SiFive Interrupt Cookbook

Version 1.2

SiFive Interrupt Cookboook

Release Information

Version	Date	Changes
V1.0	December 5, 2019	• Initial release
V1.1	February 3, 2020	• Update small typos and vectored mode exception handling details
V1.2	February 5, 2020	• Fix mtvec alignment requirements

Contents

Chapter 1

SiFive Interrupt Cookbook

1.1 Introduction

Embedded systems rely heavily on handling interrupts which are asynchronous events designed to be managed by the CPU. SiFive core designs include options for a simple timer and software interrupt generator, a fully featured local interrupt controller, and optionally, a global interrupt controller. This document describes the features and configuration details of the available interrupt configurations offered by SiFive.

1.1.1 Terminology

Hardware Threads (HART) in SiFive Designs

As of this writing, all SiFive designed CPUs contain a single HART per core. Future products from SiFive may implement multi-hart designs. For simplicity, HART and CPU may be used interchangeably in this document as it relates to interrupts.

RISC-V Exception, Interrupt, Trap

The following terminology comes directly from *From The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2*:

- *We use the term **exception** to refer to an unusual condition occurring at run time associated with an **instruction** in the current RISC-V thread.*
- *We use the term **trap** to refer to the **synchronous transfer of control** to a trap handler caused by an exceptional condition occurring within a RISC-V thread. Trap handlers usually execute in a more privileged environment.*

- *We use the term **interrupt** to refer to an **external event that occurs asynchronously** to the current RISC-V thread. When an interrupt that must be serviced occurs, some instruction is selected to receive an interrupt exception and subsequently experiences a trap.*

> **Note**
>
> Our use of **exception** and **trap** matches that in the IEEE-754 floating-point standard.

Exception Example

The address of the data during a load instruction is not aligned correctly, so upon execution of this load instruction, the CPU will enter an exception handler and a *load address misaligned* exception code will appear in the `mcause` register as a result. In the exception handler, software will then need to determine the next course of action, since the misaligned load is not allowed by the design. See *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Privileged Architecture Version 1.10* for a detailed description of all available exception codes.

Trap Example

A particular CPU design contains three privilege modes: Machine, Supervisor, and User. Each privilege mode has its own user registers, control and status registers (CSRs) for trap handling, and stack area dedicated to them. While operating in User mode, a context switch is required to handle an event in Supervisor mode. The software sets up the system for a context switch, and then an `ECALL` instruction is executed which synchronously switches control to the environment-call-from-User mode exception handler.

Interrupt Example

A timer interrupt is required to trigger an event in the future, so a CPU writes its own `mtimecmp` register with a value of `mtime` + *ticks*, where *ticks* is some number of clock cycles in the future. Since `mtime` increments continually, it is independent of any instructions being executed by the CPU. At some point later, `mtimecmp` matches `mtime`, and the CPU enters an interrupt handler for the timer event.

Since an *interrupt* may occur anytime, and they are typically not part of the instruction execution sequence, they are *asynchronous* by nature.

Since an *exception* occurs as a result of executing an instruction, they are *synchronous* by nature.

1.2 Local and Global Interrupt Concepts

1.2.1 Local Interrupt Controllers

There are two available options on SiFive designs that provide low latency interrupts to the CPU.

First, the **Core Local Interrupter (CLINT)** offers a compact design with a fixed priority scheme, with preemption support for interrupts from higher privilege levels only. The primary purpose of the CLINT is to serve as a simple CPU interrrupter for software and timer interrupts, since it does not control other local interrupts wired directly to the CPU.

A second option is the **Core Local Interrupt Controller (CLIC)**, which is a fully featured local interrupt controller with configurations that support programmable interrupt levels and priorities. The CLIC also supports nested interrupts (preemption) within a given privilege level, based on the interrupt level and priority configuration.

Both the CLINT and CLIC integrate registers `mtime` and `mtimecmp` to configure timer interrupts, and `msip` to trigger software interrupts. Additionally, both the CLINT and the CLIC run at the core clock frequency.

1.2.2 Global Interrupt Controller

The global interrupt controller is termed the **Platform Local Interrupt Controller (PLIC)**. The PLIC provides system level flexibility for dispatching interrupts to a single CPU or multiple CPUs in the system. Global interrupts that route through the PLIC arrive at the CPU through a single interrupt connection with a dedicated interrupt ID. Each global interrupt has a programmable priority register available in the PLIC memory map. There is also a system level programmable threshold register which can be used to mask all interrupts below a certain level. The PLIC runs off a different clock than local interrupt controllers, which is typically an integer divided ratio from the core clock.

1.2.3 Built-in, Predefined Exceptions

RISC-V Instruction Set Architecture describes many different types of system exceptions, however none are defined to have a reserved location in the user defined interrupt vector table. One mode of operation of the local interrupt controller is called direct mode, where it does not use a vector table. In this mode, it's up to software to determine the source of the exception or interrupt, and act accordingly. There are also variations of vectored mode of operation that will be discussed in the specific sections for the CLINT and CLIC.

1.2.4 Interrupt Detection

All interrupts on SiFive designs implement level-high sensitive interrupt triggering. This is not configurable, but some custom implementations may decide to include device specific glue logic to convert interrupt sources from a rising or falling edge into a level high sensitive signal.

Chapter 2

Interrupt Configuration Registers

There are several Control and Status Registers (CSRs) within the CPU which are used for configuring interrupts. CSRs can only be read or written locally by executing variations of `csrr` and `csrw` instructions, and are not visible to other CPUs.

2.1 Interrupt Control and Status Registers (CSRs)

There are interrupt related CSRs contained in the CPU, as well as memory mapped configuration registers in the respective interrupt controllers. Both are used to configure and properly route interrupts to a CPU. Here we will discuss the Machine mode interrupt CSRs for the CPU only. Many Machine mode interrupt CSRs may have Supervisor or User mode equivalents. Refer to *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Privileged Architecture Version 1.10* for the full list.

- `mstatus` — Status register containing interrupt enables for all privilege modes, previous privilege mode, and other privilege level settings.
- `mcause` — Status register which indicates whether an exception or interrupt occurred, along with a code to distinguish details of each type.
- `mie` — Interrupt enable register for local interrupts when using CLINT modes of operation. In CLIC modes, this is hardwired to 0 and interrupt enables are handled using `clicintie[i]` memory mapped registers.
- `mip` — Interrupt pending register for local interrupts when using CLINT modes of operation. In CLIC modes, this is hardwired to 0 and pending interrupts are handled using `clicintip[i]` memory mapped registers.
- `mtvec` — Machine Trap Vector register which holds the base address of the interrupt vector table, as well as the interrupt mode configuration (direct or vectored) for CLINT and CLIC controllers. All synchronous exceptions also use `mtvec` as the base address for exception handling in all CLINT and CLIC modes.

- `mtvt` — Used only in CLIC modes of operation. Contains the base address of the interrupt vector table for selectively vectored interrupt in CLIC direct mode, and for all vectored interrupts in CLIC vectored mode. This register does not exist on designs with a CLINT.

2.1.1 Common Registers to CLIC and CLINT

The CLIC introduces new modes of operation that the CLINT does not support. However, both controllers support software and timer interrupts using the same configuration registers.

- `msip` — Machine mode software interrupt pending register, used to assert a software interrupt for a CPU.
- `mtime` — Machine mode timer register which runs at a constant frequency. Part of the CLINT and CLIC designs. There is a single `mtime` register on designs that contain one or more CPUs.
- `mtimecmp` — Memory mapped machine mode timer compare register, used to trigger an interrupt when `mtimecmp` is greater than or equal to `mtime`. There is an `mtimecmp` dedicated to each CPU.

> **Note**
>
> Timer interrupts always trap to Machine mode, unless delegated to Supervisor mode using the `mideleg` register. Similarly, Machine mode exceptions may be delegated to Supervisor mode using the `medeleg` register. For designs that also implement User mode, there exists `sideleg` and `sedeleg` registers to delegate Supervisor interrupts to User mode. Currently none of the SiFive designs support User mode interrupts.

2.1.2 Memory Mapped Interrupt Registers

There are memory mapped interrupt enable, pending, and optionally priority configuration registers based on which interrupt controller is being used. These are referenced in the following sections specific to the CLIC or PLIC. Note that there are no interrupt enable or priority configuration bits in the CLINT. Specific details for custom designs are included in the respective manual, which is part of the design tarball.

2.1.3 Early Boot: Setup `mtvec` Register

The `mtvec` register is required to be setup early in the boot flow, primarily for exception handling. Interrupts are not fully configured early in the boot flow, but exception handling is important to setup as early as possible, in the event an unexpected synchronous event needs to be handled. SiFive provides a portable software API, which also contains early boot code to support `mtvec` configuration, in a SiFive github repository called freedom-metal.

The example assembly code below shows the setup for the `early_trap_vector` which is part of the freedom-metal repository available on github.

```
/* Set up a simple trap vector to catch anything that goes wrong early in
 * the boot process. */
la t0, early_trap_vector
csrw mtvec, t0
```

The startup code also contains the functionality for `early_trap_vector`, shown below.

```
/* For sanity's sake we set up an early trap vector that just does nothing.  If
 * you end up here then there's a bug in the early boot code somewhere. */
    .section .text.metal.init.trapvec
    .align 2
early_trap_vector:
    .cfi_startproc
    csrr t0, mcause
    csrr t1, mepc
    csrr t2, mtval
    j early_trap_vector
    .cfi_endproc
```

A more sophisticated trap handler may be required later, after the initial bootup is complete. For example, using a C function to handle the trap, which might contain additional functionality based on the type of trap encountered. In this case, the `mtvec` register can be written directly using C code.

```
int mtvec_value = &my_function_handler;
__asm__ volatile ("csrr %0, mtvec" : "=r"(mtvec_value));
```

> **Note**
>
> It is recommended to disable interrupts globally using `mstatus.mie` prior to changing `mtvec`.

2.1.4 Standard Entry & Exit Behavior for Interrupt Handlers

Whenever an interrupt occurs, hardware will automatically save and restore important registers. The following steps are complete as an interrupt handler is entered.

- Save `pc` to `mepc`
- Save Privilege level to `mstatus.mpp`
- Save `mie` to `mstatus.mpie`
- Set `pc` to interrupt handler address, based on mode of operation
- Disable interrupts by setting `mstatus.mie=0`

At this point control is handed over to software where the interrupt processing begins. At the end of the interrupt handler, the `mret` instruction will do the following.

- Restore `mepc` to `pc`

- Restore `mstatus.mpp` to Priv
- Restore `mstatus.mpie` to `mie`

Note

`priv` refers to the current privilege level which is not visible while operating at that level. Possible values are Machine=3, Supervisor=1, User=0

There may be additional instructions or functionality contained within the handler, based on the interrupt controller and mode of operation. For example, saving/restoring additional user registers, enabling preemption, and handling global interrupts routed through the PLIC which require a claim/complete step. Subsequent sections for the CLINT, CLIC, and PLIC will describe these options in more detail.

Chapter 3

SiFive Interrupt Controllers

3.1 Core Local Interrupter (CLINT) Overview

The CLINT has a fixed priority scheme which implements Software, Timer, and External interrupts. Software preemption is only available between privilege levels using the CLINT. For example, while in Supervisor mode, a Machine mode interrupt will immediately take priority and preempt Supervisor mode operation. Preemption within a privilege level is not supported with the CLINT. The interrupt ID represents the fixed priority value of each interrupt, and is not configurable. There are two different CLINT modes of operation, direct mode and vectored mode. To configure CLINT modes, write `mtvec.mode` field, which is bit[0] of `mtvec` CSR. For direct mode, `mtvec.mode=0`, and for vectored mode `mtvec.mode=1`. Direct mode is the default reset value, while `mtvec.base` holds the base address for interrupts and exceptions in both modes.

The alignment requirements for `mtvec.base` are as follows:

- 4 bytes for CLINT Direct mode (`mtvec.mode=0`)
- 4*XLEN bytes for CLINT Vectored mode (`mtvec.mode=1`)
- 64 bytes for CLIC modes (`mtvec.mode=2` or `mtvec.mode=3`)

> **Note**
>
> CLIC modes are not supported with the CLINT

3.1.1 Example Interrupt Handler

The example below shows an assembly interrupt handler which pushes all registers onto the stack, calls the handler function, then pops all the registers off the stack.

```
        .align 2
        .global handler_table_entry
handler_table_entry:
        addi sp, sp, -32*REGBYTES
```

```
        STORE x1, 1*REGBYTES(sp)
        STORE x2, 2*REGBYTES(sp)
        ...
        STORE x30, 30*REGBYTES(sp)
        STORE x31, 31*REGBYTES(sp)
        //---- call C Code Handler ----
        call software_handler
        //---- end of C Code Handler ----
        LOAD x1, 1*REGBYTES(sp)
        LOAD x2, 2*REGBYTES(sp)
        ...
        LOAD x30, 30*REGBYTES(sp)
        LOAD x31, 31*REGBYTES(sp)
        addi sp, sp, 32*REGBYTES
        mret
```

For direct mode, all interrupts and exceptions would use `handler_table_entry` as the starting point of execution, which can be configured by writing the `mtvec` register with the address of the function.

The overhead of pushing and popping all registers is not usually required, and more efficient methods will be detailed in subsequent sections that introduce GCC compiler attributes specific to interrupt handler functions.

3.1.2 CLINT Direct Mode

Direct mode means all interrupts and exceptions trap to the same handler, and there is no vector table implemented. It is software's responsibility to execute code to figure out which interrupt occurred. The software handler in direct mode should first read `mcause.interrupt` to determine if an interrupt or exception occurred, then decide what to do based on `mcause.code` value which contains the respective interrupt or exception code.

3.1.3 Example Handler for CLINT Direct Mode

```
#define MCAUSE_INT_MASK  0x80000000        // [31]=1 interrupt, else exception
#define MCAUSE_CODE_MASK 0x7FFFFFFF        // low bits show code

void software_handler()
{
        unsigned long mcause_value = read_csr(mcause);

        if (mcause_value & MCAUSE_INT_MASK) {

                        // Branch to interrupt handler here
                        // Index into 32-bit array containing addresses of functions
                        async_handler[(mcause_value & MCAUSE_CODE_MASK)]();
        } else {

                        // Branch to exception handler
                        sync_handler[(mcause_value & MCAUSE_CODE_MASK)]();
        }
}
```

Software would first need to create and populate the `async_handler` and `sync_handler` tables used above, using the interrupt and exception functions designed to support their specific event, as described in the following table.

Interrupt	**Code**	**Description**
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	Reserved
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	Reserved
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	Reserved
1	11	Machine external interrupt
1	>=12 && <16	Reserved
1	>=16	Implementation defined local interrupts
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved
0	15	Store/AMO page fault
0	>= 16	Reserved

Table 1: Machine Cause (`mcause`) Register

The interrupt and exception categorizations listed in this table are standard to all CPU designs implementing RISC-V instruction set architecture.

In summary, CLINT direct mode requires software to setup the following:

1. The primary entry point for interrupt and exceptions, shown in `handler_table_entry`, the base address of which is assigned to `mtvec.base`
2. A software handler to determine whether the event is an interrupt or exception, which also contains code to jump to the appropriate interrupt function or exception handler
3. The actual interrupt or exception function, where the address of each function is written to the `asynch_handler` or the `sync_handler` arrays, respectively

3.1.4 CLINT Vectored Mode

Vectored mode introduces a method to create a vector table that hardware uses for lower interrupt handling latency. When an interrupt occurs in vectored mode, the `pc` will get assigned by the hardware to the address of the vector table index corresponding to the interrupt ID. From the vector table index, a subsequent jump will occur from there to service the interrupt. Recall that the vector table contains an opcode that is a jump instruction to a specific location.

```
.weak default_exception_handler
.balign 4, 0
.global default_exception_handler

.weak software_handler
.balign 4, 0
.global software_handler

.weak timer_handler
.balign 4, 0
.global timer_handler

.weak external_handler
.balign 4, 0
.global external_handler

.option norvc
.weak __mtvec_clint_vector_table
#if __riscv_xlen == 32
.balign 128, 0
#else
.balign 256, 0
#endif
.global __mtvec_clint_vector_table
__mtvec_clint_vector_table:

IRQ_0:
        j default_exception_handler
IRQ_1:
        j default_vector_handler
IRQ_2:
        j default_vector_handler
IRQ_3:
        j software_handler
IRQ_4:
        j default_vector_handler
IRQ_5:
        j default_vector_handler
IRQ_6:
        j default_vector_handler
IRQ_7:
        j timer_handler
IRQ_8:
        j default_vector_handler
IRQ_9:
        j default_vector_handler
IRQ_10:
        j default_vector_handler
IRQ_11:
        j external_handler
IRQ_12:
        j default_vector_handler
IRQ_13:
        j default_vector_handler
IRQ_14:
        j default_vector_handler
IRQ_15:
        j default_vector_handler
```

Figure 1: CLINT Vector Table Example Using Jump Instructions

The interrupt handler offset is calculated by `mtvec.base` + (`mcause.code` * 4). For example, software interrupts with ID of 3 would trap to offset `mtvec.base` + 0xC. The first entry in the table supports **exceptions** since they trap to the base address defined in `mtvec`. There is no vector table for exception handling.

```
#define MCAUSE_INT_MASK  0x80000000        // [31]=1 interrupt, else exception
#define MCAUSE_CODE_MASK 0x7FFFFFFF        // low bits show code

void default_exception_handler()
{
        // Vectored interrupts will jump directly to their vector table offset,
        // and will not enter software_handler() here.

        // read mcause for exception handling
        unsigned long mcause_value = read_csr(mcause);

        // Branch to synchronous handler,
        // or add code directly here
        sync_handler[(mcause_value & MCAUSE_CODE_MASK)]();
}
```

CLINT vectored mode does not require the same software overhead shown previously in the `software_handler` function for interrupt handling. In this mode when an interrupt occurs, program execution jumps directly to the vector table offset for the corresponding interrupt.

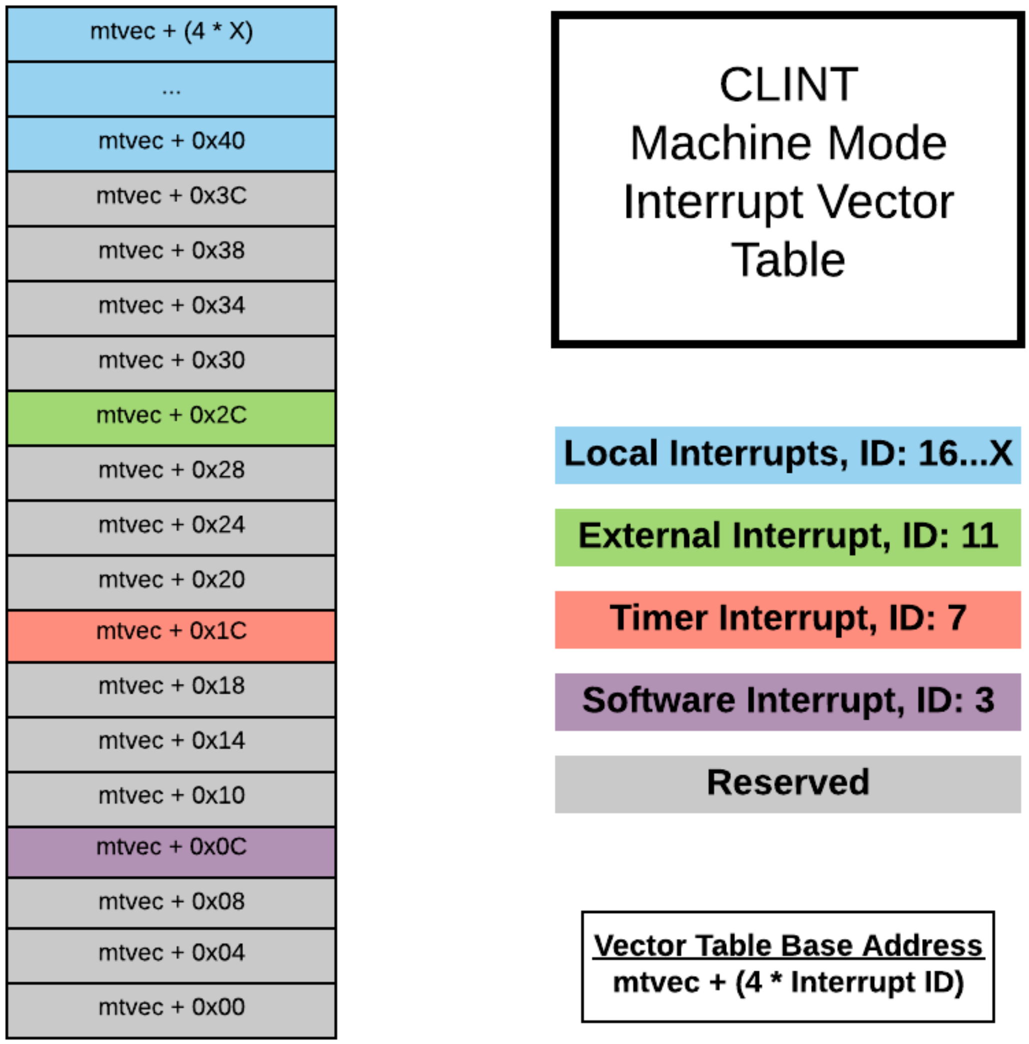

Figure 2: CLINT Vector Table Offsets for Interrupts

3.1.5 CLINT Interrupt Levels, Priorities, and Preemption

For CPU designs that utilize Machine mode only, the CLINT would have the following configuration:

- Software interrupts — Interrupt ID #3.
    - Software interrupts are triggered by writing the memory mapped interrupt pending register `msip` for a particular CPU. In a multi-CPU system, other CPUs are able to write `msip` to trigger a software interrupt on any other CPU in the system. This allows for efficient inter-processor communication.
- Timer interrupt — Interrupt ID #7.

    - Timer interrupts are triggered when the memory mapped register `mtime` is greater than or equal to the global timebase register `mtimecmp`, and both registers are part of the CLINT and CLIC memory map. In a multi-CPU system, `mtimecmp` can be written by other CPUs to setup timer interrupts.
- External interrupts — Interrupt ID #11.
    - Global interrupts are usually first routed to the PLIC, then into the CPU using External interrupt ID #11. For systems that do not implement a PLIC, this interrupt can optionally be disabled by tying it to logic 0.
- Local Interrupts are Interrupt ID #16 and higher.
    - Local Interrupts may connect directly to an interrupt source, and do not need to be routed through the PLIC. Specifically to the CLINT, they all have fixed interrupt priority based on their interrupt ID. The maximum number of local interrupts for 32-bit designs is 32, however the first 16 have predefined usage. Likewise, the maximum number of local interrupt for 64-bit designs is 64, where the additional 48 are available for custom usage.

3.1.6 System Level Block Diagram using CLINT Only

An example configuration using CLINT with no global interrupt controller is shown below.

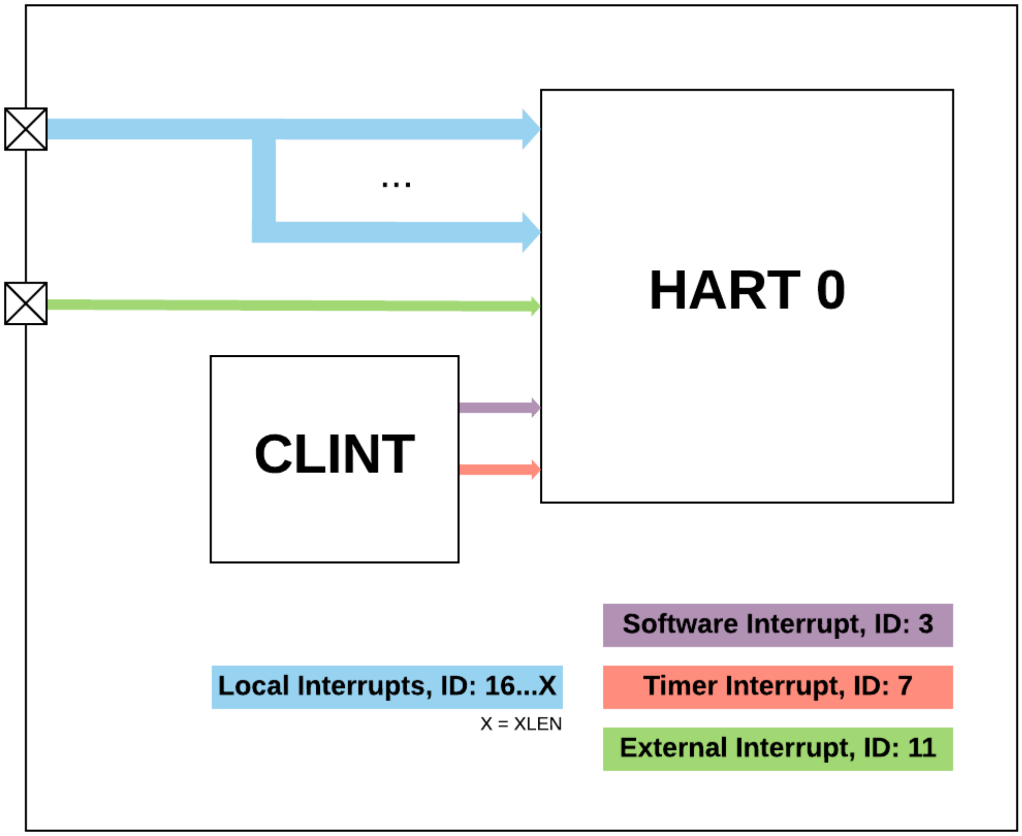

Figure 3: CLINT Block Diagram for Machine Mode

3.2 Core Local Interrupt Controller (CLIC) Overview

The CLIC has a more flexible configuration than the CLINT, however the CLINT is a smaller design overall. Some CLIC features include:

- Reverse compatibility with the CLINT for software, timer, and external interrupts, when programmed to use legacy CLINT modes through the `mtvec` register.
- Introduces new CLIC direct and CLIC vectored modes that offer programmable interrupt levels and priorities, which support preemption from interrupts of higher levels and priorities.
- Retains the `mtime` and `mtimecmp` memory mapped timer registers, and `msip` register for triggering software interrupts.
- Extends the `mode` field of `mtvec` by six bits, [5:0], where [1:0] are defined to support two additional modes: **CLIC direct** and **CLIC vectored**.
  - To configure CLIC direct mode, write `mtvec.mode=0x02`, and for CLIC vectored mode, `mtvec.mode=0x03`.

- Flexibility to implement CLIC without vectored mode for applications that do not require low latency real time interrupt handling.

3.2.1 CLIC Direct Mode

CLIC direct mode operates in a similar fashion to CLINT direct mode, however it introduces a feature called *selective vectoring*. Selective vectoring allows each interrupt to be configured for CLIC hardware vectored operation, while all other interrupts use CLIC direct mode. The `clicintcfg[i].SHV` field is used to configure selective vectoring. CLIC direct modes uses `mtvec` as the base address for exception and interrupt handling, but introduces `mtvt` as the base address for interrupts configured for selective hardware vectoring.

3.2.2 CLIC Vectored Mode

CLIC vectored mode has a similar concept to CLINT vectored mode, where an interrupt vector table is used for specific interrupts. However, in CLIC vectored mode, the handler table contains the address of the interrupt handler instead of an opcode containing a jump instruction. When an interrupt occurs in CLIC vectored mode, the address of the handler entry from the vector table is loaded and then jumped to in hardware. CLIC vectored mode uses `mtvec` exclusively for exception handling, since `mtvt` is used to define the base address for all vectored interrupts.

It should be noted that access to `mtvt` may need to be done directly using the CSR number (0x307) instead of the `mtvt` keyword if it is not supported in the toolchain. For example:

```
int mtvt_read_value;
__asm__ volatile ("csrr %0, 0x307" : "=r"(mtvt_read_value));
```

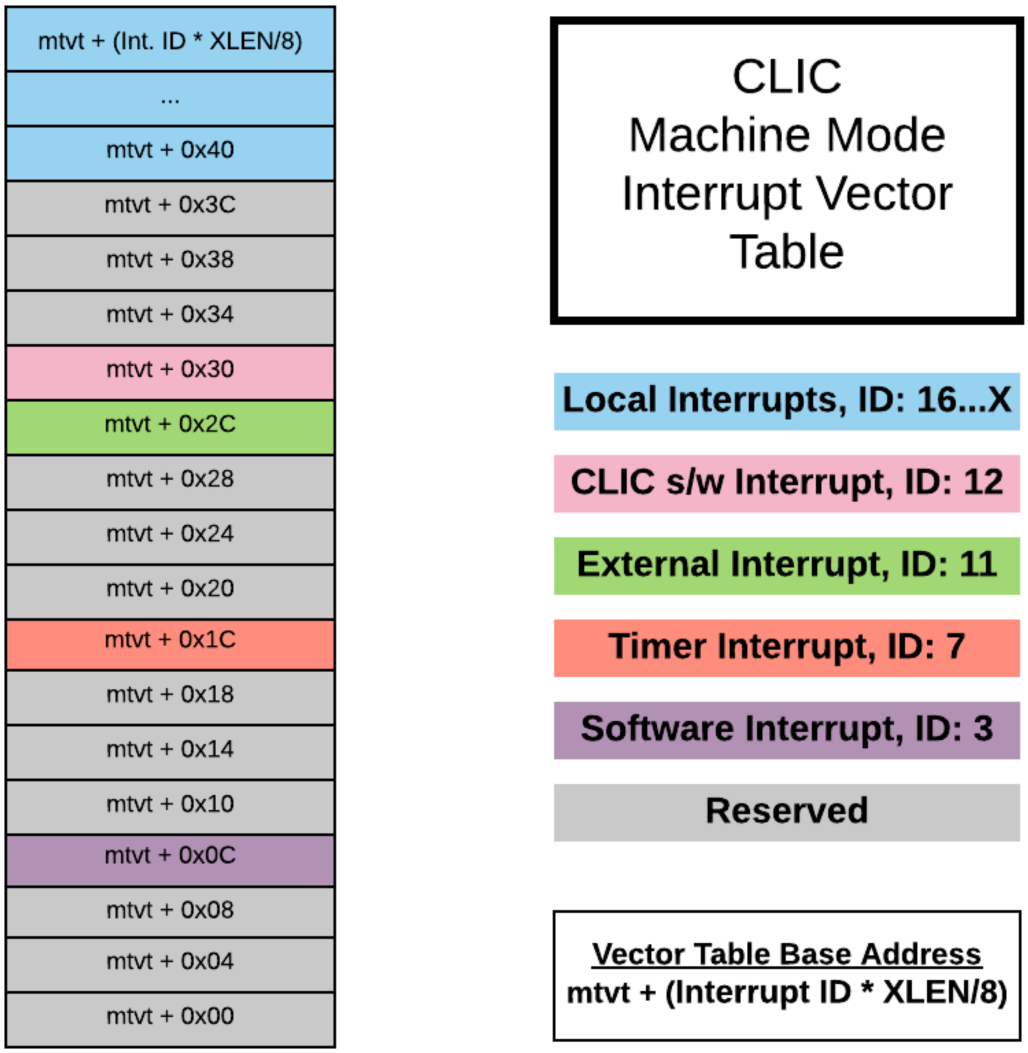

Figure 4: CLIC Vector Table Offsets for Interrupts

3.2.3 CLIC Interrupt Levels, Priorities, and Preemption

The CLIC allows programmable interrupt levels and priorities for all supported interrupts. The interrupt level is the first step to determine which interrupt gets serviced first, whereas the priority is used to break the tie in the event two interrupts of the same level are received by the CPU at the same time. The CLIC can support up to 1024 interrupts, where 0 through 16 are reserved for software, timer, external, and CLIC software interrupt for all privilege modes. The CLIC software interrupt (ID #12) serves a similar function as the legacy machine software interrupt, except its typical use interrupting software threads. This leaves a total of 1008 available external interrupts for custom use. Note that interrupt ID #12 is likely a future addition to the standard RISC-V Machine Cause (`mcause`) Register Table, referenced previously.

3.2.4 Software Attributes for Interrupts

To help with efficiency of save and restore context, interrupt attributes can be applied to functions used for interrupt handling.

```
void __attribute__ ((interrupt))
software_handler (void) {
        // handler code
}
```

This attribute will save and restore additional registers that are used within the handler, and add an `mret` instruction at the end of the handler.

The functionality can be demonstrated by comparing the list output of functions with and without the attribute applied.

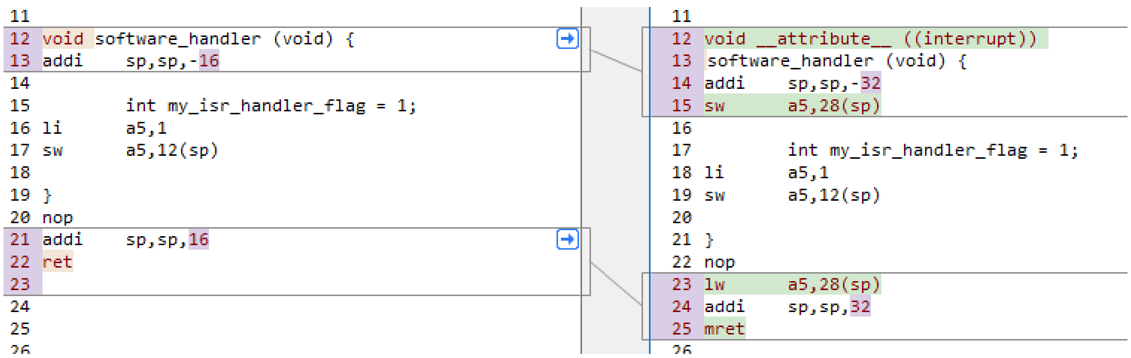

Figure 5: No Attribute Compared to Standard Interrupt Attribute

Enabling Preemption in CLIC Modes

In order for an interrupt of a higher level to preempt an active interrupt of a lower level, `mstatus.mie` needs to be enabled (non zero) within the handler, since it is disabled by hardware automatically upon entry. Prior to re-enabling interrupts through `mstatus.mie`, first `mepc` and `mcause` must be saved, and subsequently restored before `mret` is executed at the end of the handler. There is a CLIC specific interrupt attribute that will do these steps automatically.

```
void __attribute__ ((interrupt("SiFive-CLIC-preemptible")))
software_handler (void) {
        // handler code
}
```

> **Note**
>
> Using the **SiFive-CLIC-preemptible** attribute requires the addition of the -fomit-frame-pointer compiler flag.

The functionality of this CLIC specific attribute can be demonstrated by comparing the list output of functions with and without the attribute applied.

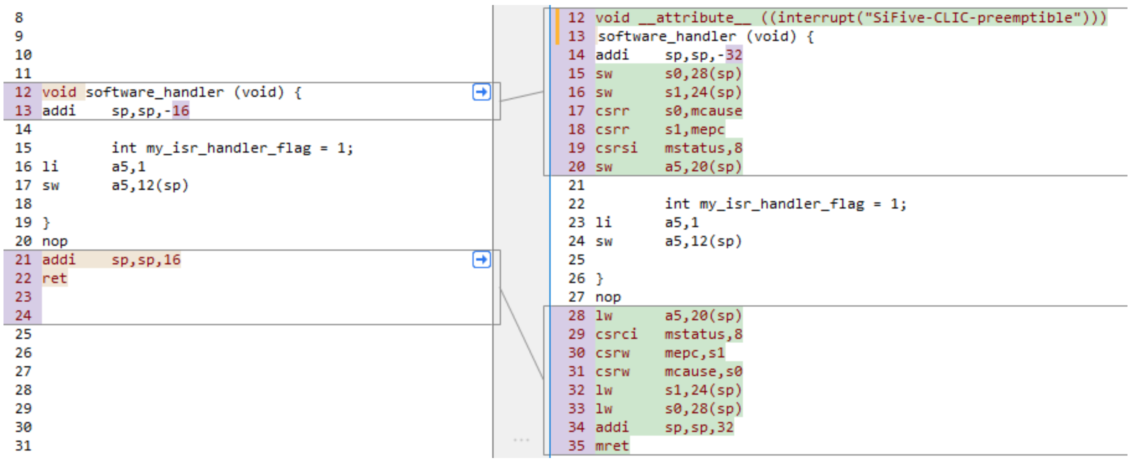

Figure 6: No Attribute Compared to CLIC Interrupt Attribute

This attribute applies to vectored interrupts. To support preemption for non-vectored interrupts, refer to the CLIC spec example here. Also, refer to the CLIC section on how to manage interrupt stacks across privilege modes here.

3.2.5 Details for CLIC Modes of Operation

In CLIC modes of operation, both `mie` (machine interrupt enable) and `mip` (machine interrupt pending) registers are hard wired to zero, and their functionality moves to `clicintie[i]` and `clicintip[i]` registers. Additionally, `mideleg` (interrupt delegation register), which can direct interrupts to be handled in different privilege levels, is not available. Instead, the `mode` field in the `clicintcfg[i]` register is used to determine which privilege mode the interrupt is taken.

> **Note**
>
> The `mideleg` register is not implemented on designs which implement only Machine mode.

New Registers within the CLIC

CLIC modes of operation introduce new registers compared to designs that implement a CLINT.

- `cliccfg` - Memory mapped CLIC configuration register
  - Determines the number of levels and priorities set by `clicintcfg[i]`. Also contains selective hardware vector configuration, which allows direct mode or vectored mode on a per-interrupt basis
- `clicintcfg[i]` - Memory mapped CLIC interrupt configuration register

  - Sets the pre-emption level and priority of a given interrupt
- `clicintie[i]` - Memory mapped CLIC interrupt enable register
- `clicintip[i]` - Memory mapped CLIC interrupt pending register
- `mtvt` - CSR which holds the Machine Trap Vector Table base address for CLIC vectored interrupts
  - Write Always, Read Legal (WARL) register allows for relocatable vector tables, where `mtvt.base` requires a minimum 64-byte alignment, but can increase depending on the total number of CLIC interrupts implemented. Refer to the CLIC specification for more details.
- `mnxti` - CSR containing the Machine Next Interrupt Handler Address and Interrupt-Enable
  - Used by software in CLIC direct mode to service the next interrupt of equal or greater level before returning to a lower level context
  - A read to this CSR returns the address of an entry in the vector table (mtvt)
  - Can simultaneously be written to, which affects mstatus - this is used to disable interrupts on the same cycle as the read
  - Returns 0 if no higher level interrupt than the saved context, or if a HART is not in CLIC mode, or if the next highest ranked interrupt is selectively hardware vectored
- `mintstatus` - Read only CSR which holds the Active Machine Mode Interrupt Level
  - Read only register containing current interrupt level for the respective privilege mode

3.2.6 Changes to CSRs in CLIC Mode

- `mstatus`
  - The `mstatus.mpp` and `mstatus.mpie` are accessible via fields in the `mcause` register
- `mie` and `mip`
  - `mie` and `mip` are hardwired to zero and replaced with memory mapped `clicintie[i]` and `clicintip[i]` registers
- `mtvec`
  - Additional modes which enable CLIC mode of operation
- `mcause`
  - Stores previous privilege mode, and previous interrupt enable

3.2.7 System Level Block Diagram using CLIC only

An example configuration using a CLIC with no global interrupt controller is shown below.

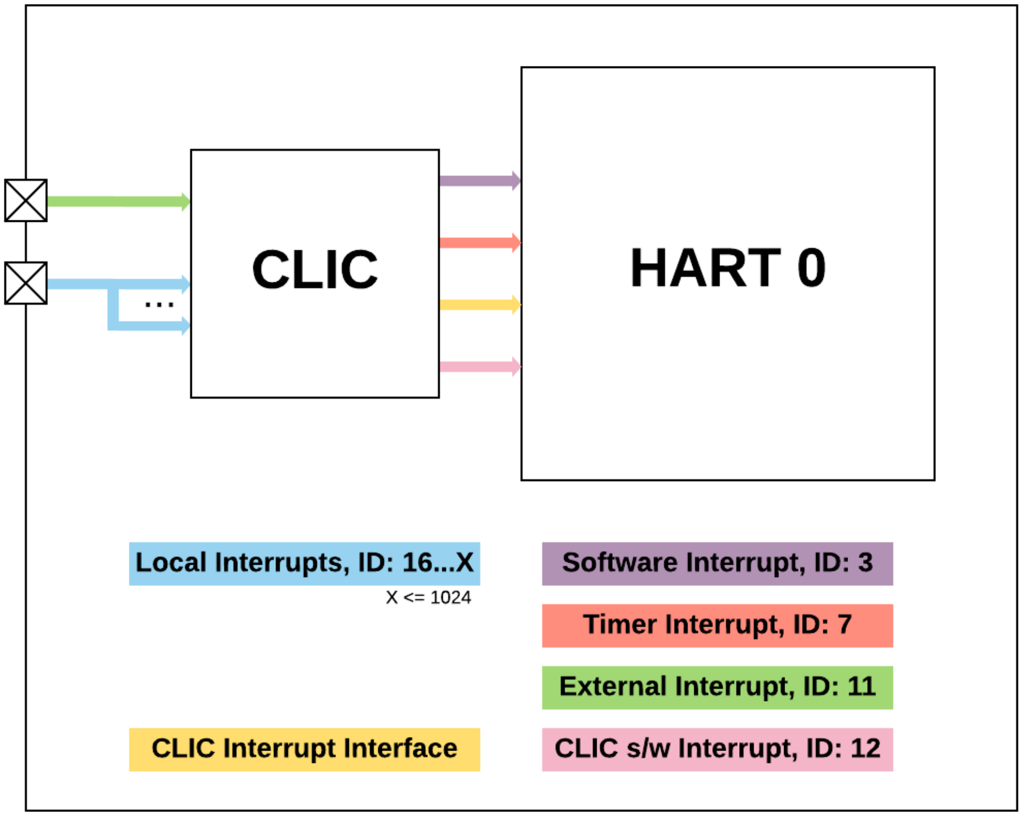

Figure 7: CLIC Block Diagram for Machine Mode

> **Note**
>
> In CLIC modes of operation all local interrupts have programmable levels and priorities, which is not determined by the Interrupt ID.

The external interrupt connection may not be needed for designs that do not require global interrupts routed through the PLIC, and may be disabled by tying it to logic zero. In legacy CLINT mode of operation, the software, timer, and external interrupt lines are wired directly to the CPU. These lines are not used when CLIC modes are selected.

3.3 Platform Level Interrupt Controller (PLIC) Overview

The PLIC is used to manage all global interrupts and route them to one or many CPUs in the system. It is possible for the PLIC to route a single interrupt source to multiple CPUs. Upon entry to the PLIC handler, a CPU reads the claim register to acquire the interrupt ID. A successful claim will atomically clear the pending bit in the PLIC interrupt pending register, signaling to the system that the interrupt is being serviced. During the PLIC interrupt handling process, the pending flag at the interrupt source should also be cleared, if necessary. It is legal for a CPU to attempt to claim an interrupt even when the pending bit is not set. This may happen, for example, when a global interrupt is routed to multiple CPUs, and one CPU has already claimed the interrupt just prior to the claim attempt on another CPU. Before exiting the PLIC handler with MRET instruction (or SRET/URET), the claim/complete register is written back with the non zero claim/complete value obtained upon handler entry. Interrupts routed through the PLIC incur an additional three cycle penalty compared to local interrupts. Cycles in this case are measured at the PLIC frequency, which is typically an integer divided value from the CPU and local interrupt controller frequency.

3.3.1 PLIC IDs, Priorities and Preemption

PLIC ID

There are up to 1024 available interrupts routed into the PLIC, which are numbered sequentially 1 through 1024. PLIC interrupt ID 1 is tied to `global_interrupt[0]` which is described by *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10* to mean "no interrupt". Thus, the first usable PLIC interrupt has ID value of 2.

PLIC Priorities

Each interrupt into the PLIC has a configurable priority, from 1-7, with 7 being the highest priority. A value of 0 means do not interrupt, effectively disabling that interrupt. If global interrupts with the same priority arrive at the same time into the PLIC, priority is given to the *lower* of the two Interrupt IDs.

PLIC Thresholds

There is a per-CPU threshold register within the PLIC that allows interrupts configured below a certain level to be blocked. For example, if the threshold register contains a value of 5, all PLIC interrupt configured with priorities from 1 through 5 will not be allowed to propagate to the CPU.

PLIC Connectivity

Global interrupts routed through the PLIC are connected to the CPU in slighty different ways depending on the local interrupt selection. If the PLIC is used with the CLINT, then the external interrupt connection routed from the PLIC is tied directly to the CPU. If the PLIC is used with the CLIC, then the external interrupt connection is not used, and the interrupt is routed from the PLIC through the CLIC interface.

PLIC Detailed Functionality

By definition, the PLIC cannot forward a new interrupt to a HART that has claimed an interrupt but has not yet finished the complete step of the interrupt handler. Thus, the PLIC does not support preemption of global interrupts to an individual HART. However, since PLIC interrupts arrive at the CPU through the external interrupt connection, preemption may occur from other CLIC local interrupts that are configured with a higher priority than the external interrupt. To support preemption, `mstatus.mie` needs to be re-enabled within the handler, since it is disabled by hardware upon entry. Interrupt IDs for global interrupts routed through the PLIC are independent of the interrupt IDs for local interrupts. Thus, software may need to implement a specific handler which supports a software lookup table for the global interrupts that are managed by the PLIC, and arrive at the CPU through the external interrupt connection.

> **Note**
>
> Recall that the CLINT local interrupts priorities are fixed since they are tied to their interrupt ID, while CLIC has programmable levels and priorities.

Additionally, the PLIC handler may check for additional pending global interrupts once the initial claim/complete process has finished, prior to exiting the handler. This method could save additional PLIC save/restore context for global interrupts.

3.3.2 PLIC Handler Example

Since all global interrupts routed through the PLIC are connected to the CPU through the external interrupt, this handler requires the additional claim/complete step to notify the PLIC that the current global interrupt is being serviced.

```
void external_handler()
{
        //get the highest priority pending PLIC interrupt
        uint32_t int_num = plic.claim_comlete;

        //branch to handler
        plic_handler[int_num]();

        //complete interrupt by writing interrupt number back to PLIC
        plic.claim_complete = int_num;
}
```

The `external_handler` function would reside at vector table offset +0x2C when using CLINT vectored mode of operation. This offset changes to +0x58 for 64-bit architectures while using CLIC vectored mode of operation. For non-vectored modes, the `async_handler` resides at index 11 in the software table. The `async_handler` was referenced in the CLINT example previously.

The `plic_handler` software table would be populated with functions that support each of the PLIC specific global interrupts.

3.3.3 PLIC + CLINT, Machine Mode Interrupts Only

For a multi-CPU system implementing Machine mode only, an example configuration is shown below.

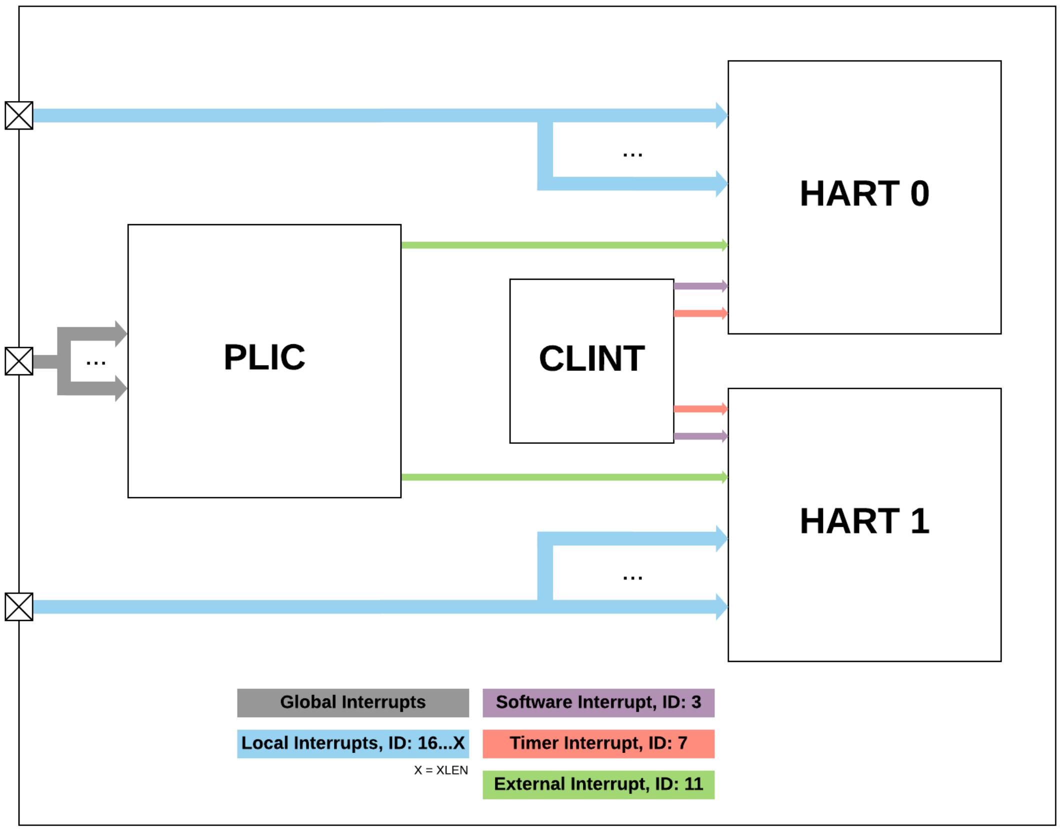

Figure 8: PLIC + CLINT PLIC Block Diagram for Machine Mode

> **Note**
>
> For systems that implement Supervisor mode, there will be additional Supervisor interrupt connections into each HART. User level interrupts are only available on devices which implement the RISC-V "N" extension.

3.3.4 PLIC + CLIC, Machine Mode Interrupts Only

Below is a representation of a design implementing multiple cores which requires a PLIC global interrupt controller.

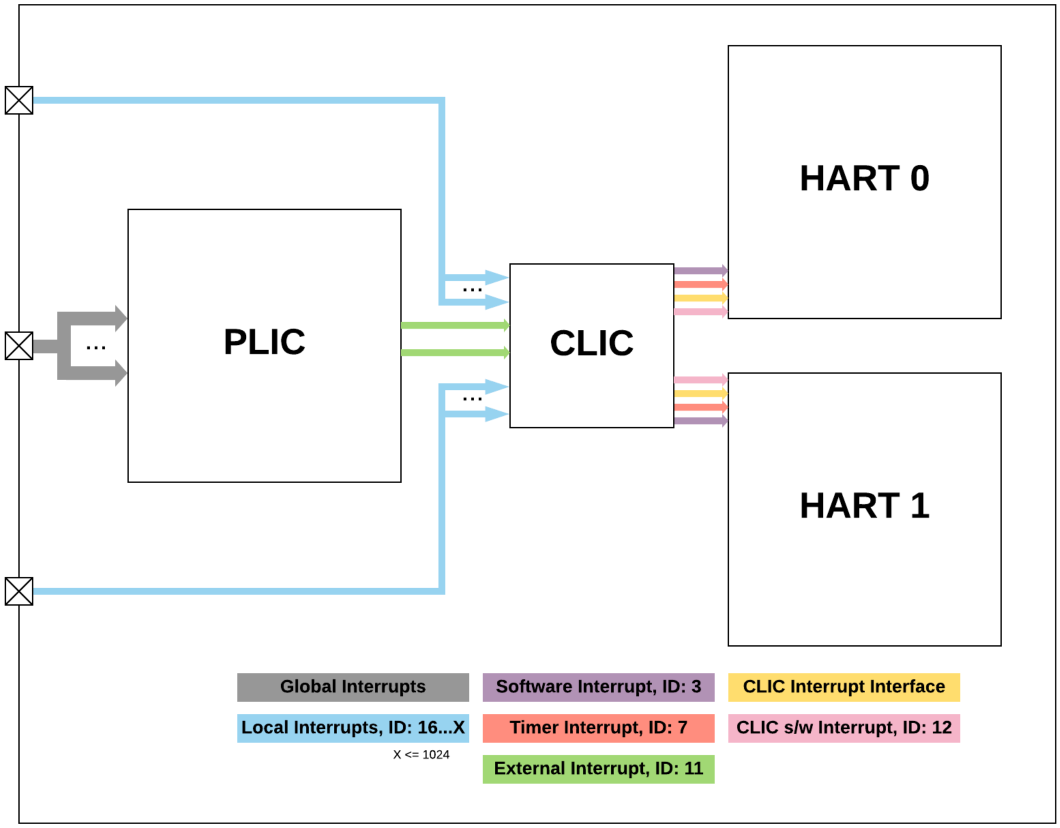

Figure 9: PLIC + CLIC Block Diagram for Machine Mode

The external interrupt connection is routed directly to the HART when CLIC operates in legacy CLINT mode, but this connection is not used in CLIC modes of operation.

Local interrupts are routed into the CLIC where the memory mapped registers exist for configurable interrupt levels and priorities. This is different from the connectivity when using the CLINT, where they are connected directly to the HART.

Chapter 4

Additional Code Examples

The steps below show the high level configuration to properly enable an interrupt.

4.1 Pseudo Code to Setup an Interrupt

1. Write `mtvec` to configure the interrupt mode and the base address for the interrupt vector table, and optionally, `mtvt` for CLIC modes. The CSR number for `mtvt` is 0x307.
2. Enable interrupts in memory mapped PLIC or CLIC register space. The CLINT does not contain interrupt enable bits.
3. Write `mie` to enable the software, timer, and external interrupt enables for each privilege mode
4. Write `mstatus` to enable interrupts globally for each supported privilege mode

> **Note**
>
> `mie` register is disabled in CLIC modes. Use `clicintie[i]` to enable interrupts in CLIC modes of operation.

4.2 Freedom Metal API

The freedom-metal library is a bare metal C programming environment provided by SiFive that utilizes an application program interface (API) designed to be portable across different CPU architectures. The freedom-e-sdk repo includes freedom-metal to leverage its functionality for software examples and startup code which run on any SiFive design. For a full list of examples, see https://github.com/sifive/freedom-e-sdk/tree/master/software. To reference the freedom-metal documentation, see https://sifive.github.io/freedom-metal-docs/.

4.3 DeviceTree Interrupt Mapping in design.dts file

SiFive's design environment generates a device tree specification file called design.dts for all supported cores. This file describes the hardware, including the base address for interfaces and peripherals, size of the available interface, and the interrupt connectivity. It is the starting point for auto-generating software components that are necessary for compiling software for new designs. The following auto generated components exist in the `freedom-e-sdk/bsp` path:

- Header files (.h) used by freedom-metal library for startup code and initialization
- Linker scripts (.lds) used map the software builds to the ports or memory available in the design
- Makefile named settings.mk which passes architecture and application binary interface information to the toolchain, and additional details to support output file formatting

To understand how the software components are generated from the design.dts file, refer to the Custom Core Getting Started Guide, available in the Application Notes section at https://www.sifive.com/documentation.

4.4 Interrupt components in design.dts - CLIC Example

The interrupt components of an E21 design.dts file are shown below. This design uses a CLIC local interrupt controller, and here we highlight the interrupt specific components described in the design.dts file.

```
L14: cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        L4: cpu@0 {
                clock-frequency = <0>;
                compatible = "sifive,caboose0", "riscv";
                device_type = "cpu";
                hardware-exec-breakpoint-count = <4>;
                reg = <0x0>;
                riscv,isa = "rv32imac";
                riscv,pmpregions = <4>;
                status = "okay";
                timebase-frequency = <1000000>;
                L3: interrupt-controller {
                        #interrupt-cells = <1>;
                        compatible = "riscv,cpu-intc";
                        interrupt-controller;
                };
        };
};
```

The L3 instance above is the **interrupt-controller**, which is part of the CPU, and is the parent node to **interrupt-controller@20000000** below.

```
L1: interrupt-controller@2000000 {
        #interrupt-cells = <1>;
```

```
        compatible = "sifive,clic0";
        interrupt-controller;
        interrupts-extended = <&L3 3 &L3 7 &L3 11>;
        reg = <0x2000000 0x1000000>;
        reg-names = "control";
        sifive,numintbits = <4>;
        sifive,numints = <143>;
        sifive,numlevels = <16>;
};
```

The line containing **<&L3 3 &L3 7 &L3 11>** describe the software, timer, and external interrupt IDs of the local interrupt controller.

```
L9: local-external-interrupts-0 {
        compatible = "sifive,local-external-interrupts0";
        interrupt-parent = <&L1>;
        interrupts = <16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 ... >;
};
```

The local-external-interrupts-0 node **L9**, above, has **L1** as the parent node, and lists the additional local interrupt lines available in the design, starting at 16. The total number of interrupts in this list is defined by the user when designing a core on SiFive.com.

> **Note**
>
> The compatible string above refers directly to the file freedom-metal/src/drivers/sifive_local-external-interrupts0.c

4.5 Interrupt components in design.dts - CLINT Example

The interrupt components of an S76 design.dts file are shown below. This design uses a CLINT local interrupt controller, as well as a PLIC global interrupt controller. Here we highlight the interrupt specific components described in the file.

```
L7: cpu@0 {
        clock-frequency = <0>;
        compatible = "sifive,bullet0", "riscv";
        d-cache-block-size = <64>;
        d-cache-sets = <128>;
        d-cache-size = <32768>;
        device_type = "cpu";
        hardware-exec-breakpoint-count = <4>;
        i-cache-block-size = <64>;
        i-cache-sets = <128>;
        i-cache-size = <32768>;
        next-level-cache = <&L10>;
        reg = <0x0>;
        riscv,isa = "rv64imafdc";
        riscv,pmpregions = <8>;
        sifive,dls = <&L6>;
        sifive,itim = <&L5>;
        status = "okay";
```

```
        timebase-frequency = <1000000>;
        L4: interrupt-controller {
                #interrupt-cells = <1>;
                compatible = "riscv,cpu-intc";
                interrupt-controller;
        };
};
```

The L4 instance above is the **interrupt-controller**, which is part of the CPU, and is the parent node to **clint@2000000** below.

```
L2: clint@2000000 {
        compatible = "riscv,clint0";
        interrupts-extended = <&L4 3 &L4 7>;
        reg = <0x2000000 0x10000>;
        reg-names = "control";
};
```

The line containing **<&L4 3 &L4 7>** describe the software and timer interrupt IDs of the local interrupt controller. Notice the external interrupt ID #11 is not in this list like it was in the CLINT.

```
L9: global-external-interrupts {
        compatible = "sifive,global-external-interrupts0";
        interrupt-parent = <&L1>;
        interrupts = <1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ...>;
};
```

The PLIC is represented here by the **global-external-interrupts** node, but its parent node is **interrupt-controller@c000000**, below. The total number of interrupts is configured when designing a core on SiFive.com. The **interrupts** field represents the *global* interrupts that route into the PLIC controller, which are independent of the interrupt IDs for the local interrupt controller.

```
L1: interrupt-controller@c000000 {
        #interrupt-cells = <1>;
        compatible = "riscv,plic0";
        interrupt-controller;
        interrupts-extended = <&L4 11>;
        reg = <0xc000000 0x4000000>;
        reg-names = "control";
        riscv,max-priority = <7>;
        riscv,ndev = <127>;
};
```

The **interrupt-controller@c000000** node has **L4** as the parent node, and this is external interrupt ID #11 into the CPU. This is the connection into the CPU from the PLIC, which routes global interrupts to individual CPUs.

> **Note**
>
> The compatible string above refers directly to the file freedom-metal/src/drivers/riscv_plic0.c

Chapter 5

Privilege Levels

5.1 Priorities for Supervisor & Machine Mode Interrupts

The privilege levels allow protection to certain software components that only run, or have access from, higher privilege levels. At a minimum, any RISC-V implementation must implement Machine mode, as this is the highest privilege level.

Number of Levels	Supported Modes	Intended Usage
1	M	Simple Embedded Systems
2	M, U	Secure Embedded Systems
3	M, S, U	Systems running Unix-like operating systems

Table 2: Table RISC-V Privilege Modes

Only designs that include the RISC-V "N" extension allow interrupts in User mode. When Supervisor mode is implemented, additional interrupts are included, as shown below.

- *Software Interrupt, **Supervisor** Mode, Interrupt ID: 1*
- Software Interrupt, Machine Mode, Interrupt ID: 3
- *Timer Interrupt, **Supervisor** Mode, Interrupt ID: 5*
- Timer Interrupt, Machine Mode, Interrupt ID: 7
- *External Interrupt, **Supervisor** Mode, Interrupt ID: 9*
- External Interrupt, Machine Mode, Interrupt ID: 11

A CPU operating in Supervisor mode will trap to Machine mode upon the arrival of a Machine mode interrupt, unless the Machine mode interrupt has been delegated to Supervisor mode through the `mideleg` register. On the contrary, Supervisor interrupts will not immediately trigger if a CPU is in Machine mode. While operating in Supervisor mode, a CPU does not have visibility to configure Machine mode interrupts.

5.2 Context Switch Overhead

Each privilege level has its own interrupt configuration registers, or a subset of bits in the respective machine mode only registers. Additionally, there is save/restore overhead to support a context switch to a different privilege level. Context switches include the following:

- 31 User registers, x1 - x31 (x0 is hardwired to 0)
- Program Counter (`pc`)
- 32 Floating point registers
- Floating point Control and Status Register (`fcsr`)

Prior to initiating a context switch, these registers should be saved on the stack. Likewise, prior to returning from a different privilege level, they are required to be restored from the stack.

A Context switch is initiated through the `ECALL` instruction which results in a environment-call-from-x-mode exception, where x is either M, S, or U, depending on the current privilege level. Return from an `ECALL` instruction occurs by executing the respective xRET instruction, where again, x is M, S, or U. The xRET instruction can be used in privilege mode x or higher. For more information on context switches, refer to *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Privileged Architecture Version 1.10*