"Smclic" Core-Local Interrupt Controller (CLIC) RISC-V Privileged Architecture Extension

Version 0.9-draft, 12/21/2021: This document is in the Development state. Assume anything can change. See
https://wiki.riscv.org/display/HOME/Specification+States

Table of Contents

Preamble

Graphics used are either explicitly available for free, are property of RISC-V International, or were created using Wavedrom.

Chapter 1. Revision History

```
Date        Description
12/21/2021  issue #109 - add smclic arch string to spec
12/21/2021  issue #180 - change processor references to hart
11/09/2021  issue #48 - indicate when edge-triggered interrupts are cleared
11/09/2021  issue #179 - set interrupt bit during nxti access
10/28/2021  issue #154 - inhv clarification
10/28/2021  issue #31/#120 - wfi clarification
10/12/2021  issue #177 - Reduced mandatory reset requirements
09/29/2021  Added link to development states definition on top page
09/14/2021  pull #169 - nxti clarification
09/14/2021  pull #168 - only 0 or 8 level bits currently supported (other values
reserved)
09/14/2021  issue #170 - clarified position of intthresh in CSR
08/31/2021  issue #86/#165 - Update mnxti pseudo-code to handle side-effects
correctly.
08/31/2021  pull #164 - moved clicintattr.mode reset value to reset section of
spec
08/17/2021  pull #163 - spec clarification that clicintie is held in bit 0 of
byte.
07/20/2021  pull #161 - spec clarification that only writes to xnxti have side
effects.
07/06/2021  issue #156,#77,#79 - more CLIC memory mapped text clarifications,
clicintctl typo fixes
06/22/2021  issue #156 - reverted text and added clarification on CLIC memory
mapped privilege regions.
05/25/2021  issue #149 - added text that 32-bit writes are legal but effects are
not defined.
05/25/2021  issue #142 - added text that MPRV and SUM are obeyed on vector table
accesses.
05/11/2021  issue #154 - added text that clarifies behavior when inhv is set when
returning from a ret instruction.
04/27/2021  clicintip[i] state is undefined when switching from level to edge
triggered mode
04/22/2021  updated adoc format to align with risc-v template, added revision
history
04/18/2021  Added Bibliography section
04/15/2021  issue #45 - for rev1.0 mtvec not xtvec controls enabling CLIC mode for
all priv
04/13/2021  issue #141 - N-extension vs Bare S-mode note added.
04/13/2021  issue #117,#125 fix - change text to match table in M/S/U system if
nmbits==1
04/12/2021  issue #47 fix - add CLIC reset behavior section
04/12/2021  issue #26 fix - modify wording that defined micro-architectural
behavior of xINHV
```

04/12/2021   issue #91 - add DTS entry example
04/12/2021   added CLIC comparision to Advance Interupt Architecture (AIA)
04/12/2021   issue #111,#105 fix - For hardware vectoring access exceptions, both {tval} and {epc} holds the faulting address
04/08/2021   issue #49, #79 - downplay M/S/U memory map requirements
03/30/2021   issue #29 - updated memory map table reserved section to give room for clicinttrig
03/30/2021   issue #122 fix - remove wording referring to register
03/11/2021   issue #120 - update WFI wording
03/11/2021   typo fixing
03/11/2021   issue #51 - implementation of non CSRRW variants of xscratchcsw/xscratchcswl explicitly not defined/reserved.
03/11/2021   issue #58 - xintthresh was missing from table summarizing overall interrupt behavior
02/17/2021   issue #95 fix - removed N extension reference since not ratified.
02/17/2021   issue #90 fix - clarified that clicintip!=0 means interrupt pending
02/17/2021   issue #89 - updated CLIC interrupt ID ordering recommendations
02/17/2021   ihnv clarification - inhv bit has no effect except when returning from a trap using an {ret} instruction
02/17/2021   ihnv clarification - inhv only written by hw during table vector read. can be written by software.
02/02/2021   WFI wording change
01/19/2021   WFI wording change
01/07/2021   WFI section added
01/07/2021   Notes added clarifying clicintie and mstatus.xie
01/07/2021   interrupt priority clarification
12/17/2020   Added support for interrupt triggers
10/20/2020   clarified differences between level and priority
10/20/2020   fixed value range for CLICINTCTLBITS
10/20/2020   Clarified relationship among interrupt level, cliccfg.nlbits and CLICINTCTLBITS
09/08/2020   clarified description for interrupt level

Chapter 2. Background and Motivation

The "Smclic" Core-Local Interrupt Controller (CLIC) Privileged Architecture Extension is designed to provide low-latency, vectored, pre-emptive interrupts for RISC-V systems. When activated the CLIC subsumes and replaces the original RISC-V basic local interrupt scheme. The CLIC has a base design that requires minimal hardware, but supports additional extensions to provide hardware acceleration. The goal of the CLIC is to provide support for a variety of software ABI and interrupt models, without complex hardware that can impact high-performance implementations.

The CLIC also supports a new Selective Hardware Vectoring feature that allow users to optimize each interrupt for either faster response or smaller code size.

> While the current CLIC provides only hart-local interrupt control, future additions might also support directing interrupts to harts within a core, hence the name (also CLIC sounds better than HLIC or HIC).

2.1. Existing RISC-V Interrupts

The existing RISC-V interrupt system already supports interrupt preemption, but only based on privilege mode. At any point in time, a RISC-V hart is running with a current privilege mode. The global interrupt enable bits, MIE/SIE/UIE, held in the `mstatus`/`sstatus`/`ustatus` registers respectively, control whether interrupts can be taken for the current or higher privilege modes; interrupts are always disabled for lower-privileged modes. Any enabled interrupt from a higher-privilege mode will stop execution at the current privilege mode, and enter the handler at the higher privilege mode. Each privilege mode has its own interrupt state registers (`mepc`/`mcause` for M-mode, `sepc`/`scause` for S-mode, `uepc`/`ucause` for U-mode) to support preemption, or generically `xepc` for privilege mode x. Preemption by a higher-privilege-mode interrupt also pushes current privilege mode and interrupt enable status onto the `xpp` and `xpie` stacks in the `xstatus` register of the higher-privilege mode.

The `xtvec` register specifies both the interrupt mode and the base address of the interrupt vector table. The low bits of the WARL `xtvec` register indicate what interrupt model is supported. The original settings of `xtvec` mode (*00 and *01) indicate use of the original basic interrupt model with either non-vectored or vectored transfer to a handler function, with the 4-byte (or greater) aligned table base address held in the upper bits of `xtvec`.

> WARL means "Write Any, Read Legal" indicating that any value can be attempted to be written but only some supported values will actually be written.

> CLIC mode is enabled using previously reserved values (*11) in the low two bits of mtvec.

2.2. CLIC compared to PLIC

The standard RISC-V platform-level interrupt controller (PLIC) provides centralized interrupt prioritization and routing for shared platform-level interrupts, and sends only a single external interrupt signal per privilege mode (`meip`/`seip`/`ueip`) to each hart.

The CLIC complements the PLIC. Smaller single-core systems might have only a CLIC, while multicore systems might have a CLIC per-core and a single shared PLIC. The PLIC `xeip` signals are treated as hart-local interrupt sources by the CLIC at each core.

2.3. CLIC compared to Original Basic Interrupt Controller

The existing original basic interrupt controller was a small unit that provided local interrupts based on earlier designs, and managed the software, timer, and external interrupt signals (`xsip`/`xtip`/`xeip` signals in the `xip` register). This basic controller also allowed additional custom fast interrupt signals to be added in bits 16 and up of the `xip` register.

New settings of `xtvec` mode as described below are used to enable CLIC modes instead of the original basic interrupt modes. Platform profiles may require either or both of the original basic and CLIC interrupt modes.

2.4. CLIC compared to Advanced Interrupt Architecture

Advanced interrupt Architecture (AIA) supports message-signaled interrupts (MSIs) and an Advanced PLIC (APLIC) and targeted to support multiple harts, and support for virtualization. Like CLIC, the relative priority of all interrupts (not just external) can be configured. CLIC is targeted at CLIC per-core and has the option to give each interrupt source a separate trap entry address, preemption (nesting) of interrupts with adjustable priority threshold control, and support for software tailchaining.

Chapter 3. CLIC Overview

This section gives an overview for the Core-Local Interrupt Controller (CLIC) that receives interrupt signals and presents the next interrupt to be processed by the hart.

The CLIC supports up to 4096 interrupt inputs per hart. Each interrupt input *i* has four 8-bit memory-mapped control registers: an interrupt-pending bit (`clicintip[i]`), an interrupt-enable bit (`clicintie[i]`), interrupt attributes (`clicintattr[i]`) to specify privilege mode and trigger type, and interrupt control bits to specify level and priority (`clicintctl[i]`).

The first 16 interrupt inputs are reserved for the original basic mode interrupts present in the low 16 bits of the `xip` and `xie` registers, so up to 4080 local external interrupts can be added.

3.1. Interrupt Preemption

The CLIC extends interrupt preemption to support up to 256 interrupt levels for each privilege mode, where higher-numbered interrupt levels can preempt lower-numbered interrupt levels. Interrupt level 0 corresponds to regular execution outside of an interrupt handler. Levels 1—255 correspond to interrupt handler levels. Platform profiles will dictate how many interrupt levels must be supported.

Incoming interrupts with a higher interrupt level can preempt an active interrupt handler running at a lower interrupt level in the same privilege mode, provided interrupts are globally enabled in this privilege mode.

> Existing RISC-V interrupt behavior is retained, where incoming interrupts for a higher privilege mode can preempt an active interrupt handler running in a lower privilege mode, regardless of global interrupt enable in lower privilege mode.

3.2. CLIC Interaction with Other Local Interrupts

The CLIC subsumes the functionality of the basic local interrupts previously provided in bits 16 and up of `xip`/`xie`, so these are no longer visible in `xip`/`xie`.

The existing timer (`mtip`/`stip`/`utip`), software (`msip`/`ssip`/`usip`), and external interrupt inputs (`meip`/`seip`/`ueip`) are treated as additional local interrupt sources, where the privilege mode, interrupt level, and priority can be altered using memory-mapped `clicintattr[i]` and `clicintctl[i]` registers.

> In CLIC mode, interrupt delegation for these signals is achieved via changing the interrupt's privilege mode in the CLIC Interrupt Attribute Register (`clicintattr`), as with any other CLIC interrupt input.

Chapter 4. CLIC Memory-Mapped Registers

4.1. CLIC Memory Map

Each hart has a separate CLIC accessed by a separate address region. The M-mode CLIC memory map region must be made accessible to the M-mode software running on the hart.

> A bus memory map or locked PMP entries could prevent M-mode software on a particular hart from reaching the CLIC memory map.

The base address of M-mode CLIC memory-mapped registers is specified at a new CLIC Base (`mclicbase`) Control and Status Register (CSR).

> The `mclicbase` register and `clicinfo` are likely to be replaced by the general discovery mechanism that is in development.

The CLIC memory map supports up to 4096 total interrupt inputs.

```
M-mode CLIC memory map
  Offset
  ###    0x0008-0x003F            reserved    ###
  ###    0x00C0-0x07FF            reserved    ###
  ###    0x0800-0x0FFF            custom      ###


  0x0000         1B         RW        cliccfg
  0x0004         4B         R         clicinfo


  0x0040         4B         RW        clicinttrig[0]
  0x0044         4B         RW        clicinttrig[1]
  0x0048         4B         RW        clicinttrig[2]
  ...
  0x00B4         4B         RW        clicinttrig[29]
  0x00B8         4B         RW        clicinttrig[30]
  0x00BC         4B         RW        clicinttrig[31]


  0x1000+4*i     1B/input   R or RW   clicintip[i]
  0x1001+4*i     1B/input   RW        clicintie[i]
  0x1002+4*i     1B/input   RW        clicintattr[i]
  0x1003+4*i     1B/input   RW        clicintctl[i]
  ...
  0x4FFC         1B/input   R or RW   clicintip[4095]
  0x4FFD         1B/input   RW        clicintie[4095]
  0x4FFE         1B/input   RW        clicintattr[4095]
  0x4FFF         1B/input   RW        clicintctl[4095]
```

Supervisor-mode CLIC regions only expose interrupts that have been configured to be supervisor-accessible via the M-mode CLIC region.

```
Layout of Supervisor-mode CLIC regions
0x000+4*i    1B/input    R or RW    clicintip[i]
0x001+4*i    1B/input    RW         clicintie[i]
0x002+4*i    1B/input    RW         clicintattr[i]
0x003+4*i    1B/input    RW         clicintctl[i]
```

User-mode CLIC regions only expose interrupts that have been configured to be user-accessible via the M-mode CLIC region.

The location of the S-mode and U-mode CLIC regions are independent of the location of the M-mode CLIC region, and their base addresses are specified by the platform specification and made visible via the discovery mechanism for that platform.

Discovery mechanisms are still in development.

```
Layout of user-mode CLIC regions
0x000+4*i    1B/input    R or RW    clicintip[i]
0x001+4*i    1B/input    RW         clicintie[i]
0x002+4*i    1B/input    RW         clicintattr[i]
0x003+4*i    1B/input    RW         clicintctl[i]
```

A 32-bit write to {clicintctl,clicintattr,clicintie,clicintip} is legal. However, there is no specified order in which the effects of the individual byte updates take effect.

If an input i is not present in the hardware, the corresponding clicintip[i], clicintie[i], clicintattr[i], clicintctl[i] memory locations appear hardwired to zero.

All CLIC-memory mapped registers are visible to M-mode. Interrupt registers clicintip[i], clicintie[i], clicintattr[i], clicintctl[i] configured as M-mode interrupts are not acessible to S-mode and U-mode. Interrupt registers clicintip[i], clicintie[i], clicintattr[i], clicintctl[i] configured as S-mode interrupts are not acessible to U-mode.

In S-mode, any interrupt i that is not accessible to S-mode appears as hard-wired zeros in clicintip[i], clicintie[i], clicintattr[i], and clicintctl[i].

Likewise, in U-mode, any interrupt i that is not accessible to U-mode appears as hard-wired zeros in clicintip[i], clicintie[i], clicintattr[i], and clicintctl[i].

The privilege mode of an interrupt is controlled by both cliccfg.nmbits and clicintattr[i].mode as described in the Specifying Interrupt Privilege Mode section below.

It is not intended that the interconnect to the CLIC memory-mapped interrupt regions be required to carry the privilege mode of the initiator. A possible implementation of the CLIC memory map would be to alias the same physical CLIC memory-mapped registers to different address ranges, with each address range given different permissions for each privilege mode. Interrupts configured as M-mode interrupts appear as hard-wired zeros in the S-mode address range. Likewise interrupts configured as M-mode or S-mode would appear as hard-wired zeros in the U-mode address range.

The intent is that only the necessary address regions are made accessible to each privilege mode using the system's standard memory protection mechanisms. This can be done either using PMPs in microcontroller systems, or page tables (and/or PMPs) in harts with virtual memory support.

The CLIC specification does not dictate how CLIC memory-mapped registers are split between M/S/U regions as well as the layout of multiple harts as this is generally a platform issue and each platform needs to define a discovery mechanism to determine the memory map locations. Some considerations for platforms to consider are selecting regions that allow for efficient PMP and virtual memory configuration. For example, it may desired that the bases of each S/U-mode CLIC region is VM page (4k) aligned so they can be mapped through the TLBs.

4.2. CLIC Configuration (`cliccfg`)

The CLIC has a single memory-mapped 8-bit global configuration register, `cliccfg`, that defines how many privilege modes are supported, how the `clicintctl[i]` registers are subdivided into level and priority fields, and whether selective hardware vectoring is supported.

The `cliccfg` register has three WARL fields, a 2-bit `nmbits` field, a 4-bit `nlbits` field, and a 1-bit `nvbits` field, plus a reserved bit WPRI-hardwired to zero in current spec.

> WPRI means "Writes Preserve Values, Reads Ignore Values" indicating whole read/write fields are reserved for future use. Software should ignore the values read from these fields, and should preserve the values held in these fields when writing values to other fields of the same register. For forward compatibility, implementations that do not furnish these fields must hardwire them to zero.

```
cliccfg register layout


Bits    Field
7       reserved (WPRI 0)
6:5     nmbits[1:0]
4:1     nlbits[3:0]
  0     nvbits
```

Detailed explanation for each field are described in the following sections.

4.2.1. Specifying Interrupt Privilege Mode

The 2-bit `cliccfg.nmbits` WARL field specifies how many bits are physically implemented in `clicintattr[i].mode` to represent an input *i*'s privilege mode. Although `cliccfg.nmbits` field is always 2-bit wide, the physically implemented bits in this field can be fewer than two (depending how many interrupt privilege-modes are supported).

For example, in M-mode-only systems, only M-mode exists so we do not need any extra bit to represent the supported privilege-modes. In this case, no physically implemented bits are needed in the `clicintattr.mode` and thus `cliccfg.nmbits` is 0 (i.e., `cliccfg.nmbits` can be hardwired to 0).

In M/U-mode systems with N-extension user-level interrupts support, `cliccfg.nmbits` can be set to 0 or 1. If `cliccfg.nmbits` = 0, then all interrupts are treated as M-mode interrupts. If the `cliccfg.nmbits` = 1, then a value of 1 in the most-significant bit (MSB) of a `clicintattr[i].mode` register indicates that interrupt intput is taken in M-mode, while a value of 0 indicates that interrupt is taken in U-mode.

Similarly, in systems that support all M/S/U-mode interrupts, `cliccfg.nmbits` can be set to 0, 1, or 2 bits to represent privilege-modes. `cliccfg.nmbits` = 0 indicates that all local interrupts are taken in M-mode. `cliccfg.nmbits` = 1 indicates that the MSB selects between M-mode (1) and S-mode (0). `cliccfg.nmbits` = 2 indicates that the two MSBs of each `clicintattr[i].mode` register encode the interrupt's privilege mode using the same encoding as the `mstatus.mpp` field.

> Bare S-mode (no MMU, satp=0) can be used in microcontrollers to allow hardware delegation of interrupts out of M-mode. Bare S-mode has already been ratified as part of privileged architecture. There are also proposals to add S-mode PMP support to allow an RTOS running in S-mode to isolate itself from tasks running in U-mode. The proposed N-extension would also add user-mode interrupts and traps, but has not been ratified and is not currently being advanced.

```
Encoding for RISC-V privilege levels (mstatus.mpp)


Level   Encoding  Name                Abbreviation
0       00        User/Application    U
1       01        Supervisor          S
2       10        Reserved
3       11        Machine             M
```

```
priv-modes  nmbits  clicintattr[i].mode   Interpretation
      M       0        xx                  M-mode interrupt

        M/U     0        xx                  M-mode interrupt
        M/U     1        0x                  U-mode interrupt
        M/U     1        1x                  M-mode interrupt

      M/S/U     0        xx                  M-mode interrupt
      M/S/U     1        0x                  S-mode interrupt
      M/S/U     1        1x                  M-mode interrupt
      M/S/U     2        00                  U-mode interrupt
      M/S/U     2        01                  S-mode interrupt
      M/S/U     2        10                  Reserved (or extended S-mode)
      M/S/U     2        11                  M-mode interrupt


      M/S/U     3        xx                  Reserved
```

4.2.2. Specifying Interrupt Level

The 4-bit `cliccfg.nlbits` WARL field indicates how many upper bits in `clicintctl[i]` are assigned to encode the interrupt level.

Only 0 or 8 level bits are currently supported, with other values currently reserved.

> In effect, this switches the control bits from being used only for level or only for priority. The design supports a wider range of level-bit settings but this is not currently being standardized.

Although the interrupt level is an 8-bit unsigned integer, the number of bits actually assigned or implemented can be fewer than 8. As described above, the number of bits assigned is specified in `cliccfg.nlbits`. The number of bits actually implemented can be derived from `cliccfg.nlbits` and a fixed parameter `clicinfo.CLICINTCTLBITS` (with value between 0 to 8) which specifies bits implemented for both interrupt level and priority.

> The number of available level bits can be determined by subtracting the number of mode bits from CLICINTCTLBITS.

If the actual bits assigned or implemented are fewer than 8, then these bits are left-justified and appended with 1's for the lower missing bits. For example, if the `nlbits` > `CLICINTCTLBITS`, then the lower bits of the 8-bit interrupt level are assumed to be all 1s. Similarly, if `nlbits` < 8, then the lower bits of the 8-bit interrupt level are assumed to be all 1s. The following table shows how levels are encoded for these cases.

```
#bits    encoding          interrupt levels
   0     .......                                                       255
   1     l.......                             127,                     255
   2     ll......            63,              127,         191,        255
   3     lll.....     31,  63,  95,   127,    159,   191,   223,   255
   4     llll....  15,31,47,63,79,95,111,127,143,159,175,191,207,223,239,255


"l" bits are available variable bits in level specification
"." bits are non-existent bits for level encoding, assumed to be 1
```

If `nlbits` = 0, then all interrupts are treated as level 255.

Examples of `cliccfg` settings:

```
CLICINTCTLBITS nlbits clicintctl[i] interrupt levels
      0        2     .......       255
      1        2     l.......      127,255
      2        2     ll......      63,127,191,255
      3        3     lll.....      31,63,95,127,159,191,223,255
      4        1     lppp....      127,255
```

```
"." bits are non-existent bits for level encoding, assumed to be 1
"l" bits are available variable bits in level specification
"p" bits are available variable bits in priority specification
```

4.2.3. Specifying Interrupt Priority

The least-significant bits in `clicintctl[i]` that are not configured to be part of the interrupt level are interrupt priority, which are used to prioritize among interrupts pending-and-enabled at the same privilege mode and interrupt level. The highest-priority interrupt at a given privilege mode and interrupt level is taken first. In case there are multiple pending-and-enabled interrupts at the same highest priority, the highest-numbered interrupt is taken first.

> The highest numbered interrupt wins in a tie (when privilege mode, level and priority are all identical). This is the same as the original basic interrupt mode, but different than the PLIC.

Notice that the 8-bit interrupt level is used to determine preemption (for nesting interrupts). In contrast, the 8-bit interrupt priority does not affect preemption but is only used as a tie-breaker when there are multiple pending interrupts with the same interrupt level.

Any implemented priority bits are treated as the most-significant bits of a 8-bit unsigned integer with lower unimplemented bits set to 1. For example, with one priority bit (p111_1111), interrupts can be set to have priorities 127 or 255, and with two priority bits (pp11_1111), interrupts can be set to have priorities 63, 127, 191, or 255.

4.2.4. Specifying Support for Selective Interrupt Hardware Vectoring

The single-bit read-only `nvbits` field in `cliccfg` specifies whether the selective interrupt hardware vectoring feature is implemented or not.

This selective hardware vectoring feature gives users the flexibility to select the behavior for each interrupt: either hardware vectoring or non-vectoring. As a result, it allows users to optimize each interrupt and enjoy the benefits of both behaviors. More specifically, hardware vectoring has the advantage of faster interrupt response at the price of slightly increasing the code size (to save/restore contexts). On the other hand, non-vectoring has the advantage of smaller code size (by sharing and reusing one copy of common code to save/restore contexts) at the price of slightly slower interrupt response.

When `nvbits` is 0, selective interrupt hardware vectoring is not implemented. In this case, all interrupts are non-vectored and are directed to the common code at `xtvec` register.

When `nvbits` is 1, selective interrupt hardware vectoring is implemented. The bit `clicintattr[i].shv` controls the vectoring behavior of interrupt i. If `clicintattr[i].shv` is 0, then the interrupt is non-vectored and always jumps to the common code at `xtvec`. If `clicintattr[i].shv` is 1, then the interrupt is hardware vectored to the trap-handler function pointer specified in `xtvt` CSR. This allows some interrupts to all jump to a common base address held in `xtvec`, while the others are vectored in hardware via a table pointed to by the additional `xtvt` CSR.

4.3. CLIC Information (`clicinfo`)

This is a read-only register to show information useful for debugging. NOTE: `clicinfo` is likely to be replaced by the general discovery mechanism that is in development.

```
clicinfo register layout


Bits    Field
31      reserved (WARL 0)
30:25   num_trigger (number of maximum interrupt triggers supported)
24:21   CLICINTCTLBITS
20:13   version (for version control)
        20:17 for architecture version, 16:13 for implementation version
12:0    num_interrupt (number of maximum interrupt inputs supported)
```

The `num_interrupt` field specifies the actual number of maximum interrupt inputs supported in this

implementation.

The `version` field specifies the implementation version of CLIC. The upper 4-bit specifies the architecture version, and the lower 4-bit specifies the implementation version.

The `CLICINTCTLBITS` field specifies how many hardware bits are actually implemented in the `clicintctl` registers, with $0 \leq$ `CLICINTCTLBITS` ≤ 8. The implemented bits are kept left-justified in the most-significant bits of each 8-bit `clicintctl[i]` register, with the lower unimplemented bits treated as hardwired to 1.

The `num_trigger` field specifies the number of maximum interrupt triggers supported in this implementation. Valid values are 0 to 32.

4.4. CLIC Interrupt Pending (`clicintip`)

Each interrupt input has a dedicated interrupt pending bit (`clicintip[i]`) and occupies one byte in the memory map for ease of access. The pending bit is located in bit 0 of the byte. Software should assume `clicintip[i]=0` means no interrupt pending, and `clicintip[i]!=0` indicates an interrupt is pending to accomodate possible future expansion of the `clicintip` field.

When the input is configured for level-sensitive input, the `clicintip[i]` bit reflects the value of an input signal to the interrupt controller after any conditional inversion specified by the `clicintattr[i]` field, and software writes to the bit are ignored. Software clears the interrupt at the source device.

When the input is configured for edge-sensitive input, `clicintip[i]` is a read-write register that can be updated both by hardware interrupt inputs and by software. The bit is set by hardware after an edge of the appropriate polarity is observed on the interrupt input, as determined by the `clicintattr[i]` field. Hardware clears the associated interrupt pending bit when an interrupt is serviced in vectored mode. See additional detail on hardware clearing in the `xtvec` section. Software writes can set or clear edge-triggered pending bits directly by writes to the memory-mapped register. Edge-triggered pending bits can also be cleared when a CSR instruction that accesses `xnxti` includes a write.

> To improve performance, when a vectored interrupt is selected and serviced, the hardware will automatically clear a corresponding edge-triggered pending bit, so software doesn't need to clear the pending bit in the service routine.

In contrast, when a non-vectored (common code) interrupt is selected, the hardware will not automatically clear an edge-triggered pending bit.

> Software is expected to use a CSR instruction that accesses `xnxti` that includes a write to clear an edge-triggered pending bit in non-vectored mode. Additional detail on this is described in the `xnxti` section.

The value in the `clicintip[i]` is undefined when switching from level-sensitive mode to edge-triggered mode in `clicintattr[i]`.

> Software cannot rely on the underlying `clicintip[i]` register bits used in edge-triggered mode to hold state while in level-sensitive mode.

4.5. CLIC Interrupt Enable (`clicintie`)

Each interrupt input has a dedicated interrupt-enable bit (`clicintie[i]`) and occupies one byte in the memory map for ease of access. This control bit is read-write to enable/disable the corresponding interrupt. The enable

bit is located in bit 0 of the byte. Software should assume clicintie[i]=0 means no interrupt enabled, and clicintie[i]!=0 indicates an interrupt is enabled to accomodate possible future expansion of the clicintie field.

> `clicintie[i]` is the individual enable bit while `xstatus.xie` is the global enable bit for the current privilege mode. Therefore, for an interrupt *i* to be enabled in the current privilege mode, both `clicintie[i]` and `xstatus.xie` have to be set.

> In contrast, since `xstatus.xie` only takes effect in the current privilege mode according to RISC-V convention, an interrupt *i* from a higher privilege mode is enabled as long as `clicintie[i]` is set (regardless of the setting of `xstatus.xie` in the higher privilege modes).

4.6. CLIC Interrupt Attribute (`clicintattr`)

This is an 8-bit WARL read-write register to specify various attributes for each interrupt.

```
clicintattr register layout


Bits    Field
7:6     mode
5:3     reserved (WPRI 0)
2:1     trig
0       shv
```

The 1-bit `shv` field is used for Selective Hardware Vectoring. If `shv` is 0, it assigns this interrupt to be non-vectored and thus it jumps to the common code at `xtvec`. If `shv` is 1, it assigns this interrupt to be hardware vectored and thus it automatically jumps to the trap-handler function pointer specified in `xtvt` CSR. This feature allows some interrupts to all jump to a common base address held in `xtvec`, while the others are vectored in hardware via a table pointed to by the additional `xtvt` CSR.

> if `cliccfg.nvbits` is 0, the selective interrupt hardware vectoring feature is not implemented and thus `shv` field appears hardwired to zero (WARL 0).

The 2-bit `trig` WARL field specifies the trigger type and polarity for each interrupt input. Bit 1, `trig[0]`, is defined as "edge-triggered" (0: level-triggered, 1: edge-triggered); while bit 2, `trig[1]`, is defined as "negative-edge" (0: positive-edge, 1: negative-edge). More specifically, there can be four possible combinations: positive level-triggered, negative level-triggered, positive edge-triggered, and negative edge-triggered.

> Some implementations may want to save these bits so only certain trigger types are supported. In this case, these bits become hard-wired to fixed values (WARL).

The 2-bit `mode` WARL field specifies which privilege mode this interrupt operates in. This field uses the same encoding as the `mstatus.mpp` (11: machine mode, 01: supervisor mode, 00 user mode). The valid length of this field can be programmed with `cliccfg.nmbits`.

> For security purpose, the `mode` field can only be set to a privilege level that is equal to or lower than the currently running privilege level.

4.7. CLIC Interrupt Input Control (`clicintctl`)

`clicintctl[i]` is an 8-bit memory-mapped WARL control register to specify interrupt level and interrupt priority. The number of bits actually implemented in this register is specified by a fixed parameter `CLICINTCTLBITS` (in `clicinfo`), which has a value between 0 to 8. The implemented bits are kept left-justified in the most-significant bits of each 8-bit `clicintctl[i]` register, with the lower unimplemented bits treated as hardwired to 1. These control bits are interpreted as level and priority according to the setting in the CLIC Configuration register (`cliccfg.nlbits`).

To select an interrupt to present to the core, the CLIC hardware combines the valid bits in `clicintattr.mode` and `clicintctl` to form an unsigned integer, then picks the global maximum across all pending-and-enabled interrupts based on this value. Next, the `cliccfg` setting determines how to split the `clicintctl` value into interrupt level and interrupt priority. Finally, the interrupt level of this selected interrupt is compared with the interrupt-level threshold of the associated privilege mode to determine whether it is qualified or masked by the threshold (and thus no interrupt is presented).

> Selecting an interrupt at a high privilege mode masks any interrupt at a lower privilege mode since the higher-privilege mode causes the interrupt signal to appear more urgent than any lower-privilege mode interrupt.

4.7.1. Interrupt Input Identification Number

The 4096 CLIC interrupt vectors are given unique identification numbers with `xcause` Exception Code (`exccode`) values. To maintain backward compatibility, the original basic mode interrupts retain their original cause values, while the new interrupts are numbered starting at 16.

> When upgrading an earlier original basic interrupt controller that had local interrupts attached directly to bits 16 and above, these local interrupts can be now attached as CLIC inputs 16 and above to retain the same interrupt IDs.

4.8. CLIC Interrupt Trigger (`clicinttrig`)

Optional interrupt triggers (`clicinttrig[i]`) are used to generate a breakpoint exception, entry into Debug Mode, or a trace action. The actual number of triggers supported is specified in `clicinfo.num_trigger`.

Each interrupt trigger is a 32-bit memory-mapped WARL register with the following layout:

```
clicinttrig register layout


Bits    Field
31      enable
30:13   reserved (WARL 0)
12:0    interrupt_number
```

The `interrupt_number` field selects which number of interrupt input is used as the source for this interrupt trigger.

The `enable` control bit is read-write to enable/disable this interrupt trigger.

The detailed behavior of the trigger is defined in the debug spec. For example, the trigger only fires if the

interrupt is actually taken (and not when the interrupt is masked, or not taken). In addition, the requested action (e.g., breakpoint or trace) is taken just before the first instruction of the interrupt handler is executed.

Chapter 5. CLIC CSRs

This section describes the CLIC-related hart-specific Control and Status Registers (CSRs). When in original basic interrupt mode, the behavior is intended to be software compatible with basic-mode-only systems.

The interrupt-handling CSRs are listed below, with changes and additions for CLIC mode described in the following sections.

```
        Number  Name         Description
        0xm00   xstatus      Status register
        0xm02   xedeleg      Exception delegation register
        0xm03   xideleg      Interrupt delegation register (INACTIVE IN CLIC MODE)
        0xm04   xie          Interrupt-enable register    (INACTIVE IN CLIC MODE)
        0xm05   xtvec        Trap-handler base address / interrupt mode
  (NEW) 0xm07   xtvt         Trap-handler vector table base address
        0xm40   xscratch     Scratch register for trap handlers
        0xm41   xepc         Exception program counter
        0xm42   xcause       Cause of trap
        0xm43   xtval        Bad address or instruction
        0xm44   xip          Interrupt-pending register   (INACTIVE IN CLIC MODE)
  (NEW) 0xm45   xnxti        Interrupt handler address and enable modifier
  (NEW) 0xm46   xintstatus   Current interrupt levels
  (NEW) 0xm47   xintthresh   Interrupt-level threshold
  (NEW) 0xm48   xscratchcsw  Conditional scratch swap on priv mode change
  (NEW) 0xm49   xscratchcswl Conditional scratch swap on level change
  (NEW) 0x3??   mclicbase    Base address for CLIC memory mapped registers


          m is the nibble encoding the privilege mode (M=0x3, S=0x1, U=0x0)
```

5.1. Changes to `xstatus` CSRs

When in original basic interrupt mode, the `xstatus` register behavior is unchanged (i.e., backwards-compatible with original basic mode). When in CLIC mode, the `xpp` and `xpie` in `xstatus` are now accessible via fields in the `xcause` register.

5.2. Changes to Delegation (`xedeleg`/`xideleg`) CSRs

In CLIC mode, the `mode` field in Interrupt Attribute Register (`clicintattr[i].mode`) specifies the privilege mode in which each interrupt should be taken, so the `xideleg` CSR ceases to have effect in CLIC mode. The `xideleg` CSR is still accessible and state bits retain their values when switching between CLIC and original basic interrupt modes.

Exception delegation specified by `xedeleg` functions the same in CLIC mode as in original basic mode.

5.3. Changes to `xie`/`xip` CSRs

The `xie` CSR appears hardwired to zero in CLIC mode, replaced by separate memory-mapped interrupt enables (`clicintie[i]`).

The `xip` CSR appears hardwired to zero in CLIC mode, replaced by separate memory-mapped interrupt pendings (`clicintip[i]`).

Writes to `xie`/`xip` will be ignored and will not trap (i.e., no access faults). `xie`/`xip` always appear to be zero in CLIC mode.

In systems that support both original basic and CLIC modes, the state bits in `xie` and `xip` retain their value when switching between modes.

5.4. New `xtvec` CSR Mode for CLIC

The new CLIC interrupt-handling mode is encoded as a new state in the existing `xtvec` WARL register, where `xtvec.mode` (the least-significant two bits) is `11`. In this mode, the trap vector base address held in `xtvec` is constrained to be aligned on a 64-byte or larger power-of-two boundary.

> Although future CLIC versions may allow privileges to have different `xtvec.mode` settings, for now all privilege modes must run in either CLIC mode or all privilege modes must run in non-CLIC mode.

```
  mtvec    Action on Interrupt
  aaaa00   pc := OBASE                                (original non-vectored basic
mode)
  aaaa01   pc := OBASE + 4 * exccode                  (original vectored basic mode)

  000011                                              (CLIC mode)
           (non-vectored)
           pc := NBASE                                    if clicintattr[i].shv = 0
                                                          || if cliccfg.nvbits = 0
                                                              (vector not supported)
           (vectored)
           pc := M[TBASE + XLEN/8 * exccode)] & ~1     if clicintattr[i].shv = 1

  000010                                              Reserved
  xxxx1?   (xxxx!=0000)                               Reserved

  OBASE = mtvec[XLEN-1:2]<<2   # Original vector base was at least 4-byte aligned.
  NBASE = mtvec[XLEN-1:6]<<6   # New vector base is at least 64-byte aligned.
  TBASE = mtvt[XLEN-1:6]<<6    # Trap vector table base is at least 64-byte
aligned.
```

In CLIC mode, writing `0` to `clicintattr[i].shv` sets interrupt `i` to non-vectored, where the hart jumps to the trap handler address held in the upper XLEN-6 bits of `xtvec` for all exceptions and interrupts in privilege mode `x`. Similarly, if the selective hardware vectoring feature is not implemented (`cliccfg.nvbits` is `0`), all interrupts are non-vectored and behave the same.

On the other hand, writing `1` to `clicintattr[i].shv` sets interrupt `i` to vectored. When these interrupts are taken, the hart switches to the handler's privilege mode, sets the hardware vectoring bit `xinhv` in `xcause` of the handler privilege mode, and clears `xstatus.xie`. At this time, if the associated interrupt pending bit is configured for edge-sensitive input, it is cleared by hardware. The hart then fetches an XLEN-bit handler address with permissions corresponding to the handler's mode from the in-memory table whose base address (TBASE) is in `xtvt`. The trap handler function address is fetched from TBASE+XLEN/8*exccode. If the fetch is successful, the hart clears the low bit of the handler address, sets the PC to this handler address, then clears the `xinhv` bit in `xcause` of the handler privilege mode. The overall effect is:

```
pc := M[TBASE + XLEN/8 * exccode] & ~1
```

```
            # Vector table layout for RV32 (4-byte function pointers)
   mtvt ->  0x800000 # Interrupt 0 handler function pointer
            0x800004 # Interrupt 1 handler function pointer
            0x800008 # Interrupt 2 handler function pointer
            0x80000c # Interrupt 3 handler function pointer

            # Vector table layout for RV64 (8-byte function pointers)
   mtvt ->  0x800000 # Interrupt 0 handler function pointer
            0x800008 # Interrupt 1 handler function pointer
            0x800010 # Interrupt 2 handler function pointer
            0x800018 # Interrupt 3 handler function pointer
```

The original basic vectored mode simply jumped to an address in the trap vector table, while the new CLIC vectored mode reads a handler function address from the table, and jumps to it in hardware.

The vector table contains vector addresses rather than instructions because it simplifies static initialization in C. More specifically, the entries in the table are simple XLEN-bit function pointers.

The hardware vectoring bit `xinhv` is provided to allow resumable traps on fetches to the trap vector table.

The `xinhv` bits are only written by hardware during the table vector read operation. The `xinhv` bits can be written by software, including when hardware vectoring is not in effect. The `xinhv` bit has no effect except when returning from an exception using an `xret` instruction. Since successful hardware vector fetches clear `xinhv`, if `xinhv` of the previous privilege mode is set, it implies an exception occurred during previous privilege mode table vector read operation. So when `xinhv` of the previous privilege is set, `xret` should consider `xepc` as an address instead of an instruction.

When returning from an `xret` instruction, the `xinhv` bit modifies behavior as follows:

```
if (xcause.inhv) //xcause here refers to the cause CSR of the previous privilege
mode
    pc := M[xepc]
else
    pc := xepc
```

> Horizontal traps (same privilege level) are unrecoverable. The interesting case is vertical traps, where a more privileged layer is handling page faults or other synchronous faults on vector table access. The regular code path in more privileged layer will want to use xtval to determine what bad virtual address to page in, but will not normally restore xtval when returning to faulting context (potentially after some time and other contexts have run) However, it will restore xepc (using x for more privileged mode here) before using xret on normal code path. This is a rationale for why both xtval and xepc are written with the faulting address.

Implementations might support only one of original basic or CLIC mode. If only basic mode is supported, writes to bit 1 are ignored and it is always set to zero (current behavior). If only CLIC mode is supported, writes to bit 1 are also ignored and it is always set to one. CLIC mode hardwires xtvec bits 2-5 to zero (assuming no further CLIC extensions are supported).

For permissions-checking purposes, the memory access to retrieve the function pointer for vectoring is treated as a load with the privilege mode (also obeying MPRV and SUM bits) and interrupt level of the interrupt handler. If there is an access exception on the table load, both xtval and xepc holds the faulting address.

In CLIC mode, synchronous exception traps always jump to NBASE.

5.5. New xtvt CSRs

The xtvt WARL XLEN-bit CSR holds the base address of the trap vector table, aligned on a 64-byte or greater power-of-two boundary. The actual alignment can be determined by writing ones to the low-order bits then reading them back. Values other than 0 in the low 6 bits of xtvt are reserved.

In systems that support both original basic and CLIC modes, the xtvt CSR is still accessible in basic mode (but does not have any effect).

5.6. Changes to xepc CSRs

The xepc CSRs behave the same in both modes, capturing the PC at which execution was interrupted.

5.7. Changes to xcause CSRs

In both original basic and CLIC modes, the xcause CSR is written at the time an interrupt or synchronous trap is taken, recording the reason for the interrupt or trap. For CLIC mode, xcause is also extended to record more information about the interrupted context, which is used to reduce the overhead to save and restore that context for an xret instruction. CLIC mode xcause also adds state to record progress through the trap handling process.

```
mcause
Bits    Field      Description
XLEN-1 Interrupt    Interrupt=1, Exception=0
   30  minhv        Set by hardware at start of hardware vectoring, cleared by
hardware at end of successful hardware vectoring
29:28  mpp[1:0]     Previous privilege mode, same as mstatus.mpp
   27  mpie         Previous interrupt enable, same as mstatus.mpie
26:24  (reserved)
23:16  mpil[7:0]    Previous interrupt level
15:12  (reserved)
11:0   Exccode[11:0] Exception/interrupt code
```

The `mcause.mpp` and `mcause.mpie` fields mirror the `mstatus.mpp` and `mstatus.mpie` fields, and are aliased into `mcause` to reduce context save/restore code.

If the hart is currently running at some privilege mode (pp) at some interrupt level (pil) and an enabled interrupt becomes pending at any interrupt level in a higher privilege mode or if an interrupt at a higher interrupt level in the current privilege mode becomes pending and interrupts are globally enabled in this privilege mode, then execution is immediately transferred to a handler running with the new interrupt's privilege mode (x) and interrupt level (il).

The CSR `xepc` is set to the PC of the interrupted application code or preempted interrupt handler, while the `xcause` register now captures the previous privilege mode (pp), interrupt level (pil) and interrupt enable (pie), as well as the id of the interrupt in `exccode`.

In systems supporting both original basic and CLIC modes, the new CLIC-specific fields (`minhv`, `mpp`, `mpil`, `mpie`) appear to be hardwired to zero in basic mode for backwards compatibilty. When basic mode is written to `xtvec`, the new `xcause` state fields (`mhinv` and `mpil`) are zeroed. The other new `xcause` fields, `mpp` and `mpie`, appear as zero in the `xcause` CSR but the corresponding state bits in the `mstatus` register are not cleared.

The supervisor `scause` register has only a single `spp` bit (to indicate user/supervisor) mirrored from `sstatus.spp`, while the user `ucause` register has no `upp` bit as interrupts can only have come from user mode.

```
scause
Bits     Field        Description
XLEN-1 Interrupt      Interrupt=1, Exception=0
    30  sinhv         Set by hardware at start of hardware vectoring, cleared by
 hardware at end of successful hardware vectoring
    29  (reserved)
    28  spp           Previous privilege mode, same as sstatus.spp
    27  spie          Previous interrupt enable, same as sstatus.spie
 26:24  (reserved)
 23:16  spil[7:0]     Previous interrupt level
 15:12  (reserved)
 11:0   exccode[11:0] Exception/interrupt code


ucause
Bits     Field        Description
XLEN-1 Interrupt      Interrupt=1, Exception=0
    30  uinhv         Set by hardware at start of hardware vectoring, cleared by
 hardware at end of successful hardware vectoring
 29:28  (reserved)
    27  upie          Previous interrupt enable, same as ustatus.upie
 26:24  (reserved)
 23:16  upil[7:0]     Previous interrupt level
 15:12  (reserved)
 11:0   exccode[11:0] Exception/interrupt code
```

For exceptions, in CLIC mode, the `mcause` has the new CLIC format. On the other hand, in other modes, the `mcause` has the original format.

5.8. Next Interrupt Handler Address and Interrupt-Enable CSRs (`xnxti`)

The `xnxti` CSR can be used by software to service the next horizontal interrupt for the same privilege mode when it has greater level than the saved interrupt context (held in `xcause`.pil`) and greater level than the interrupt threshold of the corresponding privilege mode, without incuring the full cost of an interrupt pipeline flush and context save/restore. The `xnxti` CSR is designed to be accessed using CSRRSI/CSRRCI instructions, where the value read is a pointer to an entry in the trap handler table and the write back updates the interrupt-enable status. In addition, writes to the `xnxti` have side-effects that update the interrupt context state.

> This is different than a regular CSR instruction as the value returned is different from the value used in the read-modify-write operation.

> All CSR instructions may be used with `xnxti`. The operation will be the same as the corresponding CSR instruction used with xstatus. The value read and the side-effects will be as specified in this section.

A read of the `xnxti` CSR returns either zero, indicating there is no suitable interrupt to service or that the highest ranked interrupt is SHV or that the system is not in a CLIC mode, or returns a non-zero address of the

entry in the trap handler table for software trap vectoring.

> The xtvt CSR could be set to memory addresses such that a table entry was at address zero, and this would be indistinguishable from the no-interrupt case.

If the CSR instruction that acccesses xnxti includes a write, the xstatus CSR is the one used for the read-modify-write portion of the operation, while the xcause register's exccode field, the xintstatus register's xil field and the pending bit of the accepted interrupt can also be updated with the new interrupt id, level, and clear respectively.

> Following the usual convention for CSR instructions, if the CSR instruction does not include write side effects (e.g., csrr t0, xnxti), then no state update on any CSR occurs. This can be used to determine if an interrupt could be taken without actually updating xil and exccode.

The xnxti CSR is intended to be used inside an interrupt handler after an initial interrupt has been taken and xcause and xepc registers updated with the interrupted context and the id of the interrupt.

If the pending interrupt is edge-triggered, hardware will automatically clear the corresponding pending bit when the CSR instruction that accesses xnxti includes a write. However, if the CSR instruction does not include write side effects (e.g., csrr t0, xnxti), then no state update on any CSR occurs and thus the interrupt pending bit is not cleared. This behavior allows software to optimize the selection and execution of interrupts using xnxti.

```
 // Pseudo-code for csrrsi rd, mnxti, uimm[4:0] in M mode.
 mstatus |= uimm[4:0]; // Performed regardless of interrupt readiness.
 if (clic.priv==M && clic.level > mcause.pil && clic.level > mintthresh.th
     && (cliccfg.nvbits==0 || clicintattr.shv==0) ) {
   // There is an available, non-hardware-vectored interrupt.
   if (uimm[4:0] != 0) {  // Side-effects should occur.
     // Commit to servicing the available interrupt.
     mintstatus.mil = clic.level; // Update hart's interrupt level.
     mcause.exccode = clic.id;   // Update interrupt id in mcause.
     mcause.interrupt = 1;       // Set interrupt bit in mcause.
     if (clicintattr[clic.id][1] == 1) { // If edge interrupt,
       clicintip[clic.id] = 0;          // clear edge interrupt
     }
   }
   rd = TBASE + XLEN/8 * clic.id; // Return pointer to trap handler entry.
 } else {
   // No interrupt, or a selectively hardware vectored interrupt, or in non-CLIC
mode.
   rd = 0;
 }
 // When a different CSR instruction is used, the update of mstatus and the test
 // for whether side-effects should occur are modified accordingly.
 // When a different privileges xnxti CSR is accessed then clic.priv is compared
with
 // the corresponding privilege and xstatus, xintstatus.xil, xcause.exccode are
the
 // corresponding privileges CSRs.
```

✎ Vertical interrupts to different privilege modes will be taken preemptively by the hardware, so `xnxti` effectively only ever handles the next interrupt in the same privilege mode.

In original basic mode, reads of `xnxti` return 0, updates to `xstatus` proceed as in CLIC mode, but updates to `xintstatus` and `xcause` do not take effect.

5.9. New Interrupt Status (`xintstatus`) CSRs

A new M-mode CSR, `mintstatus`, holds the active interrupt level for each supported privilege mode. These fields are read-only. The primary reason to expose these fields is to support debug.

```
mintstatus fields
31:24 mil
23:16 (reserved) # To follow pattern of others.
15: 8 sil
 7: 0 uil
```

Corresponding supervisor mode, `sintstatus`, and user, `uintstatus`, provide restricted views of mintstatus.

```
sintstatus fields
31:16 (reserved)
15: 8 sil
 7: 0 uil
```

```
uintstatus fields
31: 8 (reserved)
 7: 0 uil
```

The `xintstatus` registers are accessible in original basic mode for system that support both modes.

5.10. New Interrupt-Level Threshold (`xintthresh`) CSRs

The interrupt-level threshold (`xintthresh`) is a new read-write CSR, which holds an 8-bit field (`th`) for the threshold level of the associated privilege mode. The `th` field is held in the least-significant 8 bits of the CSR, and zero should be written to the upper bits.

A typical usage of the interrupt-level threshold is for implementing critical sections. The current handler can temporarily raise its effective interrupt level to implement a critical section among a subset of levels, while still allowing higher interrupt levels to preempt.

The current hart's effective interrupt level would then be: effective_level = max(`xintstatus.xil`, `xintthresh.th`)

The max is used to prevent a hart from dropping below its original level which would break assumptions in design, and also makes it simple for software to remove threshold without knowing its own level by simply writing zero.

The interrupt-level threshold is only valid when running in associated privilege mode and not in other modes. This is because interrupts for lower privilege modes are always disabled, whereas interrupts for higher privilege modes are always enabled. For example, machine-mode interrupts will not be masked by machine-mode threshold setting when running in user mode. This is analogous to how mstatus.mie does not mask machine-mode interrupts when running in lower privilege modes.

> This behavior significantly reduces the hardware cost because it only needs to select one global maximum interrupt and compare with the threshold of the associated privilege mode (while ignoring thresholds in other modes). Otherwise, hardware would have to select multiple maximum interrupts (one per privilege mode), compare and qualify with their associated thresholds, then pick a qualified maximum interrupt with the highest privilege mode.

5.11. New CLIC Base (`mclicbase`) CSR

The machine mode `mclicbase` CSR is an XLEN-bit read-only register providing the base address of CLIC memory mapped registers. Its value should be configured or set up at the platform level to indicate the starting address of CLIC memory mapped registers.

Since the CLIC memory map must be aligned at a 4KiB boundary, the `mclicbase` CSR has its 12 least-

significant bits hardwired to zero. It is used to inform software about the location of CLIC memory mappped registers.

> The `mclicbase` register is likely to be replaced by the general discovery mechanism that is in development.

Chapter 6. CLIC Parameters

Name	Value Range	Description
CLICANDBASIC also?	0-1	Implements original basic mode
CLICPRIVMODES	1-3	Number privilege modes: 1=M, 2=M/U, 3=M/S/U
CLICLEVELS including 0	2-256	Number of interrupt levels
NUM_INTERRUPT	4-4096	Always has MSIP, MTIP, MEIP, CSIP
CLICMAXID	12-4095	Largest interrupt ID
CLICINTCTLBITS	0-8	Number of bits implemented in clicintctl[i]
CLICCFGMBITS	0-ceil(lg2(CLICPRIVMODES))	Number of bits implemented for cliccfg.nmbits
CLICCFGLBITS	0-ceil(lg2(CLICLEVELS))	Number of bits implemented for cliccfg.nlbits
CLICSELHVEC supported?	0-1	Selective hardware vectoring
CLICMTVECALIGN	6-13	Number of hardwired-zero least significant bits in mtvec address.
CLICXNXTI	0-1	Has xnxti CSR implemented?
CLICXCSW	0-1	Has xscratchcsw/xscratchcswl implemented?

> These parameters are likely to be available by the general discovery mechanism that is in development.

Chapter 7. CLIC Reset Behavior

In general in RISC-V, mandatory reset state is minimized but platform specifications or company policy might add additional reset requirements. Since the general privileged architecture states that mstatus.mie is reset to zero, interrupts will not be enabled coming out of reset.

> For an S-mode execution environment, the EEI should specify that status.sie is also reset on entry. It is then responsibility of the execution environment to ensure that is true before beginning execution in S-mode. Similarly for other lower-mode execution environments.

7.1. CLIC mandatory reset state

`xintstatus.xil` fields reset to 0. Interrupt level 0 corresponds to regular execution outside of an interrupt handler.

The reset behavior of other fields is platform-specific.

Chapter 8. CLIC Interrupt Operation

This section describes the operation of CLIC interrupts.

8.1. General Interrupt Overview

At any time, a hart is running in some privilege mode with some interrupt level. The hart's privilege mode is held internally but is not visible to software running on a hart (to avoid virtualization holes), but the current interrupt level is made visible in the `xintstatus` register.

Within a privilege mode `x`, if the associated global interrupt-enable `xie` is clear, then no interrupts will be taken in that privilege mode, but a pending-enabled interrupt in a higher privilege mode will preempt current execution. If `xie` is set, then pending-enabled interrupts at a higher interrupt level in the same privilege mode will preempt current execution and run the interrupt handler for the higher interrupt level.

As with the existing RISC-V mechanism, when an interrupt or synchronous exception is taken, the privilege mode and interrupt level are modified to reflect the new privilege mode and interrupt level. The global interrupt-enable bit of the handler's privilege mode is cleared, to prevent preemption by higher-level interrupts in the same privilege mode.

The overall behavior is summarized in the following table: the Current `p/ie/il` fields represent the current privilege mode `P` (not software visible), interrupt enable `ie` = (`xstatus.xie` & `clicintie[i]`) and interrupt level `L` = max(`xintstatus.xil`, `xintthresh.th`); the CLIC `priv,level`, and `id` fields represent the highest-ranked interrupt currently present in the CLIC with `nP` representing the new privilege mode, `nL` representing the new interrupt level, and `id` representing the interrupt's id; Current' shows the `p/ie/il` context in the handler's privilege mode; `pc` represents the program counter with `V` representing the result of any hardware vectoring; `cde` represents the `xcause exccode` field; while the Previous `pp/il/ie/epc` columns represent previous context fields in `xcause` and `xepc`.

```
Current |      CLIC          |->     Current'       Previous
p/ie/il | priv level   id    |->   p/ie/il  pc  cde  pp/il/ie epc
P  ?  ? | nP<P     ?        ? |->   -  -  -   -   -   -  -  -  -   # Interrupt ignored
P  0  ? | nP=P     ?        ? |->   -  -  -   -   -   -  -  -  -   # Interrupts disabled
P  1  ? | nP=P     0        ? |->   -  -  -   -   -   -  -  -  -   # No interrupt
P  1  L | nP=P     0<nL<=L  ? |->   -  -  -   -   -   -  -  -  -   # Interrupt ignored
P  1  L | nP=P     L<nL    id |->   P  0  nL  V   id  P  L  1  pc # Horizontal interrupt taken
P  ?  ? | nP>P     0        ? |->   -  -  -   -   -   -  -  -  -   # No interrupt
P  e  L | nP>P     0<nL    id |->   nP 0  nL  V   id  P  L  e  pc # Vertical interrupt taken
```

8.2. Critical Sections in Interrupt Handlers

To implement a critical section between interrupt handlers at different levels in the same privilege mode, an interrupt handler at any interrupt level can temporarily raise the interrupt-level threshold (`mintthresh.th`) to mask a subset of levels, while still allowing higher interrupt levels to preempt. Alternatively, although not recommended due to worse system impacts, it can clear the mode's global interrupt-enable bit (`xie`) to prevent any interrupts with the same privilege mode from being taken.

8.3. Responses to Wait for Interrupt (WFI)

In CLIC, similar to original mode, Wait for Interrupt instruction (WFI) is unaffected by the global interrupt-

enable bits in `xstatus.xie` but should honor `clicintie[i]` and `xintthresh`. In other words, WFI ignores the actual value in the global interrupt-enable `xstatus.xie` (by treating as if `xstatus.xie` were always enabled).

> WFI can be a NOP and not actually go to sleep. In addition, implementations can wake up WFI for any other reason.

WFI is required to wake up due to an interrupt i if interrupt i

- has a higher privilege mode than the current privilege mode and
- the interrupt priority reduction tree selects interrupt *i* as the maximum across all pending-and-enabled interrupts and
- the interrupt *i* level is not equal to 0.

WFI is required to wake up due to an interrupt i if interrupt i

- has the same privilege mode as the current privilege mode and
- the interrupt priority reduction tree selects interrupt *i* as the maximum across all pending-and-enabled interrupts and
- the interrupt *i* level is greater than max(`xintstatus.xil`, `xintthresh.th`)

WFI is required to wake up due to an interrupt i if interrupt i

- has a lower privilege mode than the current privilege mode and
- the interrupt priority reduction tree selects interrupt *i* as the maximum across all pending-and-enabled interrupts and
- the interrupt *i* level is not equal to 0.

> Interrupt *i* level is a function of CLICINTCTLBITS, `cliccfg.nlbits`, and `clicintctl[i]`. If CLICINTCTLBITS is 8 and `cliccfg.nlbits` = 8, it is possible to set `clicintctl[i]` to 0. Level 0 will behave as a locally disabled interrupt but can still mask lower-mode interrupts. For example, if there is a non-zero level supervisor interrupt pending and a level-zero machine interrupt pending, the machine interrupt will be the global maximum across all pending-and-enabled interrupts but interrupt level 0 implies no interrupt. So programming `clicintctl[i]` to 0 should not be used to disable interrupts. `clicintie[i]` should be used instead.

> `xintthresh` only applies to the current privilege mode. There is a proposal to add a new WFMI instruction ("wait for mode's interrupts") to the privilege specification. This instruction only has to wakeup for pending-and-enabled interrupts in the current mode, and is not required to wakeup for pending-and-enabled interrupts in lower privilege modes. Pending-enabled higher privilege-mode interrupts will interrupt/wakeup as usual.

8.4. Synchronous Exception Handling

Horizontal synchronous exception traps, which stay within a privilege mode, are serviced with the same interrupt level as the instruction that raised the exception.

Vertical synchronous exception traps, which are serviced at a higher privilege mode, are taken at interrupt level 0 in the higher privilege mode.

> Traps should be avoided at any time when `xepc`/`xcause` are live because these CSRs will be overwritten. Software should try to back them up if needed.

8.5. Returns from Handlers

The regular `xret` instructions are used to return from handlers in privilege mode x. Execution continues at the saved privilege mode `xcause.xpp`, at PC `xepc`, with interrupt level `xcause.xpil`, and with the global interrupt enable for the restored mode as `xcause.xpie`.

The `xret` instruction does not modify the `xcause.xpil` field in `xcause`. The `xcause.xpp` and `xcause.xpie` fields are modified following the behavior previously defined for `xstatus.xpp` and `xstatus.xpie` respectively.

Chapter 9. Interrupt Handling Software

9.1. Interrupt Stack Software Conventions

The CLIC supports multiple nested interrupt handlers, and each handler requires some working registers. To make registers available, each handler typically saves and restores registers from the interrupted context on a memory-resident stack. In addition, the memory-resident stack is used to hold other interrupted context information, such as `xepc` and `xcause`, which are required by the `xret` instruction.

The standard RISC-V ABI convention is that stacks grow downwards, and that memory addresses below the current stack pointer can be dynamically altered by another agent, such as an interrupt handler.

When interrupts are taken horizontally within the same privilege mode, the interrupt handler may be able to use the same stack as the interrupted thread, by allocating a new stack frame below the current stack pointer.

When interrupts are taken vertically into a higher privilege mode, the stack pointer must be swapped to a stack within the higher privilege mode to avoid a security hole. The `xscratch` registers can be used to hold the stack pointer of a higher-privilege mode while lower-privilege code is executing, or `xscratch` can be used to point to more extensive thread-local context that might contain a stack pointer.

9.2. Inline Interrupt Handlers and "Interrupt Attribute" for C

Inline interrupt handlers are small leaf functions that handle simple interrupts. To provide easy C coding for inline interrupt handlers, while reducing register save/restore overhead, we use standard interrupt attributes, which have the following syntax:

```
  /* Small ISR to poke device to clear interrupt and increment in-memory counter.
*/
  void __attribute__ ((interrupt))
  foo (void)
  {
    extern volatile int INTERRUPT_FLAG;
    INTERRUPT_FLAG = 0;
    extern volatile int COUNTER;
#ifdef __riscv_atomic
    __atomic_fetch_add (&COUNTER, 1, __ATOMIC_RELAXED);
#else
    COUNTER++;
#endif
  }
```

The attribute tells the C compiler to use callee-save for all registers, so the handler has to "pay as it goes" to use registers, and only save the full caller-save set if it makes a nested regular C call. The attribute also tells the C compiler to align the function entry point on an 8-byte boundary.

```
    .align 3
      # Inline non-preemptible interrupt handler.
      # Only safe for horizontal interrupts.
    foo:
      addi sp, sp, -FRAMESIZE       # Create a frame on stack.
      sw a0, OFFSET(sp)             # Save working register.
      sw x0, INTERRUPT_FLAG, a0     # Clear interrupt flag.
      sw a1, OFFSET(sp)             # Save working register.
      la a0, COUNTER                # Get counter address.
      li a1, 1
      amoadd.w x0, (a0), a1         # Increment counter in memory.
      lw a1, OFFSET(sp)             # Restore registers.
      lw a0, OFFSET(sp)
      addi sp, sp, FRAMESIZE        # Free stack frame.
      mret                          # Return from handler using saved mepc.
```

With hardware vectoring, inline interrupt handlers can provide very rapid response for small tasks.

The above entire handler executes in 13 instructions. The `INTERRUPT_FLAG` store and the `la` require two instructions each to build up a global address. A simple pipeline would encounter two pipeline flushes (on entry and on exit), plus the cycles taken to fetch the hardware vector entry.

These inline handlers can be used with the original basic mode as well as the new CLIC.

To take advantage of hardware preemption in the new CLIC, inline handlers must save and restore `xepc` and `xcause` before enabling interrupts:

```
    .align 3
        # Inline preemptible interuppt handler.
        # Only safe for horizontal interrupts.
    foo:
        #----- Interrupts disabled on entry ---#
        addi sp, sp, -FRAMESIZE      # Create a frame on stack.
        sw a0, OFFSET(sp)            # Save working register.
        csrr a0, mcause              # Read cause.
        sw a1, OFFSET(sp)            # Save working register.
        csrr a1, mepc                # Read epc.
        csrrsi x0, mstatus, MIE      # Enable interrupts.
        #----- Interrupts enabled ---------#
        sw a0, OFFSET(sp)            # Save cause on stack.
        sw x0, INTERRUPT_FLAG, a0    # Clear interrupt flag.
        sw a1, OFFSET(sp)            # Save epc on stack.
        la a0, COUNTER               # Get counter address.
        li a1, 1
        amoadd.w x0, (a0), a1        # Increment counter in memory.
        lw a1, OFFSET(sp)            # Restore epc
        lw a0, OFFSET(sp)            # Restore cause
        csrrci x0, mstatus, MIE      # Disable interrupts.
        #----- Interrupts disabled  ---------#
        csrw mepc, a1                # Put epc back.
        lw a1, OFFSET(sp)            # Restore a1.
        csrw mcause, a0              # Put cause back.
        lw s0, OFFSET(sp)            # Restore s0.
        addi sp, sp, FRAMESIZE       # Free stack frame.
        mret                         # Return from handler.
        #----------------------------------#
```

This version requires 10 more instructions, but reduces the time a preempting interrupt has to wait from a 13-instruction window to a 6-instruction window (the instruction that disables interrupts can be preempted before committing).

This form cannot be used with the existing original basic scheme, unless the original interrupt pending signal is cleared before re-enabling interrupts.

Chapter 10. Calling C-ABI Functions as Interrupt Handlers

An alternative model is where all interrupt handler routines use the standard C ABI. In this case, the CLIC would use no hardware vectoring for the C ABI handlers and instead use a common software trampoline, which uses the `xnxti` instruction to obtain the trap-handler address. The code sequence below is annotated with an explanation of its operation.

10.1. C-ABI Trampoline Code

```
  # Example Unix C ABI interrupt trampoline.
  # Only safe for horizontal interrupts.
  # FRAMESIZE should be defined appropriately to hold saved context with ABI-
specified alignment.
  # OFFSET should be replaced with individual stack frame locations.
  # Register save/restore pseudo-code should be expanded to individual
instructions.

  irq_enter:
  #----Interrupts disabled for 7 + SREGS instructions, where SREGS is number of
registers saved. ①
      addi sp, sp, -FRAMESIZE # Allocate space on stack. ②
      sw a1, OFFSET(sp)        # Save a1.
      csrr a1, mcause          # Get mcause of interrupted context.
      sw a0, OFFSET(sp)        # Save a0.
      csrr a0, mepc            # Get mepc of interrupt context.
      bgez a1, handle_exc      # Handle synchronous exception. ③
      sw a0, OFFSET(sp)        # Save mepc.
      sw a1, OFFSET(sp)        # Save mcause of interrupted context.
      sw a2-a7, OFFSET(sp)     # Save other argument registers.
      sw t0-t6, OFFSET(sp)     # Save temporaries.
      sw ra, OFFSET(sp)        # 1 return address ⑤
      csrrsi a0, mnxti, MIE    # Get highest current interrupt and enable interrupts.
                               # Will return original interrupt if no others appear.
  ⑥
    #----Interrupts enabled --------------------⑦
      beqz a0, exit            # Check if original interrupt vanished. ⑧

  service_loop:               # 5 instructions in pending-interrupt service loop.
      lw a1, (a0)              # Indirect into handler vector table for function
pointer. ⑨
      csrrsi x0, mstatus, MIE # Ensure interrupts enabled. ⑩

      jalr a1                  # Call C ABI Routine, a0 has interrupt ID encoded. ⑪
                               # Routine must clear down interrupt in CLIC.
      csrrsi a0, mnxti, MIE    # Claim any pending interrupt at level > mcause.pil ⑫
```

```
    bnez a0, service_loop    # Loop to service any interrupt. ⑬


  #--- Restore ABI registers with interrupts enabled -⑭
    lw ra, OFFSET(sp)        # Restore return address
    lw t0-t6, OFFSET(sp)     # Restore temporaries.
    lw a2-a7, OFFSET(sp)     # Restore other arguments.
    lw a1, OFFSET(sp)        # Get saved mcause,
  exit:                      # Fast exit point.
    lw a0, OFFSET(sp)        # Get saved mepc.


    csrrci x0, mstatus, MIE # Disable interrupts ⑮
  #---- Critical section with interrupts disabled ----------------------
    csrw mcause, a1          # Restore previous context.

    lw a1, OFFSET(sp)        # Restore original a1 value.
    csrw mepc, a0            # Restore previous context.

    csrrci a0, mnxti, MIE    # Claim highest current interrupt. ⑯
    bnez a0, service_loop    # Go around if new interrupt.

    lw a0, OFFSET(sp)        # Restore original a0 value.
    addi sp, sp, FRAMESIZE   # Reclaim stack space.
    mret                     # Return from interrupt.
  #-------------------------------------------------------------------
  #-------------------------------------------------------------------
  handle_exc:
    # ...
    # Perform exception processing with interrupts disabled ④
    # ...
    addi sp, sp, FRAMESIZE   # Reclaim stack space.
    mret # Return from exception
  #-------------------------------------------------------------------
```

① An initial interrupt (II) causes entry to the handler with interrupts disabled, and xepc and xcause CSRs hold values representing the original interrupted context (OIC), including the PC in xepc, the privilege mode in xpp (visible in both xcause and xstatus), the interrupt level in xpil (in xcause) and the interrupt enable state in xpie (visible in both xcause and xstatus). The xcause CSR and the xintstatus CSRs additionally hold information on the interrupt to be handled, including exccode in xcause and xil in xintstatus.

② The interrupt trampoline needs sufficient space to store the OIC's caller-save registers as well as its epc and cause values, which are saved in a frame on the memory stack to support preemption. This routine is M-mode only so does not need to consider swapping stacks from other privilege modes. A simple constant bump of the stack pointer sp is sufficient to provide space to store the OIC.

③ The trap handler could have been entered by a synchronous exception instead of an interrupt, which can be determined by examining the sign bit of the returned xcause value. If the trap was for an exception (sign bit zero), the code jumps to exception handler code while keeping interrupts disabled.

④ The exception handler code is located here out of line to reduce performance impact on interrupts. The main body of the trampoline only handles interrupts.

⑤ If this was an interrupt, the trampoline entry code continues to save all the caller-save registers to the stack. This is done with interrupts disabled, as even if an interrupt arrived with a higher interrupt level it would still require all registers to be saved.

⑥ When `xnxti` is read here, the interrupt inputs to the CLIC might have changed from the time the handler was initially entered. The return value of `xnxti`, which holds a pointer to an entry in the trap vector table, is saved in register `a0` so it can be passed as the first argument to the software-vectored interrupt handler, where it can be used to reconstruct the original interrupt id in the case where multiple vector entries use a common handler. There are multiple cases to consider, all of which are handled correctly by the definition of `xnxti`:

- The II is still the ranking interrupt (no change). In this case, as the level of the II will still be higher than `pil` from the OIC, `xil` and `exccode` will be rewritten with the same value that they already had (effectively unchanged), and `xnxti` will return the table entry for the II.

- The II has been superceded by a higher-level non-SHV interrupt. In this case, `xil` will be set to the new higher interrupt level, `exccode` will be updated to the new interrupt id, and `xnxti` will return the vector table entry for the new higher-level interrupt. The OIC is not disturbed, retaining the original `epc` and the original `pil`. This case reduces latency to service a more-important interrupt that arrives after the state-save sequence was begun for the less-important II. The II, if still pending-enabled, will be serviced sometime after the higher-level interrupt as described below.

- The II has been superceded by a higher-priority non-SHV interrupt at the same level. This operates similarly to the previous case, with `exccode` updated to the new interrupt id. Because the lower-priority interrupt had not begun to run its service routine, this optimization preserves the property that interrupt handlers at the same interrupt level but different priorities execute atomically with respect to each other (i.e., they do not preempt each other).

- The II has disappeared and a lower-ranked non-SHV interrupt, which has interrupt level greater than the OIC's `pil` is present in CLIC. In this case, the `xil` of the handler will be reduced to the lower-ranked interrupt's level, `exccode` will be updated with the new interrupt id, and `xnxti` will return a pointer to the appropriate handler in table. In this case, the new lower-ranked interrupt would still have caused the original context to have been interrupted to run the handler, and the disappearing II has simply caused the lower-ranked interrupt's entry and state-save sequence to begin earlier.

- The II has disappeared and either there is no current interrupt from the CLIC, or the current ranking interrupt is a non-SHV interrupt with level lower than `xpil`. In this case, the `xil` and `exccode` are not updated, and 0 is returned by `xnxti`. The following trampoline code will then not fetch a vector from the table, and instead just restore the OIC context and `mret` back to it. This preserves the property that the OIC completes execution before servicing any new interrupt with a lower or equal interrupt level.

- The II has been superceded by a higher-level SHV interrupt. In this case, the `xil` and `exccode` are not updated, and 0 is returned by `xnxti`. Once interrupts are reenabled for the following instruction, the hart will preempt the current handler and execute the vectored interrupt at a higher interrupt level using the function pointer stored in the vector table.

⑦ Interrupts are now enabled. If a higher-level SHV interrupt had arrived while interrupts were disabled, then the current handler will be preempted and execution starts at the SHV handler address. If a non-vectored higher-level interrupt arrives now, it will also preempt the current handler and begin a nested state-save sequence at the handler entry point `irq_enter`.

⑧ The branch checks if the II disappeared or if a higher priority SHV at the same level appeared, in which case the current handler returns to the OIC. As most registers have not been touched, the routine can skip past most of the register restore code. This preserves the property that interrupts (SHV or non-SHV) at the same level do not preempt each other.

⑨ The value returned by `xnxti` is used to index the vector table and return the function pointer.

⑩ This `csrrsi` instruction enables interrupts and is redundant when proceeding sequentially from the first

xnxti read (6) or if looping back from the end of the service_loop (13). However, it is required on the backward path from (16) to re-enable interrupts to allow preemption. It is scheduled after the table lookup to use what will often be a load-use delay slot.

⑪ The jalr instruction actually calls the C ABI function that implements the handler. Interrupts are enabled at this point, so the C function can be preempted at any time by an interrupt with a higher level than current xil.

⑫ Once the handler returns, another read of xnxti checks if there are any more interrupts to service. Interrupts remain enabled. The csrrsi includes a redundant set of the xie interrupt enable to force the CSR instruction to update CSR state. Only non-SHV interrupts with a level greater than pil will be serviced in this loop. Note that xil can decrease from its current value on the xnxti read. xil should not increase in this code, as interrupts are enabled here and if a higher-level interrupt was ready, it should have preempted this instruction.

⑬ If there was another appropriate interrupt to service, the code loops back to perform the next handler call. The service_loop only contains 5 instructions, allowing multiple back-back interrupts to be handled without saving and restoring contexts. On a simple pipeline with a one-cycle load-use penalty, single-cycle CSR access, and a one-cycle taken-branch penalty, the service loop can initiate a new interrupt service with only 7 clock cycles of overhead per handler call.

⑭ This instruction sequence restores the OIC. Interrupts are still enabled, so preemption is allowed during this restore.

⑮ Interrupts are disabled for the final steps of restoring the OIC, which requires loading mcause and mepc from the stacked values, and recovering the final register values from the OIC.

⑯ A final read of xnxti is performed before returning, to reduce the maximum interrupt latency. If a suitable interrupt arrives, it can be serviced without saving context. The csrrci instruction includes a redundant clear of the interrupt enable bit to ensure the CSR state updates occur. Interrupts must stay disabled until after the following branch to maintain the critical section used to restore the OIC in the case that there is no interrupt to service.

The following table summarizes the machine state changes that occur at the first xnxti:

```
IC    at entry |->           |       at first nxti (6)
il     CLIC                  |    CLIC
    level id V |->  mil code | level id V     |-> mil code rd
p    e<=p  ? ? |->           |                            # Shouldn't happen
p    e>p   i 0 |->   e    i  |  f>p  j 0    |-> f    j   T # Same or superceded interrupt
p    e>p   i 0 |->   e    i  |  f>p  j 1    |-> e    i   0 # Ignore vectored interrupt
p    e>p   i 0 |->   e    i  |  f<=p j ?    |-> e    i   0 # Interrupt disappeared
p    e>p   i 1 |->   e    i  |                            # Won't be in trampoline
```

10.2. Revised C-ABI for Embedded RISC-V

The overhead to save and restore registers in the interrupt trampoline can be reduced with a new embedded ABI that reduces the number of caller-save registers. Work is underway to define such an ABI, but it is likely to require around 7 integer registers to be saved/restored instead of 16 in the standard Unix ABI.

This will result in 18 instructions executed in the trampoline code before arriving at the correct handler function, of which 9 are stores (saving 7 registers plus 2 words for xepc and xcause).

10.3. Analysis of Worst-Case Interrupt Latencies for C-ABI Trampoline

The following analysis assumes a system with M-mode only and a new embedded ABI requiring 7 caller-save registers to be saved and restored. For cycle timings, we assume a simple 3-stage pipeline that has a one-cycle taken-branch or pipeline flush penalty, a one-cycle load-use delay, and single-cycle CSR access. This simple model ignores effects from contention in shared memory structures, or pipeline hazards from continuing long-latency operations in the interrupted code.

There are several cases to consider for the worst-case latency for a C-ABI higher-level interrupt handler that preempts lower-level code.

If an interrupt arrives while interrupts are enabled, either inside or outside of a current handler, the hart will jump directly to `irq_enter` at the new interrupt level. The system must flush the execution pipeline and then execute 18 instructions, the last of which is the `jalr` that calls the handler function. These 18 instructions execute in 20 cycles using the simple pipeline model.

When interrupts are disabled, the arriving preempting handler could be delayed. If the preempting interrupt arrives while interrupts are disabled during the initial entry sequence (1)--(6), there will be no additional delay as the first `xnxti` instruction (6) will cause the higher-level interrupt handler to be invoked, replacing the original interrupt cause.

If the preempting interrupt arrives after interrupts are disabled (15) but before `xnxti` is read (16), then the trampoline will observe the new interrupt during execution of the `xnxti` read (16), and take a short branch back to the `service_loop`, which is lower latency than the interrupt-disabled case.

If the preempting interrupt arrives after the read of `xnxti` commits (16), then the interrupt has to wait an additional 4 instructions until the `mret` reenables interrupts, at which point the interrupt will be taken and the handler entered at `irq_enter`. In the simple pipeline model, `mret` adds an additional pipeline flush cycle, so the preemption latency is 20+5 cycles, which represents the worst-case for a preempting C-ABI interrupt handler.

Chapter 11. Interrupt-Driven C-ABI Model

For many embedded systems, after initialization, essentially all code is run in response to an interrupt, interrupt levels are used to prioritize execution of different tasks, and the hart should sleep inbetween interrupt events to save energy.

The following code can be used as the background code that runs at interrupt level 0 and which when there is no active work to do, puts the hart to sleep with no active context, waiting for an interrupt using the `wfi` instruction. The code is entered at the `enter_loop` location and never returns directly.

```
    # Source code for interrupt-driven model background code.
sleep:
    csrrci x0, mstatus, MIE # Disable interrupts.   ①
    wfi                     # Hart waits for next interrupt event.
    csrrsi a0, mnxti, MIE   # Gather interrupt details, and enable interrupts. ②
    beqz a0, sleep          # Go back to sleep if no interrupt (will be preempted
if SHV). ③


service_loop: ④
    lw a1, (a0)             # Get handler address.
    csrrsi x0, mstatus, MIE # Enable interrupts
    jalr a1                 # Call C-ABI handler routine
    csrrsi a0, mnxti, MIE   # Claim any pending interrupt at level > 0
    bnez a0, service_loop   # Loop to service any interrupt.


    # This is also entry point to begin sleeping.
enter_sleep: ⑤
    la a0, sleep
    csrci x0, mstatus, MIE  # Disable interrupts.
    #--- Interrupts disabled
    csrw mepc, a0           # Initialize mepc to point to sleep
    li a0, (MMODE)<<PP|(0)<<PIL|(1)<<PIE
    csrw mcause, a0         # Initialize mcause to have pp=M, pil=0, pie=1
    mret                    # Jump to sleep at level 0 with interrupts enabled.
    #--- Interrupts enabled
```

① The `sleep` loop is used to stall the hart while waiting for work and is always entered at interrupt level 0. Interrupts are disabled, then a `wfi` is executed. The `wfi` will stall the hart until some event occurs. When an event, including an interrupt occurs, the `wfi` retires. Because interrupts are disabled, the hart does not jump to an interrupt handler but instead executes the next instruction, avoiding context save/restore overhead.

② The read of `xnxti` will determine if any non-SHV interrupt is available, and if so return a pointer to the table entry. Interrupts are enabled by this instruction to allow SHV interrupts to be taken via preemption.

③ The value in `a0` checked by the branch can be zero for two reasons. Either there was no interrupt detected or an SHV interrupt was detected. If there was no interrupt, the branch loops back to put the hart to sleep. Interrupts are enabled, so any SHV interrupt (which all have higher interrupt level than the current interrupt level of 0) will preempt the branch's execution and call the SHV handler. Once the SHV handler returns, the branch will resume and cause execution to return back to the `sleep_loop`.

④ The service loop is identical to that in the C-ABI interrupt handler, except that the previous interrupt level is 0, so all pending interrupts will be serviced in the loop before the loop exits. Interrupts are enabled, so preemption is allowed for both C-ABI trampoline and SHV interrupts. When an SHV interrupt at the same or lower interrupt level is the next to be serviced, the `xnxti` instruction will return 0 causing execution to drop out of the loop. The following code will reinitialize the hart's interrupt level to 0, and disable interrupts for one instruction, to ensure the SHV interrupt will be taken.

⑤ This code initializes `mepc` and `mcause` then uses an `mret` to jump to the `sleep` loop while simultaneously resetting interrupt level to 0 and enabling interrupts. This is also the entry point to initiate interrupt-driven execution. Interrupts are enabled to allow SHV interrupts to preempt execution on the first instruction in `sleep` (which disables interrupts again).

This code does not increase worst-case interrupt latency over that of the C-ABI trampoline.

Chapter 12. Alternate Interrupt Models for Software Vectoring

Platforms may only implement non-vectored CLIC mode without selective hardware vectoring (`cliccfg.nvbits=0`), in which case, hardware vectoring can be emulated by a single software trampoline present at `NBASE` using the separate vector table address in `xtvt`. There are several different software approaches possible, depending on system requirements and constraints, as detailed in following subsections.

12.1. gp Trampoline to Inline Interrupt Handlers in Single Privilege Mode

Where interrupts are known to be generated and handled in a single privilege mode (i.e., M-mode only systems, or U-mode interrupt handlers), a three-instruction sequence using the `gp` register to hold the handler address can be used to indirect to an inline interrupt handler of the type described in Inlines.

```
    # Software-vectored interrupt servicing.
    # Only safe for horizontal interrupts.
    # Must be placed three instructions back from gp.
irq_enter:
    csrrci gp, mnxti, MIE    # Overwrite gp, keep interrupts disabled.
    beqz gp, handle_exc      # Encountered exception.
    jalr gp, gp              # Recreate gp and jump to handler.
gp:                          # Must be right before system's gp location.
    # ... gp data section

    # Must be within range of beqz instruction.
handle_exc:
    # Has to recreate gp.
```

The three-instruction sequence relies on the `jalr` instruction recreating the value in the `gp` register, which is a known constant pointing into the middle of the global data area, by placing the `jalr` directly before the `gp` location in memory. The routine jumped to by the `jalr` does not return via a `j ra` but instead ends with an `mret`.

> This constraint on memory layout might not always be possible, particularly if the system does not allow placing executable memory right next to read-write memory, for example if the system does not allow a protection boundary to be placed at 'gp' and if executable code must not be writeable.

The code can be used with preemptible inline interrupt handlers.

12.2. Trampoline for Preemptible Inline Handlers

This section describes a more general software-trampoline scheme for calling preemptible inline handlers, which factors out the `xepc`/`xcause` save code into the trampoline, and which uses a different interrupt handler calling convention.

The interrupt handlers for this scheme have a calling convention where there is one caller-save argument register a0 that passes in the handler address to distinguish different interrupt inputs, and one temporary register a1 that is also caller-save. These two registers had to be saved already by the trampoline. All other registers are callee-save, except for the return address ra. The handler normally returns with a regular j ra.

```
    # Example handler with new calling convention.
    # Only safe for horizontal interrupts.
    # Handlers have two temporary registers available, a0, a1.
handler_example:
    sw x0, INTERRUPT_FLAG, a0     # Clear interrupt flag.
    la a0, COUNTER                # Get counter address.
    li a1, 1                      # Increment value.
    amoadd.w x0, (a0), a1         # Bump counter.
    j ra


    # Interrupt trampoline code.
irq_enter:
    #----- Interrupts disabled on entry ---#
    addi sp, sp, -FRAMESIZE       # Create a frame on stack.
    sw a0, OFFSET(sp)             # Save working register.
    csrr a0, mcause               # Read cause.
    bgez a0, handle_exc           # Handler exception.
    sw a1, OFFSET(sp)             # Save working register.
    csrr a1, mepc                 # Read epc.
    sw a0, OFFSET(sp)             # Save cause
    csrrsi a0, mnxti, MIE         # Get highest interrupt, enable interrupts.
    #----- Interrupts enabled ---------#
    beqz a0, exit
    sw a1, OFFSET(sp)             # Save epc.
    sw ra, OFFSET(sp)             # Save return address.

 irq_loop:
    lw a1, (a0)                   # Get function pointer.
    jalr a1                       # Call handler code.
    csrrsi a0, mnxti, MIE         # Get any next interrupt.
    bnez a0, irq_loop             # Service interrupt if any.

    lw ra, OFFSET(sp)             # Restore ra.
    lw a1, OFFSET(sp)             # Get epc.
exit:
    lw a0, OFFSET(sp)             # Get cause.
    csrrci x0, mstatus, MIE       # Disable interrupts.
    #----- Interrupts disabled  ---------#
    csrw mepc, a1                 # Put epc back.
    lw a1, OFFSET(sp)             # Restore a1.
    csrw mcause, a0               # Put cause back.
    lw a0, OFFSET(sp)             # Restore a0.
```

```
    addi sp, sp, FRAMESIZE         # Free stack frame.
    mret                           # Return from handler.
    #----------------------------------#


handle_exc:
    # ...
    # Handle exception with interrupts disabled.
    # ...
    addi sp, sp, FRAMESIZE  # Deallocate stack space
    mret                    # Return from handler.
    #----------------------------------#
```

This interrupt handler can be used together with the `wfi` sleep background routine shown above.

Chapter 13. Managing Interrupt Stacks Across Privilege Modes

Interrupt handlers need to have a place to save the previous context's state to provide working registers for the handler code. If a handler can be entered from a lower-privilege mode, a pointer to some safe memory for the context save must be swapped in at entry to the higher-privileged handler to avoid security holes. The RISC-V privileged architecture provides the `xscratch` register to hold this information for a higher-privilege mode while executing in a lower-privilege mode. For the following discussion and code examples, the assumption is that `xscratch` is used to hold the higher-privilege-mode stack pointer but other software conventions are possible (e.g., `xscratch` points to a thread context block).

Existing RISC-V ABIs allow addresses immediately below the stack pointer to be overwritten by interrupt service routines. The current stack pointer in `sp` (`x2`) should be swapped with `xscratch` whenever a handler is entered from a lower-privilege mode, but should not be swapped if entered from another handler in the same privilege mode, including when preempting an existing interrupt handler. At exit from a handler, the lower-privilege stack pointer should be swapped back in if transitioning back to the lower-privilege mode.

13.1. Software Privileged Stack Swap

In this convention, when code is running in a lower privilege mode, `xscratch` holds the stack pointer for the higher-privilege mode. When the higher-privilege mode is entered, `xscratch` is set to zero to signal to any preempting handlers that the stack pointer has already been swapped.

The old stack pointer is saved to new stack frame before new frame is created by bumping stack pointer, but this is done with interrupts disabled.

```
   # This code is out of line to reduce worst-case preemption latency.
 enter_M:
   sw sp, OFFSET-FRAMESIZE(sp)  # Save previous mscratch (M-mode sp)
   addi sp, sp, -FRAMESIZE      # Create a frame on stack.
   sw a0, OFFSET(sp)            # Save a register.
   csrrw a0, mscratch, 0        # Get previous sp, and zero mscratch.
   sw a0, OFFSET(sp)            # Save previous sp (U-mode sp)
   j  continue                  # Jump back into handler


 irq_enter:
   #----- Interrupts disabled on entry ---#
   csrrw sp, mscratch, sp       # Swap stack pointer and scratch.
   bnez sp, enter_M             # Check if entering M-mode
   csrrw sp, mscratch, sp       # Already in M-mode, so swap sp back.
   sw sp, OFFSET-FRAMESIZE(sp)  # Save previous sp to stack.
   addi sp, sp, -FRAMESIZE      # Create a frame on stack.
   sw x0, OFFSET(sp)            # Save previous mscratch to stack (was zero).
   sw a0, OFFSET(sp)            # Save a register.
 continue:
   csrr a0, mcause             # Read cause.
   bgez a0, handle_exc         # Handle exception.
   sw a1, OFFSET(sp)           # Save working register.
   csrr a1, mepc              # Read epc.
   sw a0, OFFSET(sp)           # Save cause
   csrrsi a0, mnxti, MIE       # Get highest interrupt, enable interrupts.
   #----- Interrupts enabled ---------#
   beqz a0, exit
   ...


   #---- Critical section with interrupts disabled ----------------------
     ...

   lw a0, OFFSET(sp)           # Get previous mscratch.
   csrw mscratch, a0          # Put back in mscratch.
   lw a0, OFFSET(sp)           # Restore original a0 value.
   lw sp, OFFSET(sp)           # Restore previous sp
   mret                        # Return from interrupt.
   #--------------------------------------------------------------------
```

This code can be used in a secure model where user-level code has one stack, and all interrupts and exceptions are handled on a second M-mode-only stack. In addition, background non-handler code in M-mode can either use the same M-mode stack as the interrupt handler, or a separate M-mode stack. The only difference is in the value held in xscratch while the M-mode background thread is running (either 0 to indicate use the existing stack pointer in sp or non-zero to indicate this stack pointer should be used in the handler.

13.2. Optional Scratch Swap CSR (xscratchcsw) for Multiple Privilege Modes

The above software scheme adds 7 instructions to the interrupt code path when preempting the same privilege mode, and adds an additional 6 instructions (13 total including two taken branches) for interrupts from a lower-privilege mode into a higher-privileged mode.

To accelerate interrupt handling with multiple privilege modes, a new CSR xscratchcsw can be defined for all but the lowest privilege mode to support conditional swapping of the xscratch register when transitioning between privilege modes. The CSR instruction is used once at the entry to a handler routine and once at handler exit, so only adds two instructions to the interrupt code path. Though designed to be used with csrrw instructions, these CSRs can be accessed with any CSR instruction.

For all CSR instructions accessing xscratchcsw, the value written into rd is either xscratch if xpp is different than the current privilege mode, or rs1 if xpp is the same as the current privilege mode. The xscratch register is only written if there is a privilege mode difference, and if so, it is written obeying the usual CSR read-modify-write conventions (e.g., swap/set/clear bits) using the original xscratch value as one source operand and the other source operand specified as usual in the instruction.

In practice, only the read-modify-write (swap/CSRRW) operation is useful for xscratchcsw. In contrast, the "pure read" and "pure write" operations are meaningless and not useful at all. The implementation of the non-CSRRW variants (when either rd or rs1 is x0 or when rs1 is an immediate operand) on xscratchcsw is not defined/reserved.

> This is different than a regular CSR instruction as the value returned is different from the value used in the read-modify-write operation.

> The CSR instructions are defined to always copy a result (xscratch or rs1) to the rd destination to simplify implementations using register renaming, and in normal use the instructions set both rs1 = sp and rd = sp.

```
csrrw rd, mscratchcsw, rs1


// Pseudocode operation.
if (mcause.mpp!=M-mode) then {
    t = rs1; rd = mscratch; mscratch = t;
} else {
    rd = rs1; // mscratch unchanged.
}


// Usual use: csrrw sp, mscratchcsw, sp
```

> To avoid virtualization holes, software cannot directly read the hart's current privilege mode. The swap instruction will trap if software tries to access a given mode's xscratchcsw CSR from a lesser-privileged mode, so the new CSR does not open a virtualization hole.

13.2.1. Stack Swap Example Code

Interrupt handlers running in the lowest privilege mode do not need to swap stack pointers, as they will only be entered by a horizontal interrupt from the same privilege mode. In systems with multiple privilege modes, handlers running in higher privilege modes must account for vertical interrupts taken from a lower privilege mode (in which case the stack pointer must be swapped) as well as horizontal interrupts from the same privilege mode.

```
 # Example of inline interrupt with stack swapping.
.align 3
foo:
   csrrw sp, mscratchcsw, sp    # Conditionally swap in stack pointer.
   addi sp, sp, -FRAMESIZE      # Create a frame on stack.
   sw s0, OFFSET(sp)            # Save working register.
   sw x0, INTERRUPT_FLAG, s0    # Clear interrupt flag.
   sw s1, OFFSET(sp)            # Save working register.
   la s0, COUNTER               # Get counter address.
   li s1, 1
   amoadd.w x0, (s0), s1        # Increment counter in memory.
   lw s1, OFFSET(sp)            # Restore registers.
   lw s0, OFFSET(sp)
   addi sp, sp, FRAMESIZE       # Free stack frame.
   csrrw sp, mscratchcsw, sp    # Conditionally swap out stack pointer.
   mret                         # Return from handler using saved mepc.
```

```
 # Example of inline preemptible interrupt with stack swapping.
.align 3
foo:
   #----- Interrupts disabled on entry ---#
   csrrw sp, mscratchcsw, sp    # Conditionally swap in stack pointer.
   addi sp, sp, -FRAMESIZE      # Create a frame on stack.
   sw s0, OFFSET(sp)            # Save working register.
   sw s1, OFFSET(sp)            # Save working register.
   csrr s0, mcause             # Read cause.
   csrr s1, mepc               # Read epc.
   csrrsi x0, mstatus, MIE     # Enable interrupts.
   #----- Interrupts enabled ---------#
   sw s0, OFFSET(sp)            # Save cause on stack.
   sw x0, INTERRUPT_FLAG, s0    # Clear interrupt flag.
   sw s1, OFFSET(sp)            # Save epc on stack.
   la s0, COUNTER              # Get counter address.
   li s1, 1
   amoadd.w x0, (s0), s1       # Increment counter in memory.
   lw s1, OFFSET(sp)            # Restore epc
   lw s0, OFFSET(sp)            # Restore cause
   #----- Interrupts disabled  ---------#
   csrrci x0, mstatus, MIE     # Disable interrupts.
   csrw mepc, s1               # Put epc back.
   csrw mcause, s0             # Put cause back.
   lw s1, OFFSET(sp)            # Restore s1.
   lw s0, OFFSET(sp)            # Restore s0.
   addi sp, sp, FRAMESIZE      # Free stack frame.
   csrrw sp, mscratchcsw, sp    # Conditionally swap out stack pointer.
   mret                        # Return from handler.
   #--------------------------------#
```

```
    # Example C-ABI interrupt trampoline with stack swapping.

    irq_enter:
    #----
      csrrw sp, mscratchcsw, sp # Conditionally swap in stack pointer.
      addi sp, sp, -FRAMESIZE   # Allocate space on stack.
      # ...
      # Everything else same as above.
      # ...
      addi sp, sp, FRAMESIZE    # Reclaim stack space.
      csrrw sp, mscratchcsw, sp # Conditionally swap back stack pointer.
      mret                      # Return from interrupt.
    #------------------------------------------------------------------------
    #------------------------------------------------------------------------
     handle_exc:
      # ...
      # Perform exception processing with interrupts disabled
      # ...
      addi sp, sp, FRAMESIZE    # Reclaim stack space.
      csrrw sp, mscratchcsw, sp # Conditionally swap back stack pointer.
      mret                      # Return from exception
    #------------------------------------------------------------------------
```

In all cases, conditionally swapping the stack to account for potential privilege-mode changes adds two extra instructions to all interrupt handlers.

Chapter 14. Separating Stack per Interrupt Level

Within a single privilege mode, it can be useful to separate interrupt handler tasks from application tasks to increase robustness, reduce space usage, and aid in system debugging. Interrupt handler tasks have non-zero interrupt levels, while application tasks have an interrupt level of zero.

14.1. Optional Scratch Swap CSR (xscratchcswl) for Interrupt Levels

A new xscratchcswl CSR is added to support faster swapping of the stack pointer between interrupt and non-interrupt code running in the same privilege mode.

```
csrrw rd, mscratchcswl, rs1


// Pseudocode operation.
if ( (mcause.pil==0) != (mintstatus.mil==0) ) then {
    t = rs1; rd = mscratch; mscratch = t;
} else {
    rd = rs1; // mscratch unchanged.
}


// Usual use: csrrw sp, mscratchcswl, sp
```

This new CSR operates similarly to xscratchcsw except that the swap condition is true when the interrupter and interruptee are not both application tasks or not both interrupt handlers.

Similar to xscratchcsw, only the read-modify-write (swap/CSRRW) operation is defined for xscratchcswl. The implementation of the non-CSRRW variants (when either rd or rs1 is x0 or when rs1 is an immediate operand) on xscratchcswl is not defined/reserved.

Chapter 15. CLIC Interrupt ID ordering recommendations

There are a few CLIC interrupt ID ordering recommendations as part of a profile and are not mandatory in all incarnations of the CLIC.

Original recommendation: The original basic mode interrupts retain their interrupt ID in CLIC mode. The `clicintctl` settings are now used to delegate these interrupts as required.

An additional CLIC software interrupt bit (csip) is provided. This is generally available for software use, but is usually used for the local background interrupt thread.

CLIC interrupt inputs are allocated IDs beginning at interrupt ID 16. Any fast local interrupts that would have been connected at interrupt ID 16 and above should now be mapped into corresponding inputs of the CLIC.

```
ID  Interrupt   Note

 0  usip        User software Interrupt
 1  ssip        Supervisor software Interrupt
 2  reserved
 3  msip        Machine software interrupt

 4  utip        User timer interrupt
 5  stip        Supervisor timer interrupt
 6  reserved
 7  mtip        Machine timer interrupt

 8  ueip        User external (PLIC) interrupt
 9  seip        Supervisor external (PLIC) interrupt
10  reserved
11  meip        Machine external (PLIC) interrupt

12  csip        CLIC software interrupt
13  reserved
14  reserved
15  reserved


16+ inputs     CLIC external inputs
```

M-mode only or M/U mode interrupt map recommendation: (in M-only or M/U system without N extension) 0: M-mode software interrupt 1: M-mode timer interrupt 2+: external (including legacy)

M-mode interrupt map in M/S/U system without N extension recommendation: 0: S-mode software interrupt 1: S-mode timer interrupt 2: M-mode software interrupt 3: M-mode timer interrupt 4+: external (including legacy)

Interrupt map with PLIC recommendation: 0: S-mode software interrupt 1: S-mode timer interrupt 2: S-mode external (PLIC) interrupt 3: M-mode software interrupt 4: M-mode timer interrupt 5: M-mode external (PLIC) interrupt 6: external

Appendix A: Appendix

A.1. Prototype DTS Entry

Modified from Example at elinux.org/Device_Tree_Usage

/dts-v1/;

/ { …

```
  interrupt-parent = <&intc>;
```

```
  ...
```

```
  intc: interrupt-controller@MCLICBASE {
      compatible = "xxx,xxx";
      reg = <MCLICBASE 0x5000 >;
      interrupt-controller;
      #interrupt-cells = <NUM_INTERRUPT>;
  };
```

```
  ...
```

};

Chapter 16. Bibliography

[CD3600] Interrupt and interrupt mask register with interrupt address based on interrupt number. Control Data 3600 Computer System Reference Manual, Chapter IV - Interrupt System bitsavers.org/pdf/cdc/3x00/48bit/3600/60021300E_3600_SysRef_Sep64.pdf

[BiiN] Interrupt preemption, checking pending interrupts before returning, adjusting current priority level (modpc). Chapter 12. Interrupts bitsavers.org/pdf/biin/BiiN_CPU_Architecture_Reference_Man_Jul88.pdf

Index