Qualcomm Custom Compressed Jump Table with Alternate Link Register

Version 0.1.0, 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 Xqccmt specification (in alphabetical order) include:

- Derek Hower <dhower@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 0.1.0 of Xqccmt.

Version History

		Xqccmt
0.1.0	**State**	development
	Changes	• Initial version. Custom variant of Zcmt with alternate link register support. Uses bit 0 of call table entries as metadata to select ra (x1) or t0 (x5) as the return address register for qc.cm.jalt.

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

Xqccmt is a Qualcomm custom extension that provides compressed (16-bit) jump table instructions for code-size reduction. It introduces two instructions, `qc.cm.jt` and `qc.cm.jalt`, which perform indirect jumps through a jump vector table (JVT) whose base address is held in the `jvt` CSR.

Jump Vector Table

The jump vector table is an array of XLEN-wide entries stored in memory. Its base address is configured by writing to the `jvt` CSR. The base address must be aligned to a 64-byte boundary (the lower 6 bits of the base are always zero). The memory region pointed to by `jvt.BASE` is treated as instruction memory for the purpose of execute-permission checks.

Each instruction encodes an 8-bit `index` field. The index selects which entry in the table to use:

- Indices 0–31 (bits [7:5] of index == 0b000): used by `qc.cm.jt` (jump only, no link)
- Indices 32–255 (bits [7:5] of index != 0b000): used by `qc.cm.jalt` (jump with link)

The entry address is computed as: `entry_addr = {jvt.BASE, 6'b000000} + index x (XLEN/8)`

Alternate Link Register

In standard table-jump behavior, a call instruction always saves the return address in `ra` (x1). Xqccmt extends this by repurposing bit 0 of each call table entry (index ≥ 32) as metadata to select the return address register:

- Entry bit 0 = 0: save return address in `ra` (x1)
- Entry bit 0 = 1: save return address in `t0` (x5), the alternate link register

Because XLEN-wide addresses are naturally aligned, bit 0 of a valid target address is always zero. Xqccmt repurposes this always-zero bit as a one-bit selector without affecting the jump target. The actual jump target is always computed with bit 0 forced to zero: `target = {entry[XLEN-1:1], 1'b0}`.

For `qc.cm.jt` (index < 32), bit 0 of the table entry is not used as metadata; the jump target is computed identically: `target = {entry[XLEN-1:1], 1'b0}`, and no link register is written.

Encoding

Both instructions use the 16-bit compressed encoding space (quadrant 2, funct3=0b101), the same space used by `cm.jt` and `cm.jalt` in the standard Zcmt extension. Because of this encoding overlap, Xqccmt and Zcmt are mutually exclusive and cannot both be present in the same configuration.

The 16-bit encoding is:

[15:13]	[12:10]	[9:5]	[4:2]	[1:0]
101	index[7:5]	index[4:0]	000	10

When index[7:5] == 0b000, the instruction is `qc.cm.jt` (indices 0–31). When index[7:5] != 0b000, the instruction is `qc.cm.jalt` (indices 32–255).

JVT CSR Access Control

The `jvt` CSR is accessible at address 0x017. If the Smstateen extension is implemented, access to `jvt` from non-M modes is gated by the JVT bit in the `mstateen0` / `sstateen0` / `hstateen0` CSRs. Attempting to execute `qc.cm.jt` or `qc.cm.jalt` when `jvt` is inaccessible raises an Illegal Instruction exception.

Instruction Summary

RV32	RV64	Mnemonic	Instruction
yes	yes	qc.cm.jt *index*	[insns-qc_cm_jt]
yes	yes	qc.cm.jalt *index*	[insns-qc_cm_jalt]

Chapter 3. Instruction list

The following 2 instructions must be implemented when Xqccmt is implemented:

Table 1. Instructions defined by Xqccmt

RV32	RV64	Mnemonic	Instruction
✔	✔	`qc.cm.jalt inder_7_5, inder_4_0`	Jump Via Table with Link to ra or t0
✔	✔	`qc.cm.jt inder`	Jump Via Table

Chapter 4. Instructions (in alphabetical order)

4.1. qc.cm.jalt

Synopsis

Jump Via Table with Link to ra or t0

Mnemonic

```
qc.cm.jalt inder_7_5, inder_4_0
```

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": 0x2,"type":2},{"bits":5,"name":
"index_4_0","type":4},{"bits":3,"name": "index_7_5 != 0","type":4},{"bits":6,"name":
0x28,"type":2}]}
```

Description

Read an address from the Jump Vector Table and jump to it, linking to either `ra` (x1) or `t0` (x5) depending on bit 0 of the table entry.

Bit 0 of the jump table entry is used as metadata:

- Bit 0 = 0: save return address in `ra` (x1)
- Bit 0 = 1: save return address in `t0` (x5)

The actual jump target always has bit 0 cleared: `target = {entry[XLEN-1:1], 1'b0}`

Instruction encoded in the same encoding space as cm.jalt (Zcmt). Xqccmt and Zcmt are mutually exclusive extensions.

Decode Variables

```
Bits<3> index_7_5 = $encoding[9:7];
Bits<5> index_4_0 = $encoding[6:2];
```

Operation

```
check_zcmt_enabled($encoding);
if (CSR[jvt].MODE != 0) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
```

```
}
XReg return_addr = $pc + 2;
XReg jump_table_base = {CSR[jvt].BASE, 6'b000000};
XReg index = {index_7_5, index_4_0};
XReg virtual_address = jump_table_base + index `* (xlen() / 8);
XReg entry;
TranslationResult result;
if (CSR[misa].S == 1) {
  result = translate(virtual_address, MemoryOperation::Fetch, mode(), $encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, xlen(), $pc, MemoryOperation::Fetch, ExceptionCode
::InstructionAccessFault, mode());
if (xlen() == 32) {
  entry = read_physical_memory(32, result.paddr);
} else {
  entry = read_physical_memory(64, result.paddr);
}
if (entry[0] == 1'b1) {
  X[5] = return_addr;
} else {
  X[1] = return_addr;
}
XReg addr = entry & $signed(2'b10);
jump(addr);
```

Included in

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

Extension	Version
Xqccmt	any

4.2. qc.cm.jt

Synopsis

Jump Via Table

Mnemonic

```
qc.cm.jt 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": 0x2,"type":2},{"bits":5,"name":
"index","type":4},{"bits":9,"name": 0x140,"type":2}]}
```

Description

Read an address from the Jump Vector Table and jump to it. Instruction encoded in the same encoding space as cm.jt (Zcmt). Xqccmt and Zcmt are mutually exclusive extensions.

Decode Variables

```
Bits<5> index = $encoding[6:2];
```

Operation

```
check_zcmt_enabled($encoding);
if (CSR[jvt].MODE != 0) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg jump_table_base = {CSR[jvt].BASE, 6'b000000};
XReg virtual_address = jump_table_base + index `* (xlen() / 8);
XReg addr;
TranslationResult result;
if (CSR[misa].S == 1) {
  result = translate(virtual_address, MemoryOperation::Fetch, mode(), $encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, xlen(), $pc, MemoryOperation::Fetch, ExceptionCode
::InstructionAccessFault, mode());
if (xlen() == 32) {
  addr = read_physical_memory(32, result.paddr);
} else {
  addr = read_physical_memory(64, result.paddr);
}
```

```
addr = addr & $signed(2'b10);
jump(addr);
```

Included in

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

Extension	Version
Xqccmt	any

Chapter 5. IDL Functions

5.1. abort_current_instruction (builtin)

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

Return Type	`void`
Arguments	None

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	

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);
  }
}
```

5.3. 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

5.4. check_zcmt_enabled

If the Smstateen extension is implemented, then bit 2 in `mstateen0`, `sstateen0`, and `hstateen0` is implemented. If bit 2 of a controlling `stateen0` CSR is zero, then access to the `jvt` CSR and execution of a `cm.jalt` or `cm.jt` instruction by a lower privilege level results in an illegal-instruction trap (or, if appropriate, a virtual-instruction trap).

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

```
if ((mode() != PrivilegeMode::M && implemented?(ExtensionName::Smstateen) && CSR
[mstateen0].JVT == 1'b0) || (mode() == PrivilegeMode::U && implemented?(
ExtensionName::Ssstateen) && CSR[sstateen0].JVT == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
} else if ((mode() == PrivilegeMode::VS && implemented?(ExtensionName::Ssstateen) &&
CSR[hstateen0].JVT == 1'b0) || (mode() == PrivilegeMode::VU && implemented?
(ExtensionName::Ssstateen) && (CSR[sstateen0].JVT == 1'b0 || CSR[hstateen0].JVT ==
1'b0))) {
  raise(ExceptionCode::VirtualInstruction, mode(), encoding);
}
```

5.5. csr_hw_read (generated)

Returns the raw value of csr

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

5.6. 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

5.7. 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

5.8. 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();
```

5.9. 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();
}
```

5.10. ialign

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

Return Type	`Bits<6>`
Arguments	None

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

5.11. implemented? (generated)

Return true if the implementation supports `extension`.

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

5.12. 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

5.13. jump

Jump to virtual address `target_addr`.

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

Return Type	`void`		
Arguments	**Idx**	**Type**	**Name**
	0	`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;
```

5.14. 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;
}
```

5.15. 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;
}
```

5.16. 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);
```

5.17. 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

5.18. 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;
```

5.19. 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);
}
```

5.20. 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, -;
```

5.21. 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);
}
```

5.22. 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();
```

5.23. 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");
}
```

5.24. read_physical_memory_32 (builtin)

Read four bytes from physical memory.

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

5.25. 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;
```

5.26. 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();
}
```

5.27. xlen

Returns the effective XLEN for the current privilege mode.

Return Type	Bits<8>
Arguments	None

```
if (MXLEN == 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 {
      unreachable();
    }
  } 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 {
      unreachable();
    }
  } 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 {
      unreachable();
    }
  } 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 {
      unreachable();
    }
```

```
  } 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;
    } else {
      unreachable();
    }
  } else {
    unreachable();
  }
}
```