RISC-V Exceptions and Interrupts

Sean Keller

October 2020

1 Privilege Modes and Control Status Registers

RISC-V supports three different privilege modes: M-mode (highest priority), S-mode, and U-mode (lowest priority). Currently, hypervisor-mode (H-mode) is reserved and is not a formal part of the RISC-V ISA. RISC-V also supports an optional debug mode (D-mode) for off-chip debugging that can be considered an additional privilege mode with more access than M-mode. Code run in machine-mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. M-mode can be used to manage secure execution environments on RISC-V. User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and operating system usage respectively. Privilege-level actions like exception and interrupt handling rely on a set of control and status registers (CSRs). In this document, CSRs or fields in CSRs prefixed with an *x* refer to the possibility that CSRs can have three privilege-specific implementations: M-mode (x = m), S-mode (x = s), and U-mode (x = u). These privilege-specific CSRs are controlled by their respective privilege-specific trap handlers.

2 Exceptions and Interrupts

RISC-V defines exceptions as an unusual condition occurring at run time associated with an instruction in the current hart. Exceptions can be organized into six different categories: fetch, load, store, misaligned jump/branch (i.e misaligned instruction address), illegal instruction, and ECALL/EBREAK. Interrupts refer to an external, asynchronous event that may cause the RISC-V hart to experience an unexpected transfer of control. Interrupts are organized into three different categories: external, software, and timer. External interrupts are raised by devices connected to the processor, software interrupts are raised by programs, and timer interrupts are raised when the value in the *xtime* CSR is greater than or equal the value in the *xtimecmp* CSR. For multi-hart systems, interrupts also rely on an inter-processor interface (see Section 6) to handle interrupts between multiple harts.

3 Exception Handling

If an exception occurs, control is relinquished by the program that is currently being executed and transferred to the trap handler. The trap handler can be thought of as a software function call the program executes in response to an erroneous hardware event. During this function call the program jumps to an address and writes to a set of CSRs before resuming the program at the location that raised the exception. Before an exception can be raised and handled, the trap handler must be initialized. First, software must decide whether it will be handled by the M-mode, S-mode, or U-mode trap handler. By default, all traps are handled by the M-mode trap handler but exceptions raised in S-mode and U-mode can be delegated to the lower-privilege S-mode handler through the *medeleg* CSR. Similarly, the *sedeleg* CSR can delegate exceptions raised in U-mode to the U-mode trap handler. The *(m/s)edeleg* CSR delegates exceptions by raising the bits in positions that correspond to the exception code numbers (more on this later). This delegation process is summarized by the flow chart in Figure 1. Next, software writes to the *xtvec* CSR. The *xtvec* CSR contains a base address and a mode field. This mode field supports two options: direct and vectored. Regardless of which mode is selected, the trap handler will start at the base address.

After *medeleg* and *xtvec* are set, the program jumps to the address where the trap handler is located. The first task the trap handler does is to write the *xepc* CSR with the virtual address of

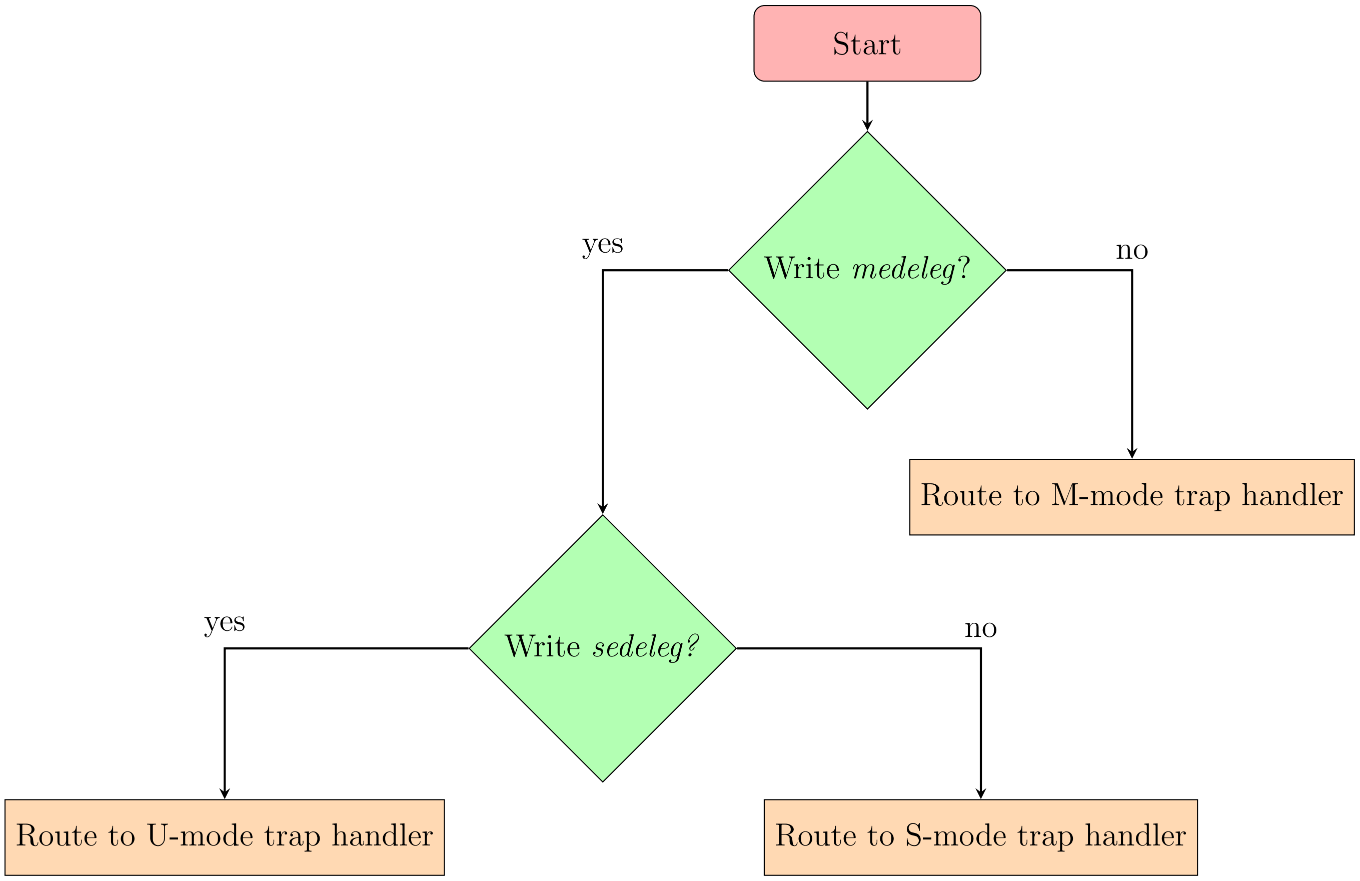
Start
Write *medeleg*?
yes
no
Route to M-mode trap handler
Write *sedeleg?*
yes
no
Route to U-mode trap handler
Route to S-mode trap handler

Figure 1: Flow Chart of Delegation Process

the instruction that took the trap. This address is stored so that the program can be returned to and resumed after the trap handler is finished. Next, the xPP field in *xstatus* is written with the two or one bit privilege level encoding that corresponds to the privilege level the program was in when the exception was raised. For the *mstatus* CSR, MPP is a two bit field that can hold the values 10 (M-mode), 1 (S-mode), or 0 (U-mode). MPP supports all three privilege modes because exceptions are delegated to the M-mode trap handler by default. For the *sstatus* and *ustatus* CSRs, SPP and UPP are one bit fields where SPP only supports 1 (S-mode) and 0 (U-mode) while UPP must be set to 0 (U-mode). After xPP is set, the *xtval* CSR is written with exception-specific information. For EBREAKs, misaligned addresses, access faults, and page faults, *mtval* will contain the faulting virtual address. When page-based virtual memory is enabled, *mtval* is written with the faulting virtual address, even for physical-memory access exceptions. For illegal instructions, *mtval* contains the faulting instruction bits of the illegal instruction i.e the faulting instruction. Once this is done, the hardware will write an exception code to the *xcause* CSR. If an instruction raises multiple synchronous exceptions, the exceptions are taken by the trap handler and reported in *xcause* according to a pre-defined priority structure. Finally, the trap handler executes an xRET instruction to jump back to the address stored in *xepc* so that the program can resume. This trap handler process is summarized by the flow chart in Figure 2.

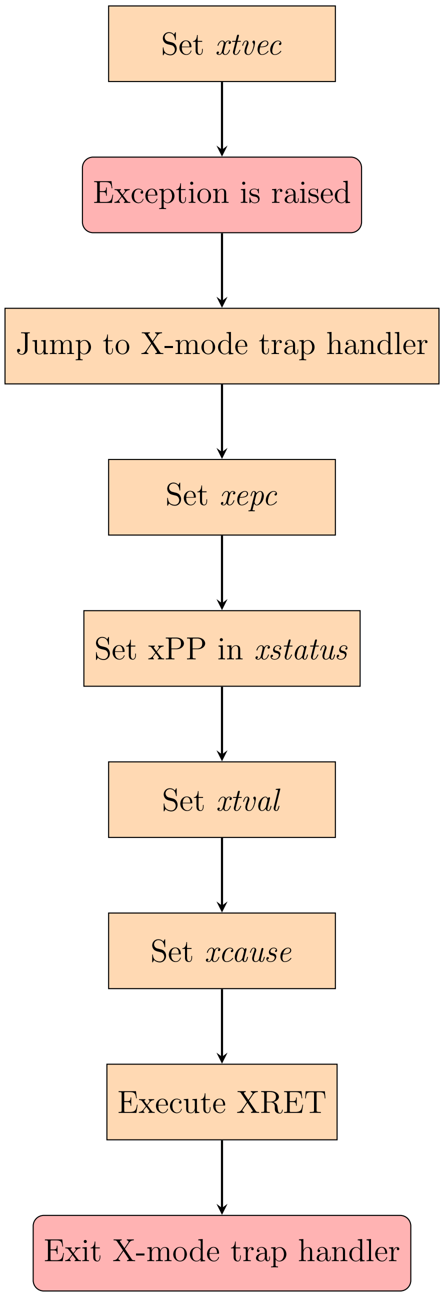

Figure 2: Flow Chart of Trap Handler Process

4 Exception Definitions

4.1 Misaligned Addresses

Misaligned instruction addresses is a misleading term that refers to misaligned branch and jump targets. In reality, the misaligned instruction address should be called misaligned branch/jump target. Misaligned instruction addresses are raised when the target of a jump or branch instruction is not aligned to a 4-byte boundary or an 8-byte boundary. 4-byte alignments are associated with RV32 systems and 8-byte alignments are associated with RV64 systems. 2-byte alignment are also possible for 16-bit instruction implementations but this is a customer instruction encoding. Customer instruction encodings refer to user-defined instruction implementations that are not a formal part of the RISC-V ISA. Misaligned loads/store addresses are raised when the data the load/store instruction accesses from memory is not aligned to the correct byte offset. In general, a data of size s bytes at byte address A is aligned if $A \bmod s = 0$.

4.2 Access Faults

Instruction, load, and store/AMO access faults are raised from failed physical memory protection checks. Physical memory protection checks verify that the instruction is accessed from a valid address in memory by the hardware. Attempting to fetch an instruction whose physical address lies in a PMP region that does not have execute permissions raises a fetch access exception. Attempting to execute a load or load-reserved instruction whose physical address lies within a PMP region without read permissions raises a load access exception. Attempting to execute a store, store-conditional (regardless of success), or AMO instruction whose physical address lies within a PMP region without write permissions raises a store access exception. Information about execute, read, and write privileges is stored in the lowest 3-bits of the page table entry.

4.3 Illegal Instructions

Illegal instructions are raised by illegal instruction encodings or problems reading from and writing to certain CSRs. The following is a non-exhaustive list of illegal instructions that could be raised:

- Instructions with bits [15:0] set to zero, this is a reserved bit pattern
- Instructions with bits [ILEN-1:0][1] set to one, this a reserved bit pattern
- Attempting to read and execute instructions that the processor does not recognize (ex. multiply or divide)
- Instructions that set rd = x0 with the exception of CSRRW, CSRRWI, JALR, and JAL
- Attempts to write to a read-only CSR
- Attempts to access CSRs from non-exsistent CSR addresses
- Attempts to access a CSR without appropriate privilege level permissions
- Machine-mode access of debug-mode CSRs

[1]ILEN is the maximum length of the maximum instruction length supported by an implementation. For RISC-V, ILEN is typically 32 or 64 bits.

- Attempts to read or write the satp CSR or execute the SFENCE.VMA instruction while executing in S-mode when the TVM bit in the *xstatus* CSR equals one
- Attempts to execute the WFI privilege instruction in any less-privileged mode, and it does not complete within an implementation-specific, bounded time limit[2] when the TW bit in the *mstatus* CSR equals one
- Attempts to execute SRET while executing in S-mode when the TSR bit in the *xstatus* register equals one
- Any instruction that attempts to read or write when the corresponding XS[1:0] field in the *xstatus* CSR is set to zero
- Attempts to read to the counter registers that correspond to the IR, TM, and CY bits in the *mcounteren* CSR when executing in a less-privileged mode

4.4 Environment Call and Environment Break

The environment call instruction (ECALL) is used to make a request to the supporting execution environment. When executed in U-mode, S-mode, or M-mode, it generates an environment-call-from-U-mode exception, environment-call-from-S-mode exception, or environment-call-from-M-mode exception, respectively, and performs no other operation. ECALL causes the receiving privilege mode's epc register to be set to the address of the ECALL instruction itself, not the address of the following instruction. On the other hand, the environment break instruction (EBREAK) raises an exception as part of the instruction execution.

4.5 Page Faults

For RV32 systems, the supervisor has a 32-bit, page-based virtual memory system called Sv32. RISC-V identifies three different types of page faults: instruction, load, and store/AMO. Instruction page faults are raised by attempting to fetch an instruction from a page that does not have execute permissions. Load page faults are raised by load or load-reserved instructions whose address lies within a page without read permissions. Store/AMO page faults are raised by store, store-conditional, or AMO instruction whose effective address lies within a page without write permissions. In other words, all instructions can raise instruction page faults, load instructions can only raise load page faults, and store instruction can only raise store page faults. This means that load and store page faults also raise instruction page faults.

The following lists a number of error that can raise a page fault exception that corresponds to the original access type (instruction, load, and store) during the Sv32 virtual to physical address translation process:

- The physical address of the page is insufficiently aligned
- A virtual page is accessed and the A bit in the PTE is cleared, or is raised and the D bit in the PTE is cleared
- PTE bit V = 0, or PTE bit R = 0 and PTE bit W = 1

[2]The time limit may always be zero, in which case WFI always causes an illegal instruction exception in less-privileged modes when TW equals one

- Performing more than two page walks (i.e $i < 0$ for $i = i - 1$ where $i = 1$ is the first walk and $i = 0$ is the second walk)
- The requested memory access is not allowed by PTE bits R, W, X and U, given the current privilege mode and the value of the SUM and MXR fields in the *mstatus* register
- PTE bit A = 0, or if the memory access is a store and PTE bit D = 0

5 Exception Table

The exceptions in Table 1 fall under the following categories fetch, load, store, misaligned branch/jump, rd != x0 illegal instruction, and ECALL/EBREAK. Fetch exceptions encompass instruction access faults and instruction page faults. Load exceptions encompass misaligned load addresses, load access faults, and load page faults. Store exceptions encompass misaligned store addresses, store access faults, and store page faults. Unless otherwise specified, the fetch, store, and load categories encompass all the respective exceptions included in each category. Misaligned branch/jump exceptions are raised by jump and branch instructions according to the definition for misaligned instruction addresses. EBREAK/ECALL exceptions are raised according to the definitions of environment call and environment break.

6 Interrupt Handling

Interrupt handling is similar to exception handling but there are some notable differences. Unlike exceptions, interrupts require additional hardware. This additional hardware includes a RISC-V Platform Level Interrupt Controller (PLIC) [1] and the option of a SiFive Core-local Interrupter (CLINT) [2] or a RISC-V Core-local Interrupt Controller (CLIC) [3]. The PLIC sources external interrupts from devices and sets the interrupt pending bit xEIP bit in the *xip* CSR if it is used with the CLINT as shown in Figure 3. Similarly, the CLINT sources local software and timer interrupts and sets the interrupt pending bits xTIP and xSIP in the *xip* CSR. Unlike the CLINT, the CLIC includes the external interrupt signals from the PLIC and sets the interrupt pending bits xEIP, xTIP, and xSIP as shown in Figure 4. RISC-V has defined a CLIC and PLIC specification that explains the interrupt control flow and defines the different registers that are used to handle interrupts. In addition to the CLINT and PLIC, interrupt handling also includes a *xie* CSR that has interrupt enable bits xTIE, xSIE, and xEIE. These bits are set by software and determine when a pending interrupt will be processed through the trap handler. For systems with multiple harts, the Wait for Interrupt (WFI) instruction can be implemented to control interrupt servicing between multiple harts by stalling a hart. If an enabled interrupt is present or later becomes present while the hart is stalled, the interrupt exception will be taken on the following instruction, i.e., execution resumes in the trap handler and *mepc* = pc + 4. The delegation process for interrupts is similar to the delegation process for exceptions. Instead of delegating through *medeleg* and *sedeleg*, interrupts delegate through *mideleg* and *sideleg*. Unlike exceptions, the mode select field in the *xtvec* CSR effects the trap handler address. Although direct mode handles interrupts at the base address, the vectored mode handles them at an address that equal to the base address plus four times interrupt cause code as shown in Table 2. The trap handler that is used to process exceptions is the same one that is used to process interrupts. Interrupts processed through this trap handler write to the same CSRs with the exception of *xtval*. This CSR is only written to when exceptions are handled by the trap handler.

Instruction	Fetch	Load	Store	Misaligned Branch/Jump	rd != x0	ECALL/EBREAK
LUI	×				×	
AUIPC	×				×	
JAL	×			×		
JALR	×			×		
BEQ	×			×		
BNE	×			×		
BLT	×			×		
BGE	×			×		
BLTU	×			×		
BGEU	×			×		
LB	×	×			×	
LH	×	×			×	
LW	×	×			×	
LBU	×	×			×	
LHU	×	×			×	
SB	×		×			
SH	×		×			
SW	×		×			
ADDI	×				×	
SLTI	×				×	
SLTIU	×				×	
XORI	×				×	
ORI	×				×	
ANDI	×				×	
SLLI	×				×	
SRLI	×				×	
SRAI	×				×	
ADD	×				×	
SUB	×				×	
SLL	×				×	
SLT	×				×	
SLTU	×				×	
XOR	×				×	
SRL	×				×	
SRA	×				×	
OR	×				×	
AND	×				×	
ECALL	×					×
EBREAK	×					×

Table 1: Possible Exceptions for the RV32I Instruction Set

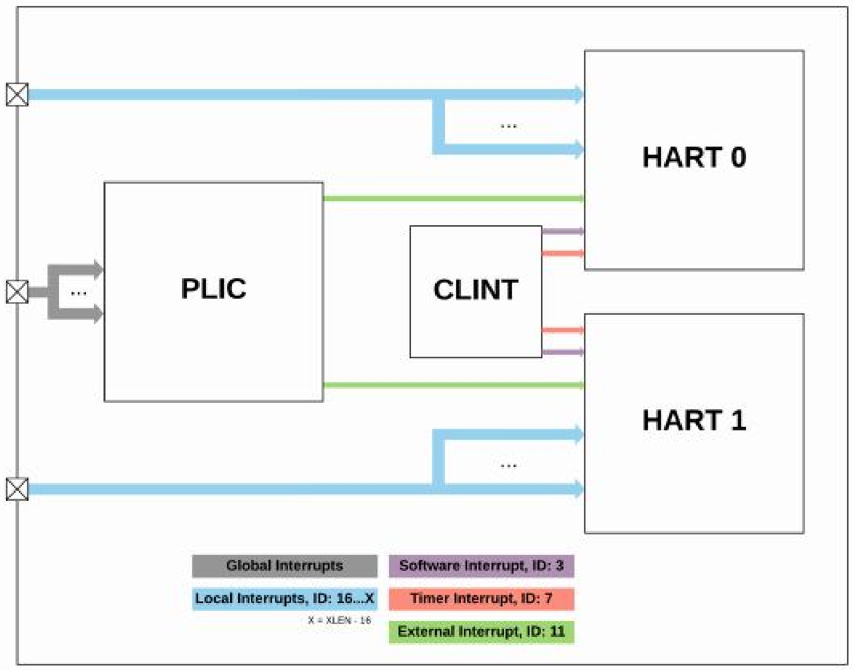
HART 0
PLIC
CLINT
HART 1
...
Global Interrupts
Local Interrupts, ID: 16...X
X = XLEN - 16
Software Interrupt, ID: 3
Timer Interrupt, ID: 7
External Interrupt, ID: 11

Figure 3: Block diagram of an Example PLIC and CLINT Configuration

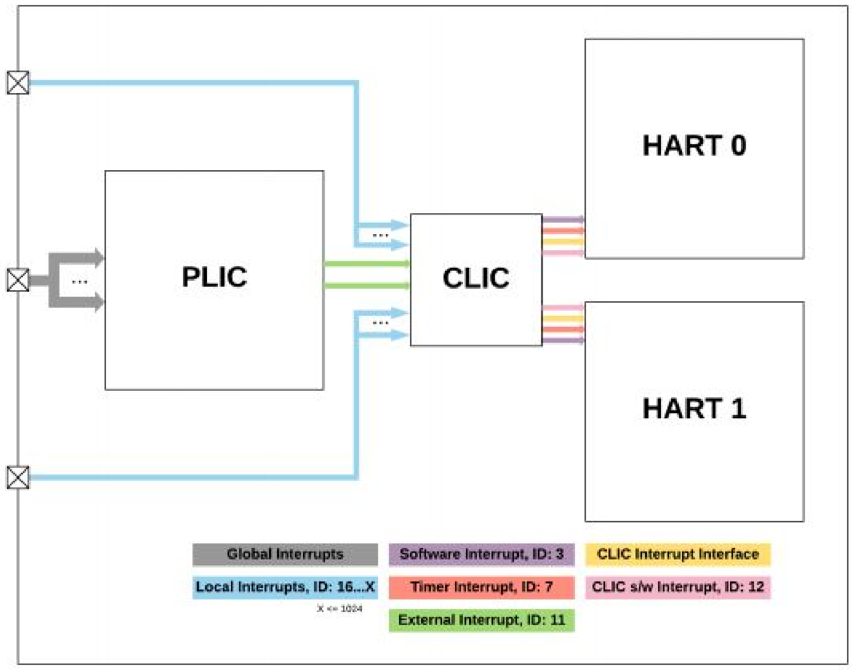
HART 0
PLIC
CLIC
HART 1
...
Global Interrupts
Local Interrupts, ID: 16...X
X <= 1024
Software Interrupt, ID: 3
Timer Interrupt, ID: 7
External Interrupt, ID: 11
CLIC Interrupt Interface
CLIC s/w Interrupt, ID: 12

Figure 4: Block diagram of an Example PLIC and CLIC Configuration

Interrupt	BASE + $4 * cause$
User Software Interrupt	BASE
Supervisor Software Interrupt	BASE + 0x4
Reserved	BASE + 0x8
Machine Software Interrupt	BASE + 0xC
User Timer Interrupt	BASE + 0x10
Supervisor Timer Interrupt	BASE + 0x14
Reserved	BASE + 0x18
Machine Timer Interrupt	BASE + 0x1C
User External Interrupt	BASE + 0x20
Supervisor External Interrupt	BASE + 0x24
Reserved	BASE + 0x28
Machine External Interrupt	BASE + 0x2C

Table 2: Interrupt Vector Table

[0]When vectored interrupts are enabled, interrupt cause 0, which corresponds to user-mode software interrupts, are vectored to the same location as synchronous exceptions. This ambiguity does not arise in practice, since user-mode software interrupts are either disabled or delegated to a less-privileged mode.

References

[1] RISC-V PLIC Specification. RISC-V Foundation. [Online]. Available: https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc

[2] SiFive Interrupt Cookbook. SiFive. [Online]. Available: https://sifive.cdn.prismic.io/sifive/0d163928-2128-42be-a75a-464df65e04e0_sifive-interrupt-cookbook.pdf

[3] RISC-V CLIC Specification. RISC-V Foundation. [Online]. Available: https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.adoc