is readable and writable in this architecture. Software may explicitly set bit 0 to 1 to force resumption from the second sub-instruction.

This is an architectural exception to the standard RISC-V requirement that *epc bit 0 is always zero. The non-zero bit 0 encodes which sub-instruction within the ILUT entry was interrupted, enabling precise trap restart.

Encoding

`qc.cm.ilut` uses the same encoding as `c.fld` (bits[1:0]=00, bits[15:13]=001). Xqccmi is therefore mutually exclusive with the Zcd extension.

Bits[15:13]	Bits[12:2]	Bits[1:0]
001	ilut_index[10:0]	00

The 11-bit `ilut_index` field (bits[12:2]) encodes the logical index into the ILUT, selecting one of up to 2048 entries.

Instruction Summary

RV32	RV64	Mnemonic	Instruction
yes	no	qc.cm.ilut *index*	[insns-qc_cm_ilut]

Chapter 3. Instruction list

Chapter 4. CSR summary

The following 2 CSRs are affected by this extension.

RV32	RV64	CSR	Name	v0.1.0	v0.2.0
✔	✔	qc.itba	Instruction Table Base Address Register	✔	✔
✔	✔	qc.itdec	Instruction Table Double Entry Count Register	✔	✔

Chapter 5. Csrs (in alphabetical order)

5.1. qc.itba

Instruction Table Base Address Register

The `qc.itba` register holds the base address of the Instruction Lookup Table (ILUT) used by the `qc.cm.ilut` instruction.

The BASE field specifies bits[XLEN-1:6] of the ILUT base address. The lower 6 bits are implicitly zero, so the ILUT base address is always 64-byte aligned.

The memory region pointed to by `qc.itba.base` is treated as instruction memory for the purpose of executing ILUT instructions, requiring execute (X) access permission. Read ® permission is not required.

`qc.itba` adds architectural state to the system software context and must be saved and restored on context switches.

5.1.1. Attributes

Requirement	`Xqccmi`	
Defining extensions	**Xqccmi**	Qualcomm Custom Compressed Instruction Lookup Table
CSR Address	0x800	
Length	32-bit	
Privilege Mode	U	

5.1.2. Format

Listing 1. qc.itba format

```
Failed to generate image: Could not find the 'wavedrom-cli' executable in PATH; add it to the PATH or specify its location using the 'wavedrom' document attribute
{"reg":[{"bits":6,"name":"mode","type":3},{"bits":58,"name":"base","type":3},{"bits":-32,"type":1}],"config":{"bits":32,"lanes":2}}
```

5.1.3. Field Summary

Name	Location	Type	Reset Value
qc.itba.base	31:6	RW	UNDEFINED_LEGAL
qc.itba.mode	5:0	RO	0

5.1.4. Fields

base

Location

31:6

Description

Bits[XLEN-1:6] of the ILUT base address. The full base address is formed by appending 6 implicit zero bits: `base_addr = {base, 6'b000000}`. The base address must be 64-byte aligned.

Type

RW

Reset value

UNDEFINED_LEGAL

mode

Location

5:0

Description

Reserved for future use. Always reads as zero. Writes are ignored.

Type

RO

Reset value

0

5.2. qc.itdec

Instruction Table Double Entry Count Register

The `qc.itdec` register specifies the number of leading 64-bit double entries in the Instruction Lookup Table (ILUT) used by the `qc.cm.ilut` instruction.

The DEC field holds an 11-bit count. The first DEC logical ILUT entries (indices 0 through DEC−1) are 64-bit double entries. The remaining entries (indices DEC through 2047) are 32-bit single entries.

The byte offset from the ILUT base address for logical index i is: - If i < DEC: offset = i × 8 - If i ≥ DEC: offset = DEC × 8 + (i − DEC) × 4

The reset value of 0 selects backward-compatible mode in which all entries are 32-bit single entries.

5.2.1. Attributes

Requirement	`Xqccmi`	
Defining extensions	**Xqccmi**	Qualcomm Custom Compressed Instruction Lookup Table
CSR Address	0x801	
Length	32-bit	
Privilege Mode	U	

5.2.2. Format

Listing 2. qc.itdec format

```
Failed to generate image: Could not find the 'wavedrom-cli' executable in PATH; add it to the PATH or specify its location using the 'wavedrom' document attribute
{"reg":[{"bits":3,"type":1},{"bits":11,"name":"dec","type":3},{"bits":18,"type":1}],"config":{"bits":32,"lanes":2}}
```

5.2.3. Field Summary

Name	Location	Type	Reset Value
qc.itdec.dec	13:3	RW	0

5.2.4. Fields

`dec`

Location

13:3

Description

11-bit double entry count. Specifies the number of leading 64-bit entries in the ILUT. Legal range: 0..2047. A value of 0 means all entries are 32-bit (backward-compatible mode).

Type

RW

Reset value

0

Chapter 6. Instructions (in alphabetical order)

6.1. qc.cm.ilut

Synopsis

Instruction Lookup Table Execute

Mnemonic

```
qc.cm.ilut ilut_inder
```

Encoding

```
Failed to generate image: Could not find the 'wavedrom-cli' executable in PATH; add it
to the PATH or specify its location using the 'wavedrom' document attribute
{"reg":[{"bits":2,"name": 0x0,"type":2},{"bits":11,"name":
"ilut_index","type":4},{"bits":3,"name": 0x1,"type":2}]}
```

Description

Fetch and execute the instruction(s) packed in the ILUT entry at index `ilut_index`.

The ILUT base address is held in `qc.itba.base` (shifted left by 6 to form a 64-byte-aligned byte address). The first `qc.itdec.dec` entries are 64-bit double entries; all remaining entries are 32-bit single entries.

Entry address calculation: - If `ilut_index` < `qc.itdec.dec`: `entry_addr = (qc.itba.BASE << 6) + ilut_index * 8` - Otherwise: `entry_addr = (qc.itba.BASE << 6) + qc.itdec.DEC * 8 + (ilut_index - qc.itdec.DEC) * 4`

Each entry may contain one or two packed RISC-V instructions. The first instruction begins at bit 0 of the entry. Its size is determined by the standard RISC-V instruction-length encoding (16-bit, 32-bit, or 48-bit). If bits remain after the first instruction and the next 16 bits are not `c.nop` (16'h0001), a second instruction is present and is also executed.

Any instruction that references PC as an input or output is not permitted inside ILUT entries. This includes: PC-relative instructions (`auipc`, branches, `jal`, `c.j`, `c.jal`, `c.beqz`, `c.bnez`); register-indirect jumps that write PC (`jalr`, `c.jr`, `c.jalr`); and return instructions that write PC (`cm.popret`, `cm.popretz`, `qc.cm.popret`, `qc.cm.popretz`). If any such instruction is present in the entry, an IllegalInstruction exception is raised before any sub-instruction in the entry takes architectural effect.

When a trap (exception, interrupt, NMI, or double-trap) occurs during execution of a `qc.cm.ilut` entry, `mepc`/`mnepc` is set to the PC of `qc.cm.ilut` with bit 0 encoding which sub-instruction caused the trap: bit 0 = 0 means the trap came from the first (or only) instruction; bit 0 = 1 means the trap came from the second instruction (the architectural effects of the first sub-instruction are already

visible). When qc.cm.ilut is re-entered with PC bit 0 = 1, it skips the first instruction and executes only the second. This applies uniformly to all trap types including exceptions, interrupts, NMIs, and double-traps.

See the Xqccmi extension description for full details on entry layout, dual-instruction packing, exception handling, and restrictions.

Decode Variables

```
Bits<11> ilut_index = $encoding[12:2];
```

Operation

```
XReg real_pc = $pc & ~32'b1;
Boolean resuming_second = ($pc[0:0] == 1'b1);
XReg N = {21'b0, CSR[qc.itdec].dec};
XReg base_addr = CSR[qc.itba].base << 6;
XReg entry_addr;
Boolean is_double;
if (ilut_index < N) {
  entry_addr = base_addr + (ilut_index * 8);
  is_double = true;
} else {
  entry_addr = base_addr + (N * 8) + ilut_index - N) * 4);   is_double = false; }
TranslationResult result; if (CSR[misa].S == 1) {   result = translate(entry_addr,
MemoryOperation::Fetch, mode(), $encoding); } else {   result.paddr = entry_addr; }
access_check(result.paddr, is_double ? 8 : 4, $pc, MemoryOperation::Fetch,
ExceptionCode::InstructionAccessFault, mode(;
XReg entry_lo = read_physical_memory(32, result.paddr);
XReg entry_hi = is_double ? read_physical_memory(32, result.paddr + 4) : 32'b0;
Boolean has_second_inst = false;
if (entry_lo[1:0] != 2'b11) {
  if (entry_lo[31:16] != 16'h0001) {
    has_second_inst = true;
  }
} else if (entry_lo[4:2] != 3'b111) {
  if (is_double && entry_hi[15:0] != 16'h0001) {
    has_second_inst = true;
  }
} else {
  if (!is_double) {
    raise_ilut(ExceptionCode::IllegalInstruction, $encoding);
  }
}
if (is_pc_referencing_inst?(entry_lo)) {
  executing_second_inst = false;
  raise_ilut(ExceptionCode::IllegalInstruction, $encoding);
}
if (has_second_inst) {
```

```
  Bits<32> second_inst = (entry_lo[1:0] != 2'b11) ? {16'b0, entry_lo[31:16]} : {16'b0,
entry_hi[15:0]};
  if (is_pc_referencing_inst?(second_inst)) {
    executing_second_inst = true;
    raise_ilut(ExceptionCode::IllegalInstruction, $encoding);
  }
}
if (!resuming_second) {
  if (implemented?(ExtensionName::Smrnmi) && CSR[mnstatus].NMIE == 1'b0) {
    CSR[mnepc].PC = real_pc;
  } else {
    CSR[mepc].PC = real_pc;
  }
  execute_instruction(entry_lo);
}
if (has_second_inst) {
  Bits<32> second_inst = (entry_lo[1:0] != 2'b11) ? {16'b0, entry_lo[31:16]} : {16'b0,
entry_hi[15:0]};
  if (implemented?(ExtensionName::Smrnmi) && CSR[mnstatus].NMIE == 1'b0) {
    CSR[mnepc].PC = real_pc | 32'b1;
  } else {
    CSR[mepc].PC = real_pc | 32'b1;
  }
  execute_instruction(second_inst);
  if (implemented?(ExtensionName::Smrnmi) && CSR[mnstatus].NMIE == 1'b0) {
    CSR[mnepc].PC = real_pc;
  } else {
    CSR[mepc].PC = real_pc;
  }
}
```

Included in

`qc.cm.ilut` is unconditionally defined by the following:

Extension	Version
Xqccmi	any

Chapter 7. IDL Functions

7.1. abort_current_instruction (builtin)

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

Return Type	`void`
Arguments	None

7.2. access_check

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

Return Type	`void`
Arguments	

Idx	Type	Name
0	`Bits<PHYS_ADDR_WIDTH>`	paddr
1	`U32`	access_size
2	`XReg`	vaddr
3	`MemoryOperation`	type
4	`ExceptionCode`	fault_type
5	`PrivilegeMode`	from_mode

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

7.3. assert (builtin)

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

Return Type	`void`
Arguments	

Idx	Type	Name
0	`Boolean`	test
1	`String`	message

7.4. csr_hw_read (generated)

Returns the raw value of csr

Return Type	Bits<64>		
Arguments	**Idx**	**Type**	**Name**
	0	Csr	csr

7.5. csr_sw_read (generated)

Returns the result of CSR[csr].sw_read(); i.e., the software view of the register

Return Type	Bits<64>		
Arguments	**Idx**	**Type**	**Name**
	0	Csr	csr

7.6. direct_csr_lookup (generated)

Return CSR info for a CSR with direct address csr_addr.

If no CSR exists, <return_value>.valid == false

Return Type	Csr		
Arguments	**Idx**	**Type**	**Name**
	0	Bits<12>	csr_addr

7.7. 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	None

```
if (mode() == PrivilegeMode::M) {
  if (CSR[misa].U == 1 && CSR[mstatus].MPRV == 1) {
    if (CSR[mstatus].MPP == 0b00) {
      if (implemented?(ExtensionName::H) && CSR[misa].H == 1 && mpv() == 0b1) {
        return PrivilegeMode::VU;
      } else {
        return PrivilegeMode::U;
      }
```

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

7.8. exception_handling_mode

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

Return Type	`PrivilegeMode`		
Arguments	**Idx**	**Type**	**Name**
	0	`ExceptionCode`	exception_code

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

7.9. execute_instruction (builtin)

Fetch and execute the instruction with the given 32-bit encoding in the current hart context. The

instruction executes as if it were a normal instruction, with the following differences:

- The PC is not advanced after execution (the calling instruction is responsible for PC management).
- If the instruction raises an exception, the exception propagates normally.

This builtin is intended for use by instructions that dynamically dispatch sub-instructions fetched from memory (e.g., instruction lookup table instructions).

Return Type	`void`		
Arguments	**Idx**	**Type**	**Name**
	0	`Bits<32>`	encoding

7.10. implemented? (generated)

Return true if the implementation supports `extension`.

Return Type	`Boolean`		
Arguments	**Idx**	**Type**	**Name**
	0	`ExtensionName`	extension

7.11. implemented_version? (generated)

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

Return Type	`Boolean`		
Arguments	**Idx**	**Type**	**Name**
	0	`ExtensionName`	extension
	1	`String`	version_requirement

7.12. is_pc_referencing_inst?

Returns true if inst references PC as an input or output, making it forbidden inside an ILUT entry. This covers three categories: (1) PC-relative instructions: auipc, branches, jal, c.j, c.jal, c.beqz, c.bnez; (2) register-indirect jumps that write PC: jalr, c.jr, c.jalr; (3) return instructions that write PC: cm.popret, cm.popretz, qc.cm.popret, qc.cm.popretz. For 16-bit compressed instructions, only the lower 16 bits of inst are examined. For 48-bit Xqci instructions, only the lower 32 bits are passed here (the upper 16 bits are in a separate word and are not needed for this check).

Return Type	`Boolean`

Arguments	Idx	Type	Name
	0	Bits<32>	inst

```
if (inst[1:0] != 2'b11) {
  Bits<16> cinst = inst[15:0];
  if (cinst[15:13] == 3'b101 && cinst[1:0] == 2'b01) {
    return true;
  }
  if (cinst[15:13] == 3'b001 && cinst[1:0] == 2'b01) {
    return true;
  }
  if (cinst[15:13] == 3'b110 && cinst[1:0] == 2'b01) {
    return true;
  }
  if (cinst[15:13] == 3'b111 && cinst[1:0] == 2'b01) {
    return true;
  }
  if (cinst[15:12] == 4'b1000 && cinst[6:2] != 5'b00000 && cinst[1:0] == 2'b10) {
    return true;
  }
  if (cinst[15:12] == 4'b1001 && cinst[6:2] != 5'b00000 && cinst[1:0] == 2'b10) {
    return true;
  }
  if (cinst[15:8] == 8'b10111110 && cinst[1:0] == 2'b10) {
    return true;
  }
  if (cinst[15:8] == 8'b10111100 && cinst[1:0] == 2'b10) {
    return true;
  }
  return false;
} else {
  Bits<7> opcode = inst[6:0];
  if (opcode == 7'b0010111) {
    return true;
  }
  if (opcode == 7'b1101111) {
    return true;
  }
  if (opcode == 7'b1100111) {
    return true;
  }
  if (opcode == 7'b1100011) {
    return true;
  }
  if (inst[4:2] == 3'b111) {
    if (opcode == 7'b0011111) {
      if (inst[14:12] == 3'b000 && inst[24:20] == 5'b00000) {
        return true;
      }
```

```
      if (inst[14:12] == 3'b100 && inst[24:23] == 2'b11) {
        return true;
      }
    }
  }
  return false;
}
```

7.13. mode

Returns the current active privilege mode.

Return Type	PrivilegeMode
Arguments	None

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

7.14. 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	**Idx**	**Type**	**Name**
	0	ExceptionCode	exception_code
	1	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;
}
```

7.15. mtval_readonly?

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

Return Type	Boolean
Arguments	None

```
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);
```

7.16. notify_mode_change (builtin)

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

Return Type	void

Arguments

Idx	Type	Name
0	PrivilegeMode	new_mode
1	PrivilegeMode	old_mode

7.17. pmp_check

Given a physical address and operation type, return whether or not the access is allowed by PMP.

Return Type	Boolean

Arguments

Idx	Type	Name
0	Bits<PHYS_ADDR_WIDTH>	paddr
1	U32	access_size
2	MemoryOperation	type

```
PrivilegeMode mode = effective_ldst_mode();
PmpMatchResult match_result;
PmpCfg cfg;
(match_result, cfg = pmp_match(paddr, access_size));
if (match_result == PmpMatchResult::FullMatch) {
  if (mode == PrivilegeMode::M && (cfg.L == 0)) {
    return true;
  }
  if (type == MemoryOperation::Write && (cfg.W == 0)) {
    return false;
  } else if (type == MemoryOperation::Read && (cfg.R == 0)) {
    return false;
  } else if (type == MemoryOperation::Fetch && (cfg.X == 0)) {
    return false;
  }
} else if (match_result == PmpMatchResult::NoMatch) {
  if (mode == PrivilegeMode::M) {
    return true;
  } else {
    return false;
  }
} else {
  assert(match_result == PmpMatchResult::PartialMatch, "PMP matching logic error");
  return false;
}
return true;
```

7.18. pmp_match

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	PmpMatchResult, PmpCfg

Arguments	Idx	Type	Name
	0	Bits<PHYS_ADDR_WIDTH>	paddr
	1	U32	access_size

```
if (MXLEN == 64) {
  return pmp_match_64(paddr, access_size);
} else {
  return pmp_match_32(paddr, access_size);
}
```

7.19. pmp_match_32

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	PmpMatchResult, PmpCfg		
Arguments	Idx	Type	Name
	0	Bits<PHYS_ADDR_WIDTH>	paddr
	1	U32	access_size

```
Bits<12> pmpcfg0_addr = 0x3a0;
Bits<12> pmpaddr0_addr = 0x3b0;
for (U32 i = 0; i < NUM_PMP_ENTRIES; i++) {
  Bits<12> pmpcfg_idx = pmpcfg0_addr + (i / 4);
  Bits<6> shamt = (i % 4) * 8;
  Csr pmpcfg_csr = direct_csr_lookup(pmpcfg_idx);
  PmpCfg cfg = (csr_hw_read(pmpcfg_csr) >> shamt)[7:0];
  Bits<12> pmpaddr_idx = pmpaddr0_addr + i;
  Csr pmpaddr_csr = direct_csr_lookup(pmpaddr_idx);
  Bits<32> pmpaddr_csr_value = csr_sw_read(pmpaddr_csr);
  Bits<PHYS_ADDR_WIDTH> range_base = 0;
  Bits<PHYS_ADDR_WIDTH> range_limit = 0;
  if (cfg.A == $bits(PmpCfg_A::TOR)) {
    if (i == 0) {
      range_base = 0;
    } else {
      Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);
      range_base = csr_sw_read(tor_pmpaddr_csr)[PHYS_ADDR_WIDTH - 1:0];
    }
    range_limit = (pmpaddr_csr_value)[PHYS_ADDR_WIDTH - 1:0] - 1;
  } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {
    Bits<PHYS_ADDR_WIDTH - 1> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 3:
0];
```

```
    Bits<PHYS_ADDR_WIDTH - 1> mask = pmpaddr_value ^ (pmpaddr_value + 1);
    range_base = pmpaddr_value & ~mask;
    range_limit = range_base + mask;
  } else if (cfg.A == $bits(PmpCfg_A::NA4)) {
    range_base = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 1:0];
    range_limit = range_base + 3;
  }
  if (paddr {
    return PmpMatchResult::FullMatch, cfg;
  } else if (! {
    return PmpMatchResult::PartialMatch, -;
  }
}
return PmpMatchResult::NoMatch, -;
```

7.20. 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	**Idx**	**Type**	**Name**
	0	`ExceptionCode`	exception_code
	1	`PrivilegeMode`	from_mode
	2	`XReg`	tval

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

7.21. raise_ilut

Raise a precise exception for an ILUT-executed instruction. Sets mepc.PC = $pc | (executing_second_inst ? 1 : 0) so that the trap handler can determine whether the fault came from the first or second packed instruction in the ILUT entry. For a double-trap, sets mnepc.PC instead. Only the M-mode handling path is included because qc_iu is M-mode only.

Return Type	`void`

Arguments	Idx	Type	Name
	0	ExceptionCode	exception_code
	1	XReg	tval

```
Bits<1> slot_bit = executing_second_inst ? 1'b1 : 1'b0;
if (implemented?(ExtensionName::Smrnmi) && CSR[mnstatus].NMIE == 1'b0) {
  CSR[mnepc].PC = $pc | slot_bit;
} else {
  CSR[mepc].PC = $pc | slot_bit;
}
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);
set_mode(PrivilegeMode::M);
abort_current_instruction();
```

7.22. raise_precise

Raise synchronous exception number exception_code.

Return Type	void		
Arguments	Idx	Type	Name
	0	ExceptionCode	exception_code
	1	PrivilegeMode	from_mode
	2	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 (implemented?(ExtensionName::H) && CSR[misa].H == 1) {
    CSR[mtval2].VALUE = 0;
    CSR[mtinst].VALUE = 0;
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      if (MXLEN == 32) {
        CSR[mstatush].MPV = 1;
```

```
      } else {
        CSR[mstatus].MPV = 1;
      }
    } else {
      if (MXLEN == 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();
```

7.23. read_physical_memory

Read from physical memory.

Return Type	XReg		
Arguments	**Idx**	**Type**	**Name**
	0	U32	len
	1	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");
}
```

7.24. read_physical_memory_32 (builtin)

Read four bytes from physical memory.

Return Type	Bits<32>		
Arguments	**Idx**	**Type**	**Name**
	0	XReg	paddr

7.25. 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	None

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

7.26. set_mode

Set the current privilege mode to new_mode

Return Type	void		
Arguments	**Idx**	**Type**	**Name**
	0	PrivilegeMode	new_mode

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