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 to be 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). Each privilege mode supports a set of privilege-specific CSRs for the privilege-specific trap handlers. 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. RISC-V processors must handle each of these exception meaning select exceptions cannot be disabled. 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. RISC-V processors have the option of handling certain interrupts meaning select interrupts can be enabled or disabled (see Section 6.2).

3 Exception Handling

3.1 M-Mode Exceptions

If an exception occurs, control is relinquished by the program that is currently being executed and transferred to the exception handler. The exception 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 exception handler must be initialized. By default, M-mode, S-mode, and U-mode exceptions are handled by the M-mode exception handler. Software initializes the M-mode exception handler by setting the `mtvec` CSR (see Figure 6 in the Appendix). The `mtvec` CSR contains a base address and a mode field. This mode field supports two options: direct and vectored. For both modes, the program counter is set to the base address in `mtvec`. This is where the M-mode exception handler is located.

On entry to the M-mode exception handler, the current execution environment is interrupted and software sets the `mepc`, `mstatus`, `mcause`, and `mtval` CSRs before resuming execution at the instruction that raised the exception. The `mepc` CSR (see Figure 7 in the Appendix) is written with the value the program counter was set to for the instruction that took the exception. This value is stored so that the previous execution environment can be returned to and resumed once the exception handler is finished. In M-mode, software changes the MPP, MPIE and MIE fields (see Section 6.2 for more information) in the `mstatus` CSR (see Figure 8 in the Appendix). `mstatus.MPP` is a 2-bit field that is written with the privilege level encoding that corresponds to the privilege level the program was in when the exception was raised. `mstatus.MPP` can hold three values: 11 (M-mode), 01 (S-mode), and 00 (U-mode). `mstatus.MPIE` is a 1-bit field that is written with the value the 1-bit, M-mode global interrupt-enable field `mstatus.MIE` held when the exception was raised. Once `mstatus.MPIE` is written with the value in `mstatus.MIE`, the `mstatus.MIE` field is set to zero so that the exception handler does not attempt to process an interrupt while the current trap is being handled. The `mcause` CSR (see Figure 10 in the Appendix) is written with the exception cause code that corresponds to the exception that was raised. If an instruction raises multiple synchronous exceptions, the exceptions are taken by the exception handler and reported in `mcause` according to a pre-defined priority structure. The `mtval` CSR (see Figure 9 in the Appendix) is written with exception-specific information. For misaligned addresses, access faults, and page faults, `mtval` will contain the faulting virtual address (see Section 4 for more information). Conversely, illegal instructions set `mtval` to the faulting instruction whereas EBREAKs and ECALLs set `mtval` to zero. If software intends to exit the M-mode exception handler and return to the location that raised the exception after these CSRs are set, it must first determine whether any register states were corrupted during the service routine and resolve these critical errors. Once this is done, software may execute a MRET instruction which sets the program counter to the value stored in `mepc` and resumes the previous execution environment. In addition to setting the program counter, MRET also sets `mstatus.MIE` to the value in `mstatus.MPIE` before setting `mstatus.MPIE` to one to indicate that the exception handler is ready for the next interrupt.

3.2 S-Mode Exceptions

Exceptions raised in S-mode can be handled in M-mode or delegated to S-mode through the `medeleg` CSR (see Figure 11 in the Appendix). Delegating to S-mode means that only S-mode CSRs are visible to software and exceptions are processed by the S-mode exception handler. The `medeleg` CSR delegates exceptions by raising the bits in positions that correspond to exception code numbers as shown in Figure 1.

MXLEN-1 16	15 0
WARL	Code 15 - Code 0
MXLEN-16	16

Figure 1: Machine Exception Delegation Register `medeleg`.

Software initializes the S-mode exception handler by setting the `stvec` CSR (see Figure 12 in the Appendix). The `stvec` CSR contains a base address and a mode field. This mode field supports two options: direct and vectored. For both modes, the program counter is set to the base address in `stvec`. This is where the S-mode exception handler is located.

On entry to the S-mode exception handler, the current execution environment is interrupted and software sets the `sepc`, `mstatus`, `scause`, and `stval` CSRs before resuming execution at the instruction that raised the exception. The `sepc` CSR (see Figure 13 in the Appendix) is written with the value the program counter was set to for the instruction that took the exception. This value is stored so that the previous execution environment can be returned to and resumed once the exception handler is finished. In S-mode, software changes the SPP, SPIE and SIE fields (see Section 6.2 for more information) in the `mstatus`[1] CSR. `mstatus.SPP` is a 1-bit field that is written with the privilege level encoding that corresponds to the privilege level the program was in when the exception was raised. `mstatus.SPP` can hold two values: 1 (S-mode) and 0 (U-mode). `mstatus.SPIE` is a 1-bit field that is written with the value the 1-bit, S-mode global interrupt-enable field `mstatus.SIE` held when the exception was raised. Once `mstatus.SPIE` is written with the value in `mstatus.SIE`, the `mstatus.SIE` field is set to zero so that the exception handler does not attempt to process an interrupt while the current trap is being handled. The `scause` CSR (see Figure 15 in the Appendix) is written with the exception cause code that corresponds to the exception that was raised. If an instruction raises multiple synchronous exceptions, the exceptions are taken by the exception handler and reported in `scause` according to a pre-defined priority structure. The `stval` CSR (see Figure 14 in the Appendix) is written with exception-specific information. For misaligned addresses, access faults, and page faults, `stval` will contain the faulting virtual address (see Section 4 for more information). Conversely, illegal instructions set `stval` to the faulting instruction whereas EBREAKs and ECALLs set `stval` to zero. If software intends to exit the S-mode exception handler and return to the location that raised the exception after these CSRs are set, it must first determine whether any register states were corrupted during the service routine and resolve these critical errors. Once this is done, software may execute a SRET instruction which sets the program counter to the value stored in `sepc` and resumes the previous execution environment. In addition to setting the program counter, SRET also sets `mstatus.SIE` to the value in `mstatus.SPIE` before setting `mstatus.SPIE` to one to indicate that the exception handler is ready for the next interrupt.

3.3 U-Mode Exceptions

Exceptions raised in U-mode can be handled in M-mode, delegated to S-mode through the `medeleg` CSR, or delegated to U-mode through the `sedeleg` CSR (see Figure 16 in the Appendix). Delegating to U-mode means that only U-mode CSRs are visible to software and exceptions are processed by the U-mode exception handler as shown in Figure 3. The `sedeleg` CSR delegates exceptions that first must be delegated by the `medeleg` CSR by raising the bits in positions that correspond to exception code numbers as shown in Figure 2.

SXLEN-1 16	15 0
0	Code 15 - Code 0
SXLEN-16	16

Figure 2: Machine Exception Delegation Register `sedeleg`.

Software initializes the U-mode exception handler by setting the `utvec` CSR (see Figure 17 in the Appendix). The `utvec` CSR contains a base address and a mode field. This mode field supports two options: direct and vectored. For both modes, the program counter is set to the base address in `utvec`. This is where the U-mode exception handler is located.

On entry to the U-mode exception handler, the current execution environment is interrupted and software sets the `uepc`, `mstatus`, `ucause`, and `utval` CSRs before resuming execution at the

[1]S-mode and U-mode can utilize `mstatus` instead of `sstatus` and `ustatus`. These CSRs simply restrict the higher privilege level fields from being visible to hardware.

instruction that raised the exception. The `uepc` CSR (see Figure 18 in the Appendix) is written with the value the program counter was set to for the instruction that took the exception. This value is stored so that the previous execution environment can be returned to and resumed once the exception handler is finished. In U-mode, software changes the UPIE and UIE fields (see Section 6.2 for more information) in the `mstatus` CSR. The `mstatus` CSR does not include a UPP field for U-mode since it is implicitly set to zero. `mstatus.UPIE` is a 1-bit field that is written with the value the 1-bit, S-mode global interrupt-enable field `mstatus.UIE` held when the exception was raised. Once `mstatus.UPIE` is written with the value in `mstatus.UIE`, the `mstatus.UIE` field is set to zero so that the exception handler does not attempt to process an interrupt while the current trap is being handled. The `ucause` CSR (see Figure 20 in the Appendix) is written with the exception cause code that corresponds to the exception that was raised. If an instruction raises multiple synchronous exceptions, the exceptions are taken by the exception handler and reported in `ucause` according to a pre-defined priority structure. The `utval` CSR (see Figure 19 in the Appendix) is written with exception-specific information. For misaligned addresses, access faults, and page faults, `utval` will contain the faulting virtual address (see Section 4 for more information). Conversely, illegal instructions set `utval` to the faulting instruction whereas EBREAKs and ECALLs set `utval` to zero. If software intends to exit the U-mode exception handler and return to the location that raised the exception after these CSRs are set, it must first determine whether any register states were corrupted during the service routine and resolve these critical errors. Once this is done, software may execute a URET instruction which sets the program counter to the value stored in `uepc` and resumes the previous execution environment. In addition to setting the program counter, URET also sets `mstatus.UIE` to the value in `mstatus.UPIE` before setting `mstatus.UPIE` to one to indicate that the exception handler is ready for the next interrupt.

3.4 Exception Priority

If an instruction raises multiple synchronous exceptions, the decreasing priority order of Table 1 indicates which exception is taken. Synchronous exceptions are of lower priority than all interrupts and the priority of any custom synchronous exceptions is implementation-defined.

Priority	Exception Code	Description
Highest	12	Instruction Page Fault
	1	Instruction Access Fault
	2	Illegal Instruction
	0	Instruction Address Misaligned
	8, 9, 11	Environment Call (U, S, M)
	3	Environment Break
	6	Store/AMO Address Misaligned
	4	Load Address Misaligned
	15	Store/AMO Page Fault
	13	Load Page Fault
	7	Store/AMO Access Fault
Lowest	5	Load Access Fault

Table 1: Exception Fault Ordering

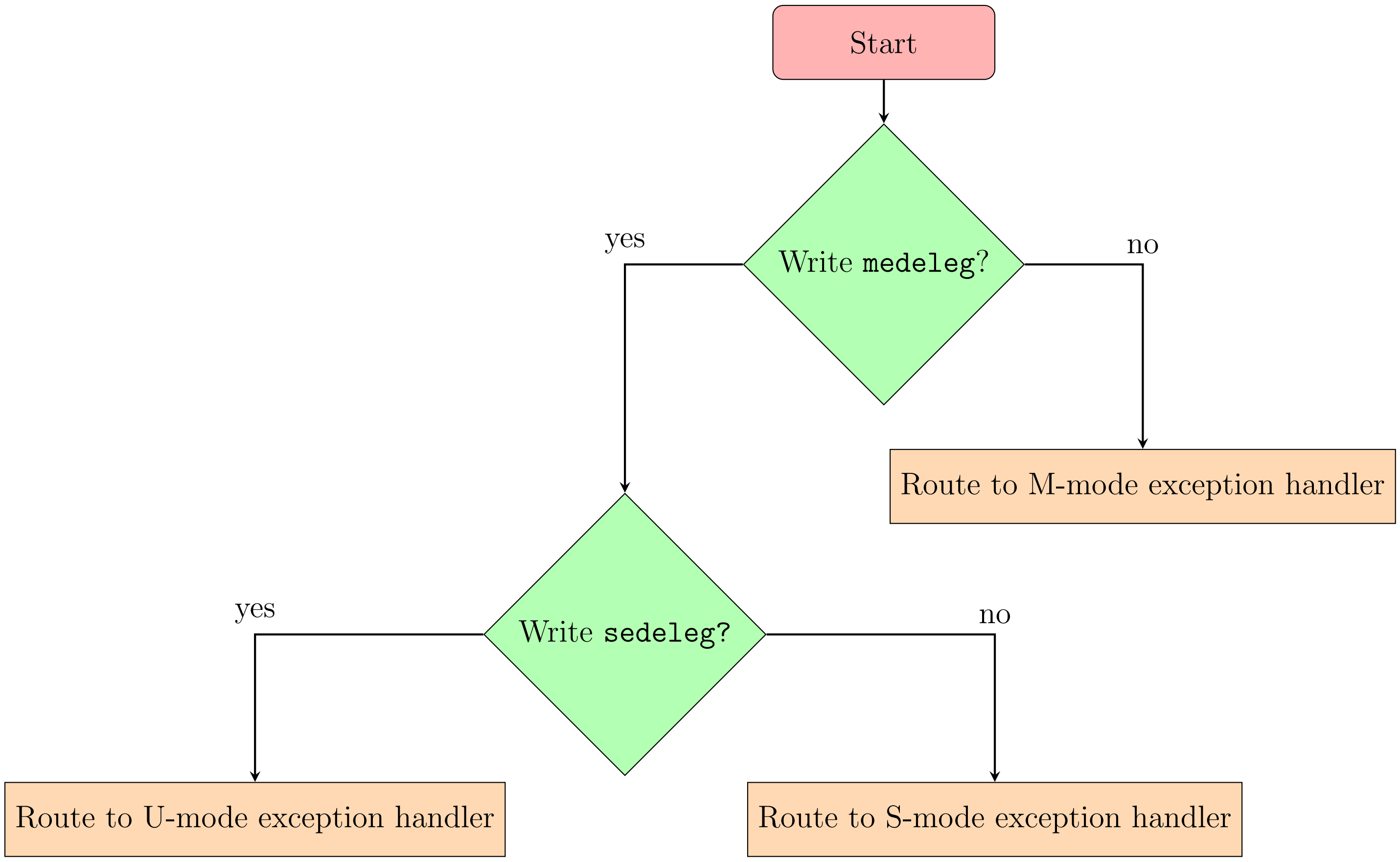

Start
Write medeleg?
yes
no
Route to M-mode exception handler
Write sedeleg?
yes
no
Route to U-mode exception handler
Route to S-mode exception handler

Figure 3: Flow Chart of Delegation Process

4 Exception Definitions

4.1 Misaligned Addresses

Misaligned instruction addresses is a misleading term that refers to misaligned branch and jump targets. 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 alignments are also possible for 16-bit instruction implementations but this is a customer instruction encoding that is 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 (PMP) checks verify that the instruction is accessed from a valid address in memory by the hardware. Access fault exceptions rely on bits L, X, R, and W in an 8-bit PMP configuration register. Bits X, R, and W encode execute, read, and write privileges respectively. The L-bit indicates that the PMP entry is locked, i.e., writes to the configuration register (see Figure 21 in the Appendix) and associated address registers are ignored. In addition to locking the PMP entry, the L bit indicates whether the R/W/X permissions are enforced on M-mode accesses. When the L bit is set, these permissions are enforced for all privilege modes. When the L bit is clear, any M-mode access matching the PMP entry will succeed and the R/W/X permissions only apply to S and U modes. Access fault exceptions can only be raised if the L-bit is set or the access is in S-mode or U-mode. Attempting to fetch an instruction whose physical address lies in a PMP region that does not have execute permissions ($X = 0$) 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 ($R = 0$) 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 ($W = 0$) raises a store access exception.

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][2] set to one, this a reserved bit pattern
- Bit patterns 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
- 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.TVM` = 1
- Attempts to execute the WFI privilege instruction in any less-privileged mode, and it does not complete within an implementation-specific, bounded time limit[3] when `mstatus.TW` = 1
- Attempts to execute SRET while executing in S-mode when `xstatus.TSR` = 1
- Any instruction that attempts to read or write when `mstatus.XS[1:0]` = 0
- 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

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

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

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. Similarly, 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
- PTE.V = 0, or PTE.R = 0 and PTE.W = 1
- R, W, and X are zero when the page walk is on level two
- Accessing a page in U-mode when the PTE.U != 1
- Attempting to execute S-mode code on page where the PTE bit U = 1
- Attempting to execute loads from pages where R = 0 and X = 0 for `mstatus.MXR` = 1
- S-mode accesses of pages that are accessible in U-mode (U = 1) when `mstatus.SUM` = 0
- PTE.A = 0, or if the memory access is a store and PTE.D = 0

5 Exception Table

The exceptions in Table 2 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.

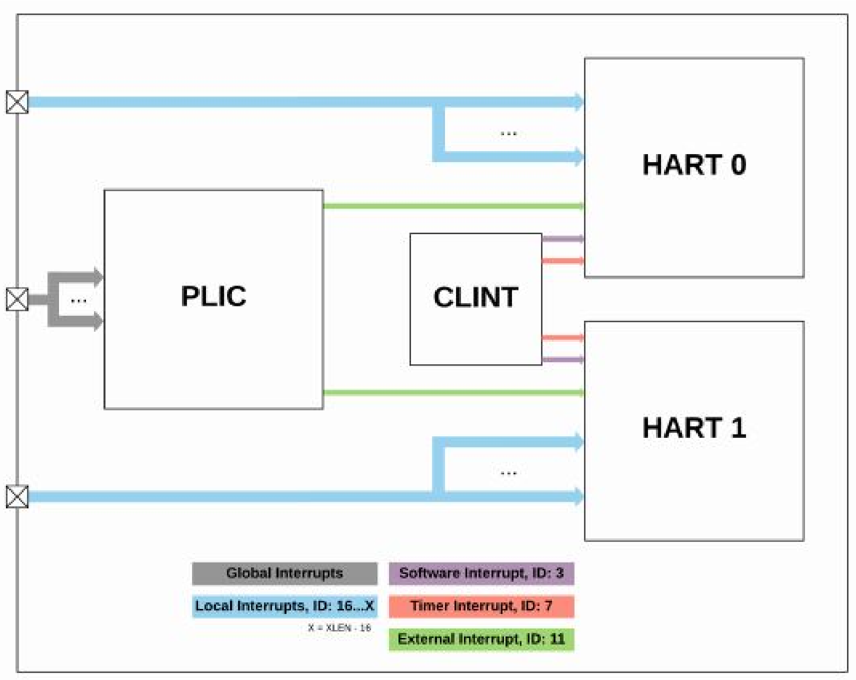
Instruction	Fetch	Load	Store	Misaligned Branch/Jump	rd != x0	ECALL/EBREAK
LUI	×				×	
AUIPC	×				×	
JAL	×			×		
JALR	×			×		
BEQ	×			×		
BNE	×			×		
BLT	×			×		
BGE	×			×		
BLTU	×			×		
BGEU	×			×		
LB	×	×			×	
LH	×	×			×	
LW	×	×			×	
LBU	×	×			×	
LHU	×	×			×	
SB	×		×			
SH	×		×			
SW	×		×			
ADDI	×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

Table 2: Possible Exceptions for the RV32I Instruction Set

6 Interrupt Handling

6.1 PLIC, CLINT, and CLIC

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 routes them to the hart(s) as shown in Figure 4. Similarly, the CLINT sources local software and timer interrupts and routes them to the hart(s). Unlike the CLINT, the CLIC routes external, software, and timer interrupt signals to the hart(s) as shown in Figure 5. RISC-V has defined detailed CLIC and PLIC specifications that explain the interrupt control flow and define the different registers that are used to handle interrupts. 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.



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 4: Block diagram of an example PLIC and CLINT configuration

6.2 Interrupt Control Flow

Unlike exceptions, interrupts do not depend on synchronous hardware problems in the processor. Instead, interrupts are asynchronous events external to the processor that are driven devices, timers, or software. Before information about interrupts can be routed from the interrupt controllers to the processor, they must be globally and locally enabled. Global interrupt enables are controlled by the `mstatus.xie` fields and the current privilege mode. M-mode interrupts are globally enabled if the current privilege mode is less than M or the current privilege mode is M and `mstatus.MIE` = 1. S-mode interrupts are globally enabled if the current privilege mode is less the S or the current

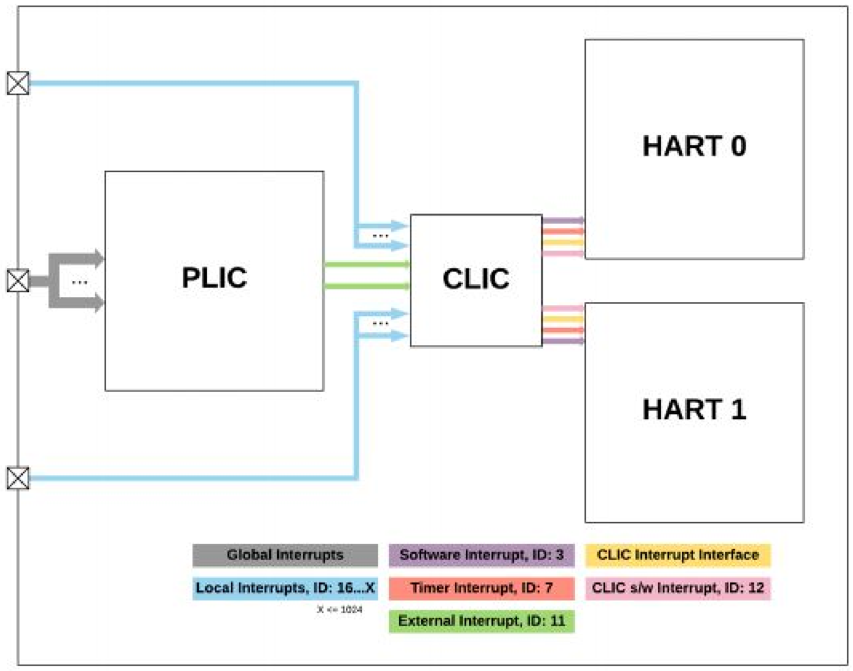

HART 0
PLIC
CLIC
HART 1
...
Global Interrupts
Software Interrupt, ID: 3
CLIC Interrupt Interface
Local Interrupts, ID: 16...X
Timer Interrupt, ID: 7
CLIC s/w Interrupt, ID: 12
X <= 1024
External Interrupt, ID: 11

Figure 5: Block diagram of an example PLIC and CLIC configuration

privilege mode is S and `mstatus.SIE` = 1. U-mode interrupts are globally enabled if the current privilege mode is U and `mstatus.UIE` = 1. Local interrupts are controlled by the `mie` CSR (see Figure 22 in the Appendix). External interrupts in M-mode, S-mode, and U-mode are enabled by raising the 1-bit `mie.MEIE`, `mie.SEIE`, and `mie.UEIE` fields respectively. Timer interrupts in M-mode, S-mode, and U-mode are enabled by raising the 1-bit `mie.MTIE`, `mie.STIE`, and `mie.UTIE` fields respectively. Software interrupts in M-mode, S-mode, and U-mode are enabled by raising the 1-bit `mie.MSIE`, `mie.SSIE`, and `mie.USIE` fields respectively.

Once the selected interrupts are globally and locally enabled, the interrupt controllers are responsible for identifying pending interrupts and raising the appropriate fields in the `mip` CSR (see Figure 23 in the Appendix). M-mode external, software, and timer interrupts are raised by setting the 1-bit `mip.MEIP`, `mip.MSIP`, and `mip.MTIP` fields respectively. S-mode external, software, and timer interrupts are raised by setting the 1-bit `mip.SEIP`, `mip.SSIP`, and `mip.STIP` fields respectively. U-mode external, software, and timer interrupts are raised by setting the 1-bit `mip.UEIP`, `mip.USIP`, and `mip.UTIP` fields respectively.

6.3 M-Mode Interrupts

If M-mode, S-mode, or U-mode bits in `mip` are raised, control is relinquished by the program that is currently being executed and transferred to an interrupt handler. By default, M-mode, S-mode, and U-mode interrupts are handled by the M-mode interrupt handler. Software initializes the M-mode interrupt handler by setting the `mtvec` CSR. The `mtvec` CSR contains a base address and a mode field. This mode field supports two options: direct and vectored. In direct mode, the program counter is set to the base address in `mtvec`. For vectored mode, the program counter is set to the base address plus four times the interrupt cause code as shown in Table 3[4]. This can

[4]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

be thought of as a jump table that contains different jump targets for interrupt-specific handling routines. Once the M-mode interrupt handler is initialized, interrupts are processed in the same manner as exceptions as specified in Section 3.1. The only difference is that interrupts do not set `mtval` and it is therefore left cleared.

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 3: Interrupt Vector Table

6.4 S-Mode Interrupts

If M-mode, S-mode, or U-mode bits in `mip` are raised, control is relinquished by the program that is currently being executed and transferred to an interrupt handler. Interrupts raised in S-mode can be handled in M-mode or delegated to S-mode through the `mideleg` CSR. Delegating to S-mode means that only S-mode CSRs are visible in hardware and interrupts are processed by the S-mode interrupt handler. The S-mode interrupt handler can process S-mode and U-mode interrupts. Software initializes the S-mode interrupt handler by setting the `stvec` CSR. The `stvec` CSR contains a base address and a mode field. This mode field supports two options: direct and vectored. In direct mode, the program counter is set to the base address in `stvec`. For vectored mode, the program counter is set to the base address plus four times the interrupt cause code. This can be thought of as a jump table that contains different jump targets for interrupt-specific handling routines. Once the S-mode interrupt handler is initialized, interrupts processed by the S-mode interrupt handler are processed in the same manner as exceptions as specified in Section 3.2. The only difference is that interrupts do not set `stval` and it is therefore left cleared.

6.5 U-Mode Interrupts

If M-mode, S-mode, or U-mode bits in `mip` are raised, control is relinquished by the program that is currently being executed and transferred to an interrupt handler. Interrupts raised in U-mode can be handled in M-mode, delegated to S-mode through the `mideleg` CSR, or delegated to U-mode through the *sideleg* CSR. Delegating to U-mode means that only U-mode CSRs are visible in hardware and interrupts are processed by the U-mode interrupt handler. The U-mode interrupt handler can only process U-mode interrupts. Software initializes the U-mode interrupt handler by

software interrupts are either disabled or delegated to a less-privileged mode.

setting the `utvec` CSR. The `utvec` CSR contains a base address and a mode field. This mode field supports two options: direct and vectored. In direct mode, the program counter is set to the base address in `utvec`. For vectored mode, the program counter is set to the base address plus four times the interrupt cause code. This can be thought of as a jump table that contains different jump targets for interrupt-specific handling routines. Once the M-mode interrupt handler is initialized, interrupts processed by the U-mode interrupt handler are processed in the same manner as exceptions as specified in Section 3.3. The only difference is that interrupts do not set `utval` and it is therefore left cleared.

6.6 Interrupt Priority

Like exceptions, simultaneous interrupts are processed by the interrupt handler according to a fixed priority structure as show in Table 4. The exception codes for interrupts are differentiated from the exception codes for exceptions with a 1-bit interrupt field in the MSB of `xcause`. Raising this field indicates that the exception code in `xcause` is for an interrupt.

Priority	Exception Code	Description
Highest	11	Machine External Interrupt
	3	Machine Software Interrupt
	7	Machine Timer Interrupt
	9	Supervisor External Interrupt
	1	Supervisor Software Interrupt
	5	Supervisor Timer Interrupt
	8	User External Interrupt
	0	User Software Interrupt
Lowest	4	User Timer Interrupt

Table 4: Interrupt Priorities

7 CSR Appendix

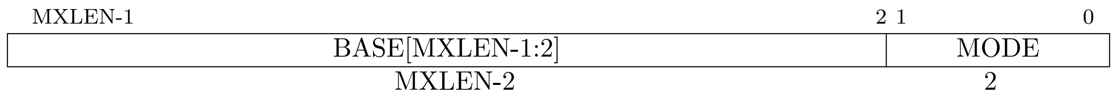


Figure 6: Machine trap-vector base-address register (`mtvec`).

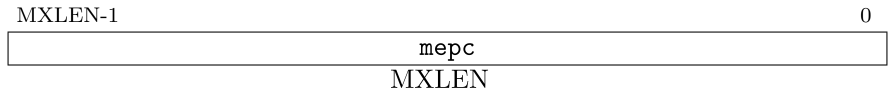


Figure 7: Machine exception program counter register.

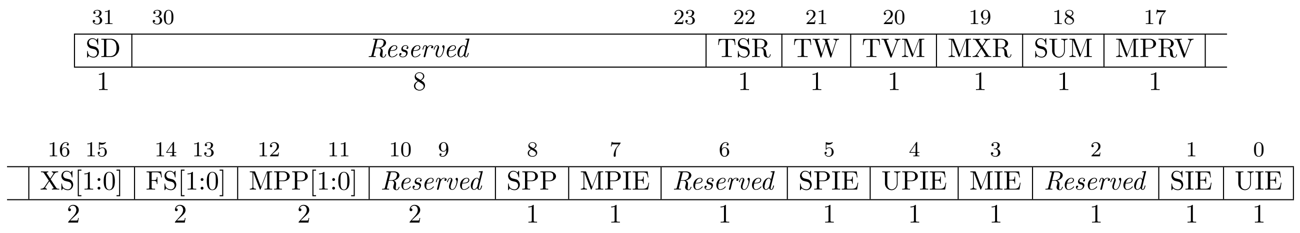


Figure 8: Machine-mode status register (`mstatus`) for RV32.

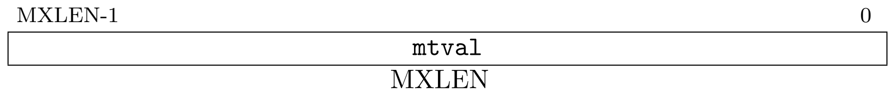


Figure 9: Machine Trap Value register.

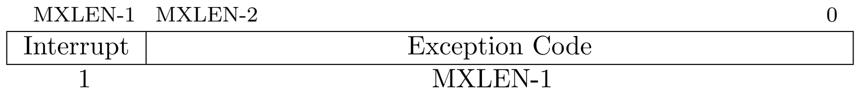


Figure 10: Machine Cause register `mcause`.

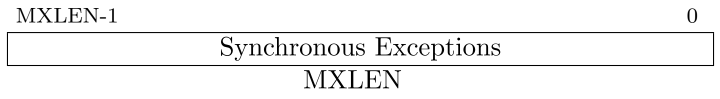


Figure 11: Machine Exception Delegation Register `medeleg`.

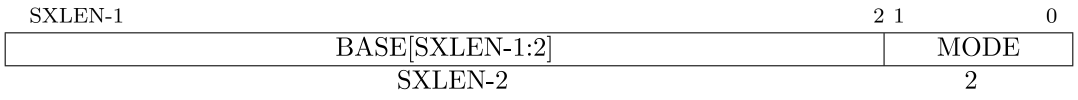


Figure 12: Supervisor trap vector base address register (`stvec`).

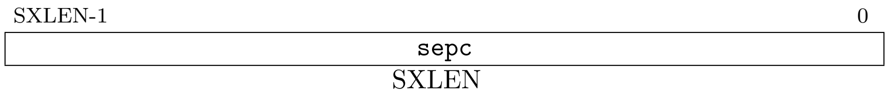


Figure 13: Supervisor exception program counter register.

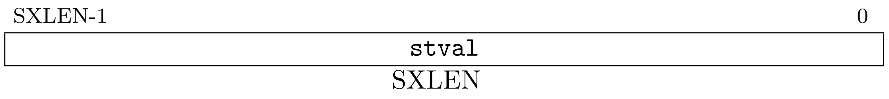


Figure 14: Supervisor Trap Value register.

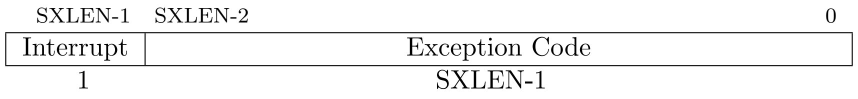


Figure 15: Supervisor Cause register `scause`.

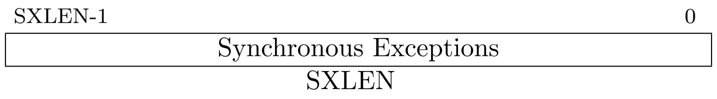


Figure 16: Supervisor Exception Delegation Register `sedeleg`.

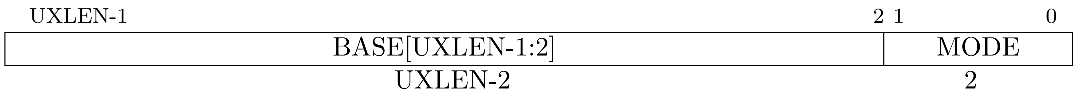


Figure 17: User trap vector base address register (`utvec`).

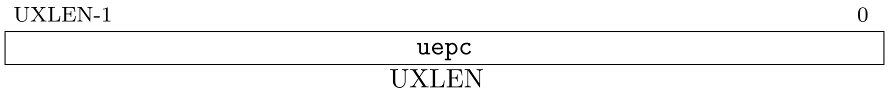


Figure 18: User exception program counter register.

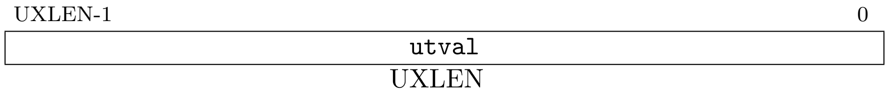


Figure 19: User Trap Value register.

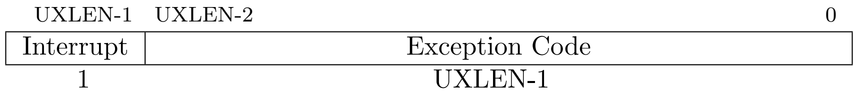


Figure 20: User Cause register `ucause`.

7	6 5	4 3	2	1	0
L	0	A	X	W	R
1	2	2	1	1	1

Figure 21: PMP configuration register format.

MXLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIE	0	SEIE	0	MTIE	0	STIE	0	MSIE	0	SSIE	0
MXLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figure 22: Machine Interrupt Enable Register `mie`.

MXLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIP	0	SEIP	0	MTIP	0	STIP	0	MSIP	0	SSIP	0
MXLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figure 23: Machine Interrupt Pending Register `mip`.

MXLEN-1 0
Interrupts
MXLEN

Figure 24: Machine Interrupt Delegation Register `mideleg`.

SXLEN-1 0
Interrupts
SXLEN

Figure 25: Supervisor Interrupt Delegation Register `sideleg`.

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