The RISC-V Advanced Interrupt Architecture

Editors: John Hauser, jh.riscv@jhauser.us

Version 1.0, Revised 20250312

Contributors to all versions of this specification in alphabetical order (please contact the editor to suggest corrections): Krste Asanović, Paul Donahue, Greg Favor, John Hauser, James Kenney, David Kruckemyer, Shubu Mukherjee, Stefan O'Rear, Vernon Pang, Anup Patel, Josh Scheid, Ved Shanbhogue, and Andrew Waterman.

This document describes an Advanced Interrupt Architecture (AIA) for RISC-V systems. This specification was ratified by the RISC-V International Association in June of 2023.

The table below indicates which chapters of this document specify extensions to the RISC-V ISA (instruction set architecture) and which are non-ISA.

Chapter	ISA?
1. Introduction	—
2. Control and Status Registers (CSRs) Added to Harts	Yes
3. Incoming MSI Controller (IMSIC)	Yes
4. Advanced Platform-Level Interrupt Controller (APLIC)	No
5. Interrupts for Machine and Supervisor Levels	Yes
6. Interrupts for Virtual Machines (VS Level)	Yes
7. Interprocessor Interrupts (IPIs)	No
8. IOMMU Support for MSIs to Virtual Machines	No

Changes for version 20250312

Made the following clarifications to AIA 1.0:

- Where there are irreconcilable conflicts between the AIA and other implemented RISC-V extensions, the AIA usually has priority by default.
- Deference is given to extension Smcsrind/Sscsrind (indirectly accessed CSRs).
- Names are given to the bits defined in `mstateen0` and `hstateen0` when extension Smstateen/Ssstateen is also implemented.
- An IMSIC interrupt file's `eidelivery` register affects only whether an interrupt appears in a hart's `mip` or `hgeip` register.
- IMSIC CSRs `mtopei`, `stopei`, and `vstopei` are not affected by the values of `mie`, `sie`, `hie`, `hgeie`, or `vsie`.
- There may be a visible delay between a change of state of an IMSIC interrupt file and its effect on a bit in `mip`, `sip`, or `hgeip`.
- An APLIC's `idelivery` registers and the IE bits of its `domaincfg` registers affect only whether pending-and-enabled interrupts are delivered to harts.
- The default priority order for major interrupts is applicable only when multiple interrupts would trap to the same privilege mode.
- The example pseudocode given for handling major interrupts at M-level and S-level has additional requirements not mentioned previously.
- An interrupt priority number in the S-level `iprio` array may be writable (not read-only zero) if the correponding bit is writable in either `sie` or `hie`.
- If a supervisor external interrupt (SEI) is injected from M-level when there is no actual interrupt from an external interrupt controller, the injected SEI is assigned an S-level priority number of 256.

- CSR `hvictl` affects only `vstopi` and the trapping of some instructions, not `mip`, `sip`, `hip`, or `vsip`.

Changes for the ratified version 1.0

Resolved some inconsistencies in Chapter 2 about when to raise a virtual instruction exception versus an illegal instruction exception.

Changes for RC5 (Release Candidate 5)

Better aligned the rules for indirectly accessed registers with the hypervisor extension and with forthcoming extension Smcsrind/Sscsrind. In particular, when `vsiselect` has a reserved value, attempts to access `sireg` from a virtual machine (VS or VU-mode) should preferably raise an illegal instruction exception instead of a virtual instruction exception.

Added clarification about the term *IOMMU* used in Chapter 8.

Added clarification about MSI write replaced by MRIF update and notice MSI sent after the update.

Changes for RC4

For alignment with other forthcoming RISC-V ISA extensions, the widths of the indirect-access CSRs, `miselect`, `mireg`, `siselect`, `sireg`, `vsiselect`, and `vsireg`, were changed to all be the current XLEN rather than being tied to their respective privilege levels (previously MXLEN for `miselect` and `mireg`, SXLEN for `siselect` and `sireg`, and VSXLEN for `vsiselect` and `vsireg`).

Changed the description (but not the actual function) of *high-half* CSRs and their partner CSRs to match the latest RISC-V Privileged ISA specification. (An example of a high-half CSR is `miph`, and its partner here is `mip`.)

Changes for RC3

Removed the still-draft Duo-PLIC chapter to a separate document.

Allocated major interrupts 35 and 43 for signaling RAS events (Section 5.1).

In Section 5.3 added the options for bits 1 and 9 to be writable in CSR `mvien`, and specified the effects of setting each of these bits.

Upgraded Chapter 8 ("IOMMU Support") to the *frozen* state.

Changes for RC2

Clarified that field IID of CSR `hvictl` must support all unsigned integer values of the number of bits implemented for that field, and that writes to `hvictl` always set IID in the most straightforward way.

A comment was added to Chapter 7 warning about the possible need for FENCE instructions when IPIs are sent to other harts by writing MSIs to those harts' IMSICs.

Table of Contents

Chapter 1. Introduction

This document specifies the Advanced Interrupt Architecture for RISC-V, consisting of: (a) an extension to the standard RISC-V Privileged Architecture; (b) two standard interrupt controllers for RISC-V systems, an Advanced Platform-Level Interrupt Controller (APLIC) and an Incoming Message-Signaled Interrupt Controller (IMSIC); and (c) requirements on other system components concerning interrupts.

Commentary on our design decisions, implementation options, and application is formatted as in this paragraph, and can be skipped if the reader is only interested in the specification itself.

1.1. Goals

The RISC-V Advanced Interrupt Architecture has these goals:

- Build upon the interrupt-handling functionality of the RISC-V Privileged Architecture, minimizing the replacement of existing functionality.
- Provide facilities for RISC-V systems to work directly with message-signaled interrupts (MSIs) as employed by PCI Express and other device standards, in addition to basic wired interrupts.
- For wired interrupts, define a new Platform-Level Interrupt Controller (the Advanced PLIC, or APLIC) that has an independent control interface for each level of privilege (such as RISC-V machine and supervisor levels), and that can convert wired interrupts into MSIs for systems supporting MSIs.
- Expand the framework for local interrupts at a RISC-V hart.
- Optionally allow software to configure the relative priorities of all sources of interrupts to a RISC-V hart (including the standard timer and software interrupts, among others), instead of being limited just to the ability of a separate interrupt controller to prioritize external interrupts only.
- When harts implement the Privileged Architecture's H extension, provide sufficient assistance for virtualizing these same interrupt facilities for virtual machines.
- With the help of an IOMMU (I/O memory management unit) for redirecting MSIs, maximize the opportunities and ability for a guest operating system running in a virtual machine to have direct control of devices with minimal involvement of a hypervisor.
- Avoid having the interrupt hardware be a limiter on the number of virtual machines.
- Achieve all of the above with the best possible compromises between speed, efficiency, and flexibility of implementation.

This initial version of the Advanced Interrupt Architecture is focused primarily on the needs of larger, high-performance RISC-V systems. Support is not currently defined for the following interrupt-handling features that are useful for minimizing interrupt response times in so-called "real-time" systems but are less appropriate for high-speed processor cores:

- the option to give each interrupt source at a hart a separate trap entry address;
- automatic stacking of register values on interrupt trap entry, and restoration on exit; and
- automatic preemption (nesting) of interrupts at a hart, based on priority.

It is intended that such features optimizing for smaller and/or real-time systems can be developed as a

follow-on extension, either separately or as part of a future version of the interrupt architecture of this document.

1.2. Limits

In its current version, the RISC-V Advanced Interrupt Architecture can support RISC-V symmetric multiprocessing (SMP) systems with up to 16,384 harts. If the harts are 64-bit (RV64) and implement the H extension, and if all features of the Advanced Interrupt Architecture are fully implemented as well, then for each physical hart there may be up to 63 active virtual harts and potentially thousands of additional idle (swapped-out) virtual harts, where each virtual hart has direct control of one or more physical devices.

Table 1 summarizes the main limits on the numbers of harts, both physical and virtual, and the numbers of distinct interrupt identities that may be supported with the Advanced Interrupt Architecture.

We assume that any single RISC-V computer (or any single node in a cluster or distributed system) with many thousands of physical harts will probably need an interrupt infrastructure adapted to the machine's specific organization, which we do not attempt to predict.

Table 1. Absolute limits on the numbers of harts and interrupt identities in a system. Individual implementations are likely to have smaller limits.

	Maximum	Requirements
Physical harts	16,384	
Active virtual harts having direct control of a device, per physical hart	31 for RV32, 63 for RV64	RISC-V H extension; IMSICs with guest interrupt files; and an IOMMU
Idle (swapped-out) virtual harts having direct control of a device, per physical hart	potentially thousands	An IOMMU with support for memory-resident interrupt files
Wired interrupts at a single APLIC	1023	
Distinct identities usable for MSIs at each hart (physical or virtual)	2047	IMSICs

1.3. Overview of main components

A RISC-V system's overall architecture for signaling interrupts depends on whether it is built mainly for message-signaled interrupts (MSIs) or for more traditional wired interrupts. In systems with full support for MSIs, every hart has an *Incoming MSI Controller* (IMSIC) that serves as the hart's own private interrupt controller for external interrupts. Conversely, in systems based primarily on traditional wired interrupts, harts do not have IMSICs. Larger systems, and especially those with PCI devices, are expected to fully support MSIs by giving harts IMSICs, whereas many smaller systems may continue to be best served with wired interrupts and simpler harts without IMSICs.

1.3.1. External interrupts without IMSICs

When RISC-V harts do not have Incoming MSI Controllers, external interrupts are signaled to harts through dedicated wires. In that case, an *Advanced Platform-Level Interrupt Controller* (APLIC) acts as a traditional central hub for interrupts, routing and prioritizing external interrupts for each hart as

illustrated in Figure 1. Interrupts may be selectively routed either to machine level or to supervisor level at each hart. The APLIC is specified in Chapter 4.

Without IMSICs, the current Advanced Interrupt Architecture does not support the direct signaling of external interrupts to virtual machines, even when RISC-V harts implement the H extension. Instead, an interrupt must be sent to the relevant hypervisor, which can then choose to inject a virtual interrupt into the virtual machine.

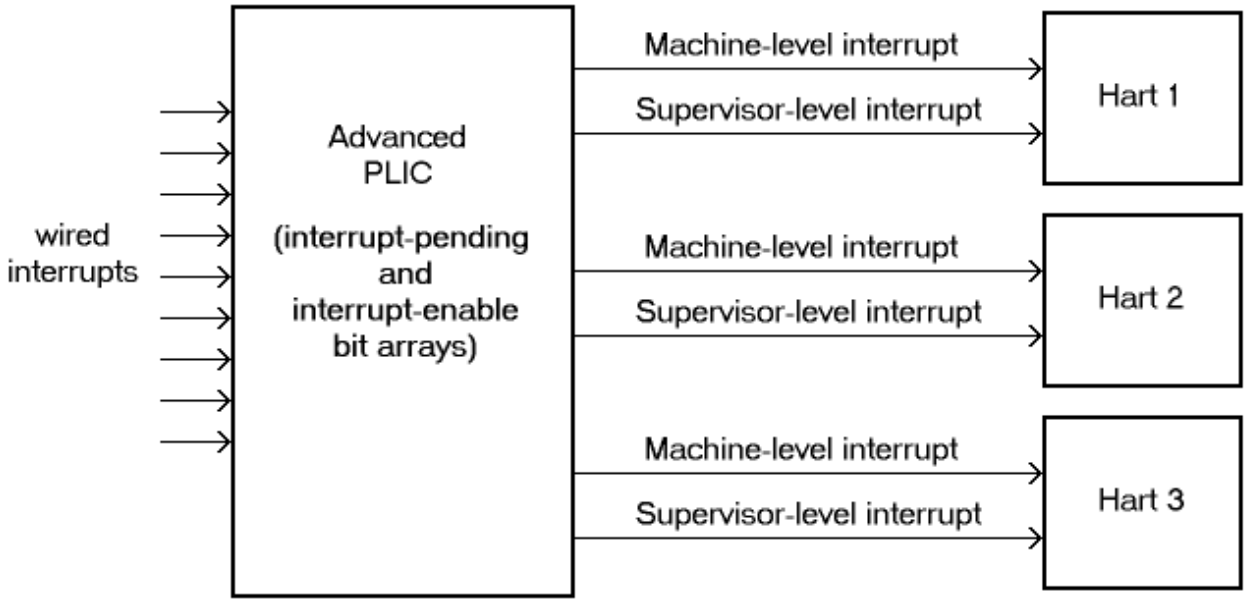

Figure 1. Traditional delivery of wired interrupts to harts without support for MSIs.

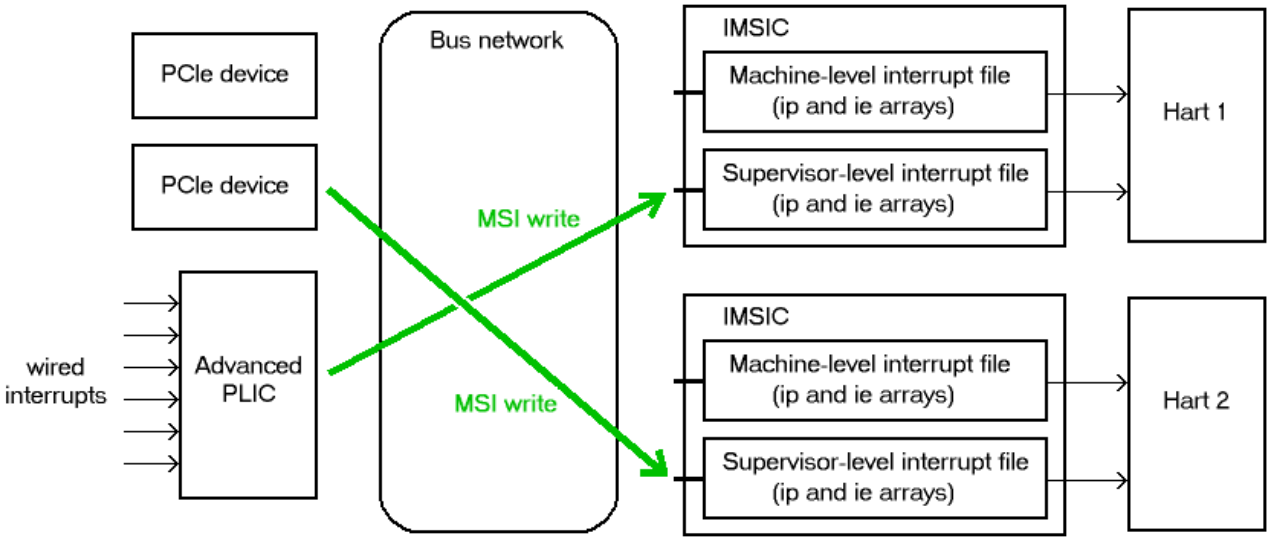

Figure 2. Interrupt delivery by MSIs when harts have IMSICs for receiving them.

1.3.2. External interrupts with IMSICs

To be able to receive message-signaled interrupts (MSIs), each RISC-V hart must have an Incoming MSI Controller (IMSIC) as shown in Figure 2. Fundamentally, a message-signaled interrupt is simply a memory write to a specific address that hardware accepts as indicating an interrupt. To that end, every IMSIC is assigned one or more distinct addresses in the machine's address space, and when a write is

made to one of those addresses in the expected format, the receiving IMSIC interprets the write as an external interrupt for the respective hart.

Because all IMSICs have unique addresses in the machine's physical address space, every IMSIC can receive MSI writes from any agent (hart or device) with permission to write to it. IMSICs have separate addresses for MSIs directed to machine and supervisor levels, in part so the ability to signal interrupts at each privilege level can be separately granted or denied by controlling write permissions at the different addresses, and in part to better support virtualizability (pretending that one privilege level is a higher level). MSIs intended for a hart at a specific privilege level are recorded within the IMSIC in an *interrupt file*, which consists mainly of an array of interrupt-pending bits and a matching array of interrupt-enable bits, the latter indicating which individual interrupts the hart is currently prepared to receive.

IMSIC units are fully defined in Chapter 3. The format of MSIs used by the RISC-V Advanced Interrupt Architecture is described in that chapter, Section 3.2.

When the harts in a RISC-V system have IMSICs, the system will normally still contain an APLIC, but its role is changed. Instead of signaling interrupts to harts directly by wires as in Figure 1, an APLIC converts incoming wired interrupts into MSI writes that are sent to harts via their IMSIC units. Each MSI is sent to a single target hart according to the APLIC's configuration set by software.

If RISC-V harts implement the H extension, IMSICs may have additional *guest interrupt files* for delivering interrupts to virtual machines. Besides Chapter 3 on the IMSIC, see Chapter 6 which specifically covers interrupts to virtual machines. If the system also contains an IOMMU to perform address translation of memory accesses made by I/O devices, then MSIs from those same devices may require special handling. This topic is addressed in Chapter 8, "IOMMU Support for MSIs to Virtual Machines."

1.3.3. Other interrupts

In addition to external interrupts from I/O devices, the RISC-V Privileged Architecture specifies a few other major classes of interrupts for harts. The Privileged Architecture's timer interrupts remain supported in full, and software interrupts remain at least partly supported, although neither appears in Figure 1 and Figure 2. For the specifics on software interrupts, refer to Chapter 7, "Interprocessor Interrupts (IPIs)""

The Advanced Interrupt Architecture adds considerable support for *local interrupts* at a hart, whereby a hart essentially interrupts itself in response to asynchronous events, usually errors. Local interrupts remain contained within a hart (or close to it), so like standard RISC-V timer and software interrupts, they do not pass through an APLIC or IMSIC.

1.4. Interrupt identities at a hart

The RISC-V Privileged Architecture gives every interrupt cause at a hart a distinct *major identity number*, which is the Exception Code automatically written to CSR `mcause` or `scause` on an interrupt trap. Interrupt causes that are standardized by the base Privileged Architecture have major identities in the range 0-15, while numbers 16 and higher are officially available for platform standards or for custom use. The Advanced Interrupt Architecture claims further authority over identity numbers in the ranges 16-23 and 32-47, leaving numbers in the range 24-31 and all major identities 48 and higher still free for custom use. Table 2 characterizes all major interrupt identities with this extension.

Table 2. Major and minor identities for all interrupt causes at a hart. Major identities 0-15 are the purview of the

base Privileged Architecture.

Major identity	Minor identity	
0	-	*Reserved by base Privileged Architecture*
1 2 3	- - -	Supervisor software interrupt Virtual supervisor software interrupt Machine software interrupt
4	-	*Reserved by base Privileged Architecture*
5 6 7	- - -	Supervisor timer interrupt Virtual supervisor timer interrupt Machine timer interrupt
8	-	*Reserved by base Privileged Architecture*
9 10 11	Determined by external interrupt controller	Supervisor external interrupt Virtual supervisor external interrupt Machine external interrupt
12 13 14-15	- - -	Supervisor guest external interrupt Counter overflow interrupt *Reserved by base Privileged Architecture*
16-23	-	*Reserved for standard local interrupts*
24-31	-	*Designated for custom use*
32-34 35 36-42 43 44-47	- - - - -	*Reserved for standard local interrupts* Low-priority RAS event interrupt *Reserved for standard local interrupts* High-priority RAS event interrupt *Reserved for standard local interrupts*
≥48	-	*Designated for custom use*

Interrupts from most I/O devices are conveyed to a hart by the *external interrupt controller* for the hart, which is either the hart's IMSIC (Figure 2) or an APLIC (Figure 1). As Table 2 shows, external interrupts at a given privilege level all share a single major identity number: 11 for machine level, 9 for supervisor level, and 10 for VS-level. External interrupts from different causes are distinguished from one another at a hart by their *minor identity numbers* supplied by the external interrupt controller.

Other interrupt causes besides external interrupts might also have their own minor identities. However, this document has need to discuss minor identities only with regard to external interrupts.

The local interrupts defined by the Advanced Interrupt Architecture and their handling are covered mainly in Chapter 5, "Interrupts for Machine and Supervisor Levels."

1.5. Selection of harts to receive an interrupt

Each signaled interrupt is delivered to only one hart at one privilege level, usually determined by software in one way or another. Unlike some other architectures, the RISC-V Advanced Interrupt Architecture provides no standard hardware mechanism for the broadcast or multicast of interrupts to multiple harts.

For local interrupts, and for any "virtual" interrupts that software injects into less-privileged levels at a

hart, the interrupts are entirely a local affair at the hart and are never visible to other harts. The RISC-V Privileged Architecture's timer interrupts are also uniquely tied to individual harts. For other interrupts, received by a hart from sources outside the hart, each interrupt signal (whether delivered by wire or by an MSI) is configured by software to go to only a single hart.

To send an interprocessor interrupt (IPI) to multiple harts, the originating hart need only execute a loop, sending an individual IPI to each destination hart. For IPIs to a single destination hart, see Chapter 7.

The effort that a source hart expends in sending individual IPIs to multiple destinations will invariably be dwarfed by the combined effort at the receiving harts to handle those interrupts. Hence, providing an automated mechanism for IPI multicast could be expected to reduce a system's total overall work only modestly at best. With a very large number of harts, a hardware mechanism for IPI multicast must contend with the question of how exactly software specifies the intended destination set with each use, and furthermore, the actual physical delivery of IPIs may not differ that much from the software version.

We do not exclude the future possibility of an optional hardware mechanism for multicast IPI, but only if a significant advantage can be demonstrated in real use. As of 2020, Linux has been observed not to make use of multicast IPI hardware even on systems that have it.

In the rare event that a single interrupt from an I/O device needs to be communicated to multiple harts, the interrupt must be sent to a single hart which can then signal the other harts by IPIs.

We contend that the need to communicate an I/O interrupt to multiple harts is sufficiently rare that standardizing hardware support for multicast cannot be justified in this case.

Along with multicast delivery, other architectures support an option for "1-of-N" delivery of interrupts, whereby the hardware chooses a single destination hart from among a configured set of N harts, with the goal of automatic load balancing of interrupt handling among the harts. Experiments in the 2010s called into question the utility of 1-of-N modes in practice, showing that software could often do a better job of load balancing than the hardware algorithms implemented in actual chips. Linux was consequently modified to discontinue using 1-of-N interrupt delivery even on systems that have it.

We remain open to the argument that hardware load balancing of interrupt handling may be beneficial for certain specialized markets, such as networking. However, the claims made so far in this regard do not justify requiring support for 1-of-N delivery in all RISC-V servers. With more evidence, some mechanism for 1-of-N delivery might become a future option.

The original Platform-Level Interrupt Controller (PLIC) for RISC-V is configurable so each interrupt source signals external interrupts to any subset of the harts, potentially all harts. When multiple harts receive an external interrupt from a single cause at the PLIC, the first hart to claim the interrupt at the PLIC is the one responsible for servicing it. Usually this sets up a race, where the subset of harts configured to receive the multicast interrupt all take an external interrupt trap simultaneously and compete to be the first to claim the interrupt at the PLIC. The intention is to provide a form of 1-of-N interrupt delivery. However, for all the harts that fail to win the claim, the interrupt trap becomes wasted effort.

For the reasons already given, the Advanced PLIC supports sending each signaled interrupt to only a single hart chosen by software, not to multiple harts.

1.6. ISA extensions Smaia and Ssaia

The Advanced Interrupt Architecture (AIA) defines two names for extensions to the RISC-V instruction set architecture (ISA), one for machine-level execution environments, and another for supervisor-level environments. For a machine-level environment, extension **Smaia** encompasses all added CSRs and all modifications to interrupt response behavior that the AIA specifies for a hart, over all privilege levels. For a supervisor-level environment, extension **Ssaia** is essentially the same as Smaia except excluding the machine-level CSRs and behavior not directly visible to supervisor level.

Extensions Smaia and Ssaia cover only those AIA features that impact the ISA at a hart. Although the following are described or discussed in this document as part of the AIA, they are not implied by Smaia or Ssaia because the components are categorized as non-ISA: APLICs, IOMMUs, and any mechanisms for initiating interprocessor interrupts apart from writing to IMSICs.

As revealed in subsequent chapters, the exact set of CSRs and behavior added by the AIA, and hence implied by Smaia or Ssaia, depends on the base ISA's XLEN (RV32 or RV64), on whether S-mode and the H extension are implemented, and on whether the hart has an IMSIC. But individual AIA extension names are not provided for each possible valid subset. Rather, the different combinations are inferable from the intersection of features indicated (such as RV64I + S-mode + Smaia, but without the H extension).

Software development tools like compilers and assemblers need not be concerned about whether an IMSIC exists but should just allow attempts to access the IMSIC CSRs (described in Chapter 2 and Chapter 3) if Smaia or Ssaia is indicated. Without an actual IMSIC, such attempts may trap, but that is not a problem for the development tools.

If extension Smaia/Ssaia is implemented, then anywhere that the AIA specification has an irreconcilable conflict with the requirements of another implemented RISC-V extension, the AIA is intended to have priority, unless the other extension explicitly extends or overrides the AIA.

Extension Smcsrind/Sscsrind explicitly extends the AIA's facility for indirect CSR access provided by the `*iselect` *and* `*ireg` *CSRs described in the next chapter. Hence, if Smcsrind/Sscsrind is also implemented, any perceived conflicts between it and the AIA should be resolved in favor of Smcsrind/Sscsrind.*

Chapter 2. Control and Status Registers (CSRs) Added to Harts

For each privilege level at which a RISC-V hart can take interrupt traps, the Advanced Interrupt Architecture adds CSRs for interrupt control and handling.

2.1. Machine-level CSRs

Table 3 lists both the CSRs added for machine level and existing machine-level CSRs whose size is changed by the Advanced Interrupt Architecture. Existing CSRs **mie**, **mip**, and **mideleg** are widened to 64 bits to support a total of 64 interrupt causes.

For RV32, the *high-half* CSRs listed in the table allow access to the upper 32 bits of registers **mideleg**, **mie**, **mvien**, **mvip**, and **mip**. The Advanced Interrupt Architecture requires that these high-half CSRs exist for RV32, but the bits they access may all be merely read-only zeros.

CSRs **miselect** and **mireg** provide a window for accessing multiple registers beyond the CSRs in Table 3. The value of **miselect** determines which register is currently accessible through alias CSR **mireg**. **miselect** is a **WARL** register, and it must support a minimum range of values depending on the implemented features. When an IMSIC is not implemented, **miselect** must be able to hold at least any 6-bit value in the range 0 to 0**x3F**. When an IMSIC is implemented, **miselect** must be able to hold any 8-bit value in the range 0 to 0**xFF**. The Advanced Interrupt Architecture makes use of these subranges of values for **miselect**:

0x30-0x3F major interrupt priorities

0x70-0xFF external interrupts (only with an IMSIC)

Table 3. Machine-level CSRs added or widened by the Advanced Interrupt Architecture.

Number	Privilege	Width	Name	Description
Machine-Level Window to Indirectly Accessed Registers				
0x350	MRW	XLEN	**miselect**	Machine indirect register select
0x351	MRW	XLEN	**mireg**	Machine indirect register alias
Machine-Level Interrupts				
0x304	MRW	64	**mie**	Machine interrupt-enable bits
0x344	MRW	64	**mip**	Machine interrupt-pending bits
0x35C	MRW	MXLEN	**mtopei**	Machine top external interrupt (only with an IMSIC)
0xFB0	MRO	MXLEN	**mtopi**	Machine top interrupt
Delegated and Virtual Interrupts for Supervisor Level				
0x303	MRW	64	**mideleg**	Machine interrupt delegation
0x308	MRW	64	**mvien**	Machine virtual interrupt enables
0x309	MRW	64	**mvip**	Machine virtual interrupt-pending bits
Machine-Level High-Half CSRs (RV32 only)				

Number	Privilege	Width	Name	Description
0x313	MRW	32	`midelegh`	Upper 32 bits of `mideleg` (only with S-mode)
0x314	MRW	32	`mieh`	Upper 32 bits of `mie`
0x318	MRW	32	`mvienh`	Upper 32 bits of `mvien` (only with S-mode)
0x319	MRW	32	`mviph`	Upper 32 bits of `mvip` (only with S-mode)
0x354	MRW	32	`miph`	Upper 32 bits of `mip`

Values of `miselect` with the most-significant bit set (bit `XLEN - 1` = `1`) are designated for custom use, presumably for accessing custom registers through `mireg`. If `XLEN` changes, the most-significant bit of `miselect` moves to the new position, retaining its value from before. An implementation is not required to support any custom values for `miselect`.

Other `miselect` values are reserved for other RISC-V extensions.

RISC-V extension Smcsrind generalizes the mechanism of indirect register access provided by `miselect` *and* `mireg`.

Normally, the range for external interrupts, 0x70-0xFF, is populated only when an IMSIC is implemented; else, attempts to access `mireg` when `miselect` is in this range cause an illegal instruction exception. The contents of the external-interrupts region are documented in Chapter 3 on the IMSIC.

CSR `mtopei` also exists only when an IMSIC is implemented, so is documented in Chapter 3 along with the indirectly accessed IMSIC registers.

CSR `mtopi` reports the highest-priority interrupt that is pending and enabled for machine level, as specified in Section 5.2.2.

When S-mode is implemented, CSRs `mvien` and `mvip` support interrupt filtering and virtual interrupts for supervisor level. These facilities are explained in Section 5.3.

If extension Smcsrind is also implemented, then when `miselect` has a value in the range 0x30-0x3F or 0x70-0xFF, attempts to access alias CSRs `mireg2` through `mireg6` raise an illegal instruction exception.

2.2. Supervisor-level CSRs

Table 4 lists the supervisor-level CSRs that are added and existing CSRs that are widened to 64 bits, if the hart implements S-mode. The functions of these registers all match their machine-level counterparts.

Table 4. Supervisor-level CSRs added or widened by the Advanced Interrupt Architecture.

Number	Privilege	Width	Name	Description
Supervisor-Level Window to Indirectly Accessed Registers				
0x150	SRW	XLEN	`siselect`	Supervisor indirect register select
0x151	SRW	XLEN	`sireg`	Supervisor indirect register alias
Supervisor-Level Interrupts				

Number	Privilege	Width	Name	Description
0x104	SRW	64	`sie`	Supervisor interrupt-enable bits
0x144	SRW	64	`sip`	Supervisor interrupt-pending bits
0x15C	SRW	SXLEN	`stopei`	Supervisor top external interrupt (only with an IMSIC)
0xDB0	SRO	SXLEN	`stopi`	Supervisor top interrupt
Supervisor-Level High-Half CSRs (RV32 only)				
0x114	SRW	32	`sieh`	Upper 32 bits of `sie`
0x154	SRW	32	`siph`	Upper 32 bits of `sip`

The space of registers accessible through the `siselect`/`sireg` window is separate from but parallels that of machine level, being for supervisor-level interrupts instead of machine-level interrupts. The subranges of values used for `siselect` are once again these:

`0x30-0x3F` major interrupt priorities

`0x70-0xFF` external interrupts (only with an IMSIC)

For maximum compatibility, it is recommended that `siselect` support at least a 9-bit range, `0` to `0x1FF`, regardless of whether an IMSIC exists.

Because the VS CSR `vsiselect` *(Section 2.3) always has at least 9 bits, and like other VS CSRs,* `vsiselect` *substitutes for* `siselect` *when executing in a virtual machine (VS-mode or VU-mode), implementing a smaller range for* `siselect` *allows software to discover it is not running in a virtual machine.*

Like `miselect`, values of `siselect` with the most-significant bit set (bit XLEN - 1 = 1) are designated for custom use. If XLEN changes, the most-significant bit of `siselect` moves to the new position, retaining its value from before. An implementation is not required to support any custom values for `siselect`.

Other `siselect` values are reserved for other RISC-V extensions.

At supervisor level, extension Sscsrind generalizes the mechanism of indirect register access provided by `siselect` *and* `sireg`*, as well as the parallel at VS-level provided by* `vsiselect` *and* `vsireg`*, described in the next subsection.*

Note that the widths of 'siselect' and 'sireg' are always the current XLEN rather than SXLEN. Hence, for example, if MXLEN = 64 and SXLEN = 32, then these registers are 64 bits when the current privilege mode is M (running RV64 code) but 32 bits when the privilege mode is S (RV32 code).

CSR `stopei` is described with the IMSIC in Chapter 3.

Register `stopi` reports the highest-priority interrupt that is pending and enabled for supervisor level, as specified in Section 5.4.2.

If extension Sscsrind is also implemented, then when `siselect` has a value in the range `0x30-0x3F` or `0x70-0xFF`, attempts to access alias CSRs `sireg2` through `sireg6` raise an illegal instruction exception (unless executing in a virtual machine, covered in the next section).

2.3. Hypervisor and VS CSRs

If a hart implements the H extension, then the hypervisor and VS CSRs listed in Table 5 are also either added or widened to 64 bits.

The new hypervisor CSRs in the table (`hvien`, `hvictl` , `hviprio1`, and `hviprio2`) augment `hvip` for injecting interrupts into VS level. The use of these registers is covered in Chapter 6 on interrupts for virtual machines.

The new VS CSRs (`vsiselect`, `vsireg`, `vstopei`, and `vstopi`) all match supervisor CSRs, and substitute for those supervisor CSRs when executing in a virtual machine (in VS-mode or VU-mode).

CSR `vsiselect` is required to support at least a 9-bit range of `0` to `0x1FF`, whether or not an IMSIC is implemented. As with `siselect`, values of `vsiselect` with the most-significant bit set (bit XLEN - 1 = 1) are designated for custom use. If XLEN changes, the most-significant bit of `vsiselect` moves to the new position, retaining its value from before.

Like `siselect` and `sireg`, the widths of `vsiselect` and `vsireg` are always the current XLEN rather than VSXLEN. Hence, for example, if HSXLEN = 64 and VSXLEN = 32, then these registers are 64 bits when accessed by a hypervisor in HS-mode (running RV64 code) but 32 bits for a guest OS in VS-mode (RV32 code).

Table 5. Hypervisor and VS CSRs added or widened by the Advanced Interrupt Architecture. (Parameter HSXLEN is just another name for SXLEN for hypervisor-extended S-mode).

Number	Privilege	Width	Name	Description
Delegated and Virtual Interrupts, Interrupt Priorities, for VS Level				
0x603	HRW	64	`hideleg`	Hypervisor interrupt delegation
0x608	HRW	64	`hvien`	Hypervisor virtual interrupt enables
0x609	HRW	HSXLEN	`hvictl`	Hypervisor virtual interrupt control
0x645	HRW	64	`hvip`	Hypervisor virtual interrupt-pending bits
0x646	HRW	64	`hviprio1`	Hypervisor VS-level interrupt priorities
0x647	HRW	64	`hviprio2`	Hypervisor VS-level interrupt priorities
VS-Level Window to Indirectly Accessed Registers				
0x250	HRW	XLEN	`vsiselect`	Virtual supervisor indirect register select
0x251	HRW	XLEN	`vsireg`	Virtual supervisor indirect register alias
VS-Level Interrupts				
0x204	HRW	64	`vsie`	Virtual supervisor interrupt-enable bits
0x244	HRW	64	`vsip`	Virtual supervisor interrupt-pending bits
0x25C	HRW	VSXLEN	`vstopei`	Virtual supervisor top external interrupt (only with an IMSIC)
0xEB0	HRO	VSXLEN	`vstopi`	Virtual supervisor top interrupt
Hypervisor and VS-Level High-Half CSRs (RV32 only)				

Number	Privilege	Width	Name	Description
0x613	HRW	32	`hidelegh`	Upper 32 bits of `hideleg`
0x618	HRW	32	`hvienh`	Upper 32 bits of `hvien`
0x655	HRW	32	`hviph`	Upper 32 bits of `hvip`
0x656	HRW	32	`hviprio1h`	Upper 32 bits of `hviprio1`
0x657	HRW	32	`hviprio2h`	Upper 32 bits of `hviprio2`
0x214	HRW	32	`vsieh`	Upper 32 bits of `vsie`
0x254	HRW	32	`vsiph`	Upper 32 bits of `vsip`

The space of registers selectable by `vsiselect` is more limited than for machine and supervisor levels:

`0x030-0x03F` inaccessible

`0x070-0x0FF` external interrupts (IMSIC only), or inaccessible

Other `vsiselect` values are reserved for other RISC-V extensions.

For alias CSRs `sireg` and `vsireg`, the H extension's usual rules for when to raise a virtual instruction exception (based on whether an instruction is *HS-qualified*) are not applicable. The rules given in this section for `sireg` and `vsireg` apply instead, unless overridden by the requirements of Section 2.5, which take precedence over this section when extension Smstateen is also implemented.

A virtual instruction exception is raised for attempts from VS-mode or VU-mode to directly access `vsireg`, or attempts from VU-mode to access `sireg`.

When `vsiselect` has the number of an *inaccessible* register, attempts from M-mode or HS-mode to access `vsireg` raise an illegal instruction exception, and attempts from VS-mode to access `sireg` (really `vsireg`) raise a virtual instruction exception.

Requiring a range of 0-0x1FF for `vsiselect`*, even though most or all of the space is reserved or inaccessible, permits a hypervisor to emulate indirectly accessed registers in the implemented range, including registers that are not currently defined but may be standardized in the future.*

The indirectly accessed registers for external interrupts (numbers 0x70-0xFF) are accessible only when field VGEIN of `hstatus` is the number of an implemented guest external interrupt, not zero. If VGEIN is not the number of an implemented guest external interrupt (including the case when no IMSIC is implemented), then all indirect register numbers in the ranges 0x030-0x03F and 0x070-0x0FF designate an inaccessible register at VS level.

Along the same lines, when `hstatus.VGEIN` is not the number of an implemented guest external interrupt, attempts from M-mode or HS-mode to access CSR `vstopei` raise an illegal instruction exception, and attempts from VS-mode to access `stopei` raise a virtual instruction exception.

If extension Sscsrind is also implemented, then when `vsiselect` has a value in the range 0x30-0x3F or 0x70-0xFF, attempts from M-mode or HS-mode to access alias CSRs `vsireg2` through `vsireg6` raise an illegal instruction exception, and attempts from VS-mode to access `sireg2` through `sireg6` raise a virtual instruction exception.

2.4. Virtual instruction exceptions

Following the default rules for the H extension, attempts from VS-mode to directly access a hypervisor

or VS CSR other than `vsireg`, or from VU-mode to access any supervisor-level CSR (including hypervisor and VS CSRs) other than `sireg` or `vsireg`, usually raise not an illegal instruction exception but instead a virtual instruction exception. For details, see the H extension documentation.

Instructions that read/write CSR `stopei` or `vstopei` are considered to be *HS-qualified* unless all of following are true: the hart has an IMSIC, extension Smstateen is implemented, and bit 58 of `mstateen0` is zero. (See the next section, Section 2.5, about `mstateen0`.)

For `sireg` and `vsireg`, see both the previous section, Section 2.3, and the next, Section 2.5, for when a virtual instruction exception is required instead of an illegal instruction exception.

2.5. Access control by the state-enable CSRs

If extension Smstateen is implemented together with the Advanced Interrupt Architecture (AIA), three bits of state-enable register `mstateen0` control access to AIA-added state from privilege modes less privileged than M-mode:

bit 60 CSRIND: CSRs `siselect`, `sireg`, `vsiselect`, and `vsireg`

bit 59 AIA: all other state added by the AIA and not controlled by bits CSRIND and IMSIC

bit 58 IMSIC: all IMSIC state, including CSRs `stopei` and `vstopei`

If one of these bits is zero in `mstateen0`, an attempt to access the corresponding state from a privilege mode less privileged than M-mode results in an illegal instruction trap. As always, the state-enable CSRs do not affect the accessibility of any state when in M-mode, only in less privileged modes. For more explanation, see the documentation for extension Smstateen.

The AIA bit controls access to AIA CSRs `siph`, `sieh`, `stopi`, `hidelegh`, `hvien`/`hvienh`, `hviph`, `hvictl`, `hviprio1`/`hviprio1h`, `hviprio2`/`hviprio2h`, `vsiph`, `vsieh`, and `vstopi`, as well as to the supervisor-level interrupt priorities accessed through `siselect` + `sireg` (the `iprio` array of Section 5.4.1).

The IMSIC bit is implemented in `mstateen0` only if the hart has an IMSIC. If the H extension is also implemented, this bit does not affect the behavior or accessibility of hypervisor CSRs `hgeip` and `hgeie`, or field VGEIN of `hstatus`. In particular, guest external interrupts from an IMSIC continue to be visible to HS-mode in `hgeip` even when `mstateen0`.IMSIC is zero.

An earlier, pre-ratification draft of Smstateen said that when `mstateen0`*.IMSIC is zero, registers* `hgeip` *and* `hgeie` *and field VGEIN of* `hstatus` *are all read-only zeros. That effect is no longer correct.*

If the hart does not have an IMSIC, the IMSIC bit of `mstateen0` is read-only zero, but Smstateen has no effect on attempts to access the nonexistent IMSIC state.

This means in particular that, when the hart does not have an IMSIC, the following raise a virtual instruction exception as described in Table 5, not an illegal instruction exception, despite that `mstateen0`*.IMSIC is zero:*

- *attempts from VS-mode to access* `sireg` *(really* `vsireg`*) while* `vsiselect` *has a value in the range 0x70–0xFF; and*
- *attempts from VS-mode to access* `stopei` *(really* `vstopei`*).*

If the CSRIND bit of `mstateen0` is one, then regardless of any other `mstateen` bits (including the AIA

and IMSIC bits of **mstateen0**), a virtual instruction exception is raised as described in Section 2.3 for all attempts from VS-mode or VU-mode to directly access **vsireg**, and for all attempts from VU-mode to access **sireg**. This behavior is overridden only when **mstateen0**.CSRIND is zero.

If the H extension is implemented, the same three bits are defined also in hypervisor CSR **hstateen0** but concern only the state potentially accessible to a virtual machine executing in privilege modes VS and VU:

bit 60 CSRIND: CSRs **siselect** and **sireg** (really **vsiselect** and **vsireg**)

bit 59 AIA: CSRs **siph** and **sieh** (RV32 only) and **stopi** (really **vsiph**, **vsieh**, and **vstopi**)

bit 58 IMSIC: all state of IMSIC guest interrupt files, including CSR **stopei**(really **vstopei**)

If one of these bits is zero in **hstateen0**, and the same bit is one in **mstateen0**, then an attempt to access the corresponding state from VS or VU-mode raises a virtual instruction exception. (But note that, for high-half CSRs **siph** and **sieh**, this applies only when XLEN = 32. When XLEN > 32, an attempt to access **siph** or **sieh** raises an illegal instruction exception as usual, not a virtual instruction exception.)

If the CSRIND bit is one in **mstateen0** but is zero in **hstateen0**, then all attempts from VS or VU-mode to access **siselect** or **sireg** raise a virtual instruction exception, not an illegal instruction exception, regardless of the value of **vsiselect** or any other **mstateen** bits.

The IMSIC bit is implemented in **hstateen0** only if the hart has an IMSIC. Furthermore, even with an IMSIC, **hstateen0**.IMSIC may (or may not) be read-only zero if the IMSIC has no *guest interrupt files* for guest external interrupts (Chapter 3). When this bit is zero (whether read-only zero or set to zero), a virtual machine is prevented from accessing the hart's IMSIC the same as when **hstatus.VGEIN** = 0.

Extension Ssstateen is defined as the supervisor-level view of Smstateen. Therefore, the combination of Ssaia and Ssstateen incorporates the bits defined above for **hstateen0** but not those for **mstateen0**, since machine-level CSRs are not visible to supervisor level.

Chapter 3. Incoming MSI Controller (IMSIC)

An Incoming MSI Controller (IMSIC) is an optional RISC-V hardware component that is closely coupled with a hart, one IMSIC per hart. An IMSIC receives and records incoming message-signaled interrupts (MSIs) for a hart, and signals to the hart when there are pending and enabled interrupts to be serviced.

An IMSIC has one or more memory-mapped registers in the machine's address space for receiving MSIs. Aside from those memory-mapped registers, software interacts with an IMSIC primarily through several RISC-V CSRs at the attached hart.

3.1. Interrupt files and interrupt identities

In a RISC-V system, MSIs are directed not just to a specific hart but to a specific privilege level of a specific hart, such as machine or supervisor level. Furthermore, when a hart implements the H extension, an IMSIC may optionally allow MSIs to be directed to a specific virtual hart at virtual supervisor level (VS level).

For each privilege level and each virtual hart to which MSIs may be directed at a hart, the hart's IMSIC contains a separate *interrupt file*. Assuming a hart implements supervisor mode, its IMSIC has at least two interrupt files, one for machine level and the other for supervisor level. When a hart also implements the H extension, its IMSIC may have additional interrupt files for virtual harts, called *guest interrupt files*. The number of guest interrupt files an IMSIC has for virtual harts is exactly *GEILEN*, the number of supported guest external interrupts, as defined by the H extension.

Each individual interrupt file consists mainly of two arrays of bits of the same size, one array for recording MSIs that have arrived but are not yet serviced (interrupt-pending bits), and the other array for specifying which interrupts the hart will currently accept (interrupt-enable bits). Each bit position in the two arrays corresponds with a different interrupt *identity number* by which MSIs from different sources are distinguished at an interrupt file. Because an IMSIC is the external interrupt controller for a hart, an interrupt file's interrupt identities become the *minor identities* for external interrupts at the attached hart.

The number of interrupt identities supported by an interrupt file (and hence the number of active bits in each array) is one less than a multiple of 64, and may be a minimum of 63 and a maximum of 2047.

Platform standards may increase the minimum number of interrupt identities that must be implemented by each interrupt file.

When an interrupt file supports N distinct interrupt identities, valid identity numbers are between 1 and N inclusive. The identity numbers within this range are said to be implemented by the interrupt file; numbers outside this range are not implemented. The number zero is never a valid interrupt identity.

IMSIC hardware does not assume any connection between the interrupt identity numbers at one interrupt file and those at another interrupt file. Software is commonly expected to assign the same interrupt identity number to different MSI sources at different interrupt files, without coordination across interrupt files. Thus the total number of MSI sources that can be separately distinguished within a system is potentially the product of the number of interrupt identities at a single interrupt

file times the total number of interrupt files in the system, over all harts.

It is not necessarily the case that all interrupt files in a system are the same size (implement the same number of interrupt identities). For a given hart, the interrupt files for guest external interrupts must all be the same size, but the interrupt files at machine level and at supervisor level may differ in size from those of guest external interrupts, and from each other. Likewise, the interrupt files of different harts may be different sizes.

A platform might provide a means for software to configure the number of interrupt files in an IMSIC and/or their sizes, such as by allowing a smaller interrupt file at machine level to be traded for a larger one at supervisor level, or vice versa, for example. Any such configurability is outside the scope of this specification. It is recommended, however, that only machine level be given the power to change the number and sizes of interrupt files in an IMSIC.

3.2. MSI encoding

Established standards (in particular, for PCI and PCI Express) dictate that an individual message-signaled interrupt (MSI) from a device takes the form of a naturally aligned 32-bit write by the device, with the address and value both configured at the device (or device controller) by software. Depending on the versions of the standards to which a device or controller conforms, the address might be restricted to the lower 4-GiB (32-bit) range, and the value written might be limited to a 16-bit range, with the upper 16 bits always being zeros.

When RISC-V harts have IMSICs, an MSI from a device is normally sent directly to an individual hart that was selected by software to handle the interrupt (presumably based on some interrupt affinity policy). An MSI is directed to a specific privilege level, or to a specific virtual hart, via the corresponding interrupt file that exists in the receiving hart's IMSIC. The MSI write address is the physical address of a particular word-size register that is physically connected to the target interrupt file. The MSI write data is simply the identity number of the interrupt to be made pending in that interrupt file (becoming eventually the minor identity for an external interrupt to the attached hart).

By configuring an MSI's address and data at a device, system software fully controls: (a) which hart receives a particular device interrupt, (b) the target privilege level or virtual hart, and (c) the identity number that represents the MSI in the target interrupt file. Elements a and b are determined by which interrupt file is targeted by the MSI address, while element c is communicated by the MSI data.

As the maximum interrupt identity number an IMSIC can support is 2047, a 16-bit limit on MSI data values presents no problem.

When the H extension is implemented and a device is being managed directly by a guest operating system, MSI addresses from the device are initially guest physical addresses, as they are configured at the device by the guest OS. These guest addresses must be translated by an IOMMU, which gets configured by the hypervisor to redirect those MSIs to the interrupt files for the correct guest external interrupts. For more on this topic, see Chapter 8.

3.3. Interrupt priorities

Within a single interrupt file, interrupt priorities are determined directly from interrupt identity numbers. Lower identity numbers have higher priority.

Because MSIs give software complete control over the assignment of identity numbers in an interrupt file, software is free to select identity numbers that reflect the relative

priorities desired for interrupts.

It is true that software could adjust interrupt priorities more dynamically if interrupt files included an array of priority numbers to assign to each interrupt identity. However, we believe that such additional flexibility would not be utilized often enough to justify the extra hardware expense. In fact, for many systems currently employing MSIs, it is common practice for software to ignore interrupt priorities entirely and act as though all interrupts had equal priority.

An interrupt file's lowest identity numbers have been given the highest priorities, not the reverse order, because it is only for the highest-priority interrupts that priority order may need to be carefully managed, yet it is the low-numbered identities, 1 through 63 (or perhaps 1 through 127), that are guaranteed to exist across all systems. Consider, for example, that an interrupt file's highest-priority interrupt—presumably the most time-critical—is always identity number 1. If priority order were reversed, the highest-priority interrupt would have different identity numbers on different machines, depending on how many identities are implemented by interrupt files. The ability for software to assign fixed identity numbers to the highest-priority interrupts is considered worth any discomfort that may be felt from interrupt priorities being the reverse of the natural number order.

3.4. Reset and revealed state

Upon reset of an IMSIC, all the state of its interrupt files becomes valid and consistent but otherwise UNSPECIFIED, except possibly for the `eidelivery` register of machine-level and supervisor-level interrupt files, as specified in Section 3.8.1.

If an IMSIC contains a supervisor-level interrupt file and software at the attached hart enables S-mode that was previously disabled (e.g. by changing bit S of CSR `misa` from zero to one), all state of the supervisor-level interrupt file is valid and consistent but otherwise UNSPECIFIED. Likewise, if an IMSIC contains guest interrupt files and software at the attached hart enables the H extension that was previously disabled (e.g. by changing bit H of `misa` from zero to one), all state of the IMSIC's guest interrupt files is valid and consistent but otherwise UNSPECIFIED.

3.5. Memory region for an interrupt file

Each interrupt file in an IMSIC has one or two memory-mapped 32-bit registers for receiving MSI writes. These memory-mapped registers are located within a naturally aligned 4-KiB region (a page) of physical address space that exists for the interrupt file, i.e., one page per interrupt file.

The layout of an interrupt-file's memory region is:

offset	size	register name
0x000	4 bytes	`seteipnum_le`
0x004	4 bytes	`seteipnum_be`

All other bytes in an interrupt file's 4-KiB memory region are reserved and must be implemented as read-only zeros.

Only naturally aligned 32-bit simple reads and writes are supported within an interrupt file's memory region. Writes to read-only bytes are ignored. For other forms of accesses (other sizes, misaligned

accesses, or AMOs), an IMSIC implementation should preferably report an access fault or bus error but must otherwise ignore the access.

If i is an implemented interrupt identity number, writing value i in little-endian byte order to `seteipnum_le` (Set External Interrupt-Pending bit by Number, Little-Endian) causes the pending bit for interrupt i to be set to one. A write to `seteipnum_le` is ignored if the value written is not an implemented interrupt identity number in little-endian byte order.

For systems that support big-endian byte order, if i is an implemented interrupt identity number, writing value i in big-endian byte order to `seteipnum_be` (Set External Interrupt-Pending bit by Number, Big-Endian) causes the pending bit for interrupt i to be set to one. A write to `seteipnum_be` is ignored if the value written is not an implemented interrupt identity number in big-endian byte order. Systems that support only little-endian byte order may choose to ignore all writes to `seteipnum_be`.

In most systems, `seteipnum_le` is the write port for MSIs directed to this interrupt file. For systems built mainly for big-endian byte order, `seteipnum_be` may serve as the write port for MSIs directed to this interrupt file from some devices.

A read of `seteipnum_le` or `seteipnum_be` returns zero in all cases.

When not ignored, writes to an interrupt file's memory region are guaranteed to be reflected in the interrupt file eventually, but not necessarily immediately. For a single interrupt file, the effects of multiple writes (stores) to its memory region, though arbitrarily delayed, always occur in the same order as the *global memory order* of the stores as defined by the RISC-V Unprivileged ISA.

In most circumstances, any delay between the completion of a write to an interrupt file's memory region and the effect of the write on the interrupt file is indistinguishable from other delays in the memory system. However, if a hart writes to a `seteipnum_le` *or* `seteipnum_be` *register of its own IMSIC, then a delay between the completion of the store instruction and the consequent setting of an interrupt-pending bit in the interrupt file may be visible to the hart.*

3.6. Arrangement of the memory regions of multiple interrupt files

Each interrupt file that an IMSIC implements has its own memory region as described in the previous section, occupying exactly one 4-KiB page of machine address space. When practical, the memory pages of the machine-level interrupt files of all IMSICs should be located together in one part of the address space, and the memory pages of all supervisor-level and guest interrupt files should similarly be located together in another part of the address space, according to the rules below.

The main reason for separating the machine-level interrupt files from the other interrupt files in the address space is so harts that implement physical memory protection (PMP) can grant supervisor-level access to all supervisor-level and guest interrupt files using only a single PMP table entry. If the memory pages for machine-level interrupt files are instead interleaved with those of lower-privilege interrupt files, the number of PMP table entries needed for granting supervisor-level access to all non-machine-level interrupt files could equal the number of harts in the system.

If a machine's construction dictates that harts be subdivided into groups, with each group relegated to its own portion of the address space, then the best that can be achieved is to locate together the machine-level interrupt files of each group of harts separately, and likewise locate together the

supervisor-level and guest interrupt files of each group of harts separately. This situation is further addressed later below.

A system may divide harts into groups in the address space because each group exists on a separate chip (or chiplet in a multi-chip module), and weaving together the address spaces of the multiple chips is impractical. In that case, granting supervisor-level access to all non-machine-level interrupt files takes one PMP table entry per group.

For the purpose of locating the memory pages of interrupt files in the address space, assume each hart (or each hart within a group) has a unique hart number that may or may not be related to the unique hart identifiers ("hart IDs") that the Privileged Architecture assigns to harts. For convenient addressing, the memory pages of all machine-level interrupt files (or all those of a single group of harts) should be arranged so that the address of the machine-level interrupt file for hart number h is given by the formula $A + h \times 2^C$ for some integer constants A and C. If the largest hart number is h_{max}, let $k = \lceil \log_2(h_{max} + 1) \rceil$, the number of bits needed to represent any hart number. Then the base address A should be aligned to a 2^{k+C} address boundary, so $A + h \times 2^C$ always equals $A \mid (h \times 2^C)$, where the vertical bar (|) represents bitwise logical OR.

The smallest that C can be is 12, with 2^C being the size of one 4-KiB page. If C>12, the start of the memory page for each machine-level interrupt file is aligned not just to a 4-KiB page but to a stricter 2^C address boundary. Within the 2^{k+C}-size address range A through $A + 2^{k+C} - 1$, every 4-KiB page that is not occupied by a machine-level interrupt file should be filled with 32-bit words of read-only zeros, such that any read of an aligned word returns zero and any write to an aligned word is ignored.

The memory pages of all supervisor-level interrupt files (or all those of a single group of harts) should similarly be arranged so that the address of the supervisor-level interrupt file for hart number h is $B + h \times 2^D$ for some integer constants B and D, with the base address B being aligned to a 2^{k+D} address boundary.

If an IMSIC implements guest interrupt files, the memory pages for the IMSIC's supervisor-level interrupt file and for its guest interrupt files should be contiguous, starting with the supervisor-level interrupt file at the lowest address and followed by the guest interrupt files, ordered by guest interrupt number. Schematically, the memory pages should be ordered contiguously as

S, G_1, G_2, G_3, ...

where S is the page for the supervisor-level interrupt file and each G_i is the page for the interrupt file of guest interrupt number i. Consequently, the smallest that constant D can be is $\lceil \log_2(maximumGEILEN + 1) \rceil + 12$, recalling that GEILEN for each IMSIC is the number of guest interrupt files the IMSIC implements.

Within the 2^{k+D}-size address range B through $B + 2^{k+D} - 1$, every 4-KiB page that is not occupied by an interrupt file (supervisor-level or guest) should be filled with 32-bit words of read-only zeros.

When a system divides harts into groups, each in its own separate portion of the address space, the memory page addresses of interrupt files should follow the formulas $g \times 2^E + A + h \times 2^C$ for machine-level interrupt files, and $g \times 2^E + B + h \times 2^D$ for supervisor-level interrupt files, with g being a *group number*, h being a hart number relative to the group, and E being another integer constant $\geq k + \max(C, D)$ but usually much larger. If the largest group number is g_{max}, let $j = \lceil \log_2(g_{max} + 1) \rceil$, the number of bits needed to represent any group number. Besides being multiples of 2^{k+C} and 2^{k+D} respectively, A and B should be chosen so

$((2^j - 1) \times 2^E)$ & $A = 0$ and $((2^j - 1) \times 2^E)$ & $B = 0$

where an ampersand (&) represents bitwise logical AND. This ensures that

$g \times 2^E + A + h \times 2^C$ always equals $(g \times 2^E) \mid A \mid (h \times 2^C)$, and
$g \times 2^E + B + h \times 2^D$ always equals $(g \times 2^E) \mid B \mid (h \times 2^D)$.

Infilling with read-only-zero pages is expected only within each group, not between separate groups. Specifically, if g is any integer between 0 and $2^j - 1$ inclusive, then within the address ranges,

$g \times 2^E + A$ through $g \times 2^E + A + 2^{k+C} - 1$, and
$g \times 2^E + B$ through $g \times 2^E + B + 2^{k+D} - 1$,

pages not occupied by an interrupt file should be read-only zeros.

See also Section 4.9.1 for the default algorithms an Advanced PLIC may use to determine the destination addresses of outgoing MSIs, which should be the addresses of IMSIC interrupt files.

3.7. CSRs for external interrupts via an IMSIC

Software accesses a hart's IMSIC primarily through the CSRs introduced in Chapter 2. There is a separate set of CSRs for each implemented privilege level that can receive interrupts. The machine-level CSRs interact with the IMSIC's machine-level interrupt file, while, if supervisor mode is implemented, the supervisor-level CSRs interact with the IMSIC's supervisor-level interrupt file. When an IMSIC has guest interrupt files, the VS CSRs interact with a single guest interrupt file, selected by the VGEIN field of CSR `hstatus`.

For machine level, the relevant CSRs are `miselect`, `mireg`, and `mtopei`. When supervisor mode is implemented, the set of supervisor-level CSRs matches those of machine level: `siselect`, `sireg`, and `stopei`. And when the H extension is implemented, there are three corresponding VS CSRs: `vsiselect`, `vsireg`, and `vstopei`.

As explained in Chapter 2, registers `miselect` and `mireg` provide indirect access to additional machine-level registers. Likewise for supervisor-level `siselect` and `sireg`, and VS-level `vsiselect` and `vsireg` . In each case, a value of the **iselect CSR* (`miselect`, `siselect` , or `vsiselect`)) in the range 0x70-0xFF selects a register of the corresponding IMSIC interrupt file, either the machine-level interrupt file (`miselect`), the supervisor-level interrupt file (`siselect`), or a guest interrupt file (`vsiselect`).

Interrupt files at each level act identically. For a given privilege level, values of the `*iselect` CSR in the range `0x70-0xFF` select these registers of the corresponding interrupt file:

0x70	`eidelivery`
0x72	`eithreshold`
0x80	`eip0`
0x81	`eip1`
…	…
0xBF	`eip63`
0xC0	`eie0`
0xC1	`eie1`
…	…

0xFF **eie63**

Register numbers 0x71 and 0x73-0x7F are reserved. When an *iselect *CSR* has one of these values, reads from the matching *ireg *CSR* (**mireg**, **sireg**, or **vsireg**) return zero, and writes to the *ireg *CSR* are ignored. (For **vsiselect** and **vsireg**, all accesses depend on **hstatus**.VGEIN being the valid number of a guest interrupt file.)

Registers **eip0** through **eip63** contain the pending bits for all implemented interrupt identities, and are collectively called the eip *array*. Registers **eie0** through **eie63** contain the enable bits for the same interrupt identities, and are collectively called the eie *array*.

The indirectly accessed interrupt-file registers and CSRs **mtopei**, **stopei**, and **vstopei** are all documented in more detail in the next two sections.

3.8. Indirectly accessed interrupt-file registers

This section describes the registers of an interrupt file that are accessed indirectly through a *iselect *CSR* (**miselect**, **siselect**, or **vsiselect**) and its partner *ireg *CSR* (**mireg**, **sireg**, or **vsireg**). The width of these indirect accesses is always the current XLEN, 32 bits for RV32 code, or 64 bits for RV64 code.

3.8.1. External interrupt delivery enable register (eidelivery)

eidelivery is a **WARL** register that controls whether interrupts from this interrupt file are delivered from the IMSIC to the attached hart so they appear as a pending external interrupt in the hart's **mip** or **hgeip** CSR. Register **eidelivery** may optionally also support the direct delivery of interrupts from a PLIC (Platform-Level Interrupt Controller) or APLIC (Advanced PLIC) to the attached hart. Three possible values are currently defined for **eidelivery**:

0 = Interrupt delivery is disabled

1 = Interrupt delivery from the interrupt file is enabled

0x40000000 = Interrupt delivery from a PLIC or APLIC is enabled (optional)

If **eidelivery** supports value 0x40000000, then a specific PLIC or APLIC in the system may act as an alternate external interrupt controller for the attached hart at the same privilege level as this interrupt file. When **eidelivery** is 0x40000000, the interrupt file functions the same as though **eidelivery** is 0, and the PLIC or APLIC replaces the interrupt file in supplying pending external interrupts at this privilege level at the hart.

Guest interrupt files do not support value 0x40000000 for **eidelivery**.

Reset initializes **eidelivery** to 0x40000000 if that value is supported; otherwise, **eidelivery** has an UNSPECIFIED valid value (0 or 1) after reset.

eidelivery *value 0x40000000 supports system software that is oblivious to IMSICs and assumes instead that the external interrupt controller is a PLIC or APLIC. Such software may exist either because it predates the existence of IMSICs or because bypassing IMSICs is believed to reduce programming effort.*

The **eidelivery** register affects only whether an external interrupt appears in a hart's ***ip** register (MEI or SEI in **mip** or **sip**, or a bit in **hgeip**) and what the source of such an interrupt may be (either

the interrupt file or a separate external interrupt controller such as an APLIC). It has no effect on other state within the interrupt file, or on any *topei CSR (mtopei, stopei, or vstopei).

3.8.2. External interrupt enable threshold register (eithreshold)

eithreshold is a WLRL register that determines the minimum interrupt priority (maximum interrupt identity number) allowing an interrupt to be signaled from this interrupt file to the attached hart. If N is the maximum implemented interrupt identity number for this interrupt file, eithreshold must be capable of holding all values between 0 and N, inclusive.

When eithreshold is a nonzero value P, interrupt identities P and higher do not contribute to signaling interrupts, as though those identities were not enabled, regardless of the settings of their corresponding interrupt-enable bits in the eie array. When eithreshold is zero, all enabled interrupt identities contribute to signaling interrupts from the interrupt file.

3.8.3. External interrupt-pending registers (eip0-eip63)

When the current XLEN = 32, register eip$k$ contains the pending bits for interrupts with identity numbers $k \times 32$ through $k \times 32 + 31$. For an implemented interrupt identity i within that range, the pending bit for interrupt i is bit (i mod 32) of eip$k$.

When the current XLEN = 64, the odd-numbered registers eip1, eip3, … eip63 do not exist. In that case, if the *iselect CSR is an odd value in the range 0x81–0xBF, an attempt to access the matching *ireg CSR raises an illegal instruction exception, unless done in VS-mode, in which case it raises a virtual instruction exception. For even k, register eip$k$ contains the pending bits for interrupts with identity numbers $k \times 32$ through $k \times 32 + 63$. For an implemented interrupt identity i within that range, the pending bit for interrupt i is bit (i mod 64) of eip$k$.

Bit positions in a valid eip$k$ register that don't correspond to a supported interrupt identity (such as bit 0 of eip0) are read-only zeros.

3.8.4. External interrupt-enable registers (eie0-eie63)

When the current XLEN = 32, register eie$k$ contains the enable bits for interrupts with identity numbers $k \times 32$ through $k \times 32 + 31$. For an implemented interrupt identity i within that range, the enable bit for interrupt i is bit (i mod 32) of eie$k$.

When the current XLEN = 64, the odd-numbered registers eie1, eie3, … eie63 do not exist. In that case, if the *iselect CSR is an odd value in the range 0xC1–0xFF, an attempt to access the matching *ireg CSR raises an illegal instruction exception, unless done in VS-mode, in which case it raises a virtual instruction exception. For even k, register eie$k$ contains the enable bits for interrupts with identity numbers $k \times 32$ through $k \times 32 + 63$. For an implemented interrupt identity i within that range, the enable bit for interrupt i is bit (i mod 64) of eie$k$.

Bit positions in a valid eie$k$ register that don't correspond to a supported interrupt identity (such as bit 0 of eie0) are read-only zeros.

3.9. Top external interrupt CSRs (mtopei, stopei, vstopei)

CSR mtopei interacts directly with an IMSIC's machine-level interrupt file. If supervisor mode is implemented, CSR stopei interacts directly with the supervisor-level interrupt file. And if the H extension is implemented and field VGEIN of hstatus is the number of an implemented guest

interrupt file, **vstopei** interacts with the chosen guest interrupt file.

The value of a *topei *CSR* (**mtopei**, **stopei**, or **vstopei**) indicates the interrupt file's current highest-priority pending-and-enabled interrupt that also exceeds the priority threshold specified by its **eithreshold** register if **eithreshold** is not zero. Interrupts with lower identity numbers have higher priorities.

A read of a ***topei** CSR returns zero either if no interrupt is both pending in the interrupt file's **eip** array and enabled in its **eie** array, or if **eithreshold** is not zero and no pending-and-enabled interrupt has an identity number less than the value of **eithreshold**. Otherwise, the value returned from a read of ***topei** has this format:

bits 26:16	Interrupt identity
bits 10:0	Interrupt priority (same as identity)

All other bit positions are zeros.

The interrupt identity reported in a ***topei** CSR is the minor identity for an external interrupt at the hart.

*The redundancy in the value read from a **topei** CSR is consistent with the Advanced PLIC, which returns both an interrupt identity number and its priority in the same format as above, but with the two components being independent of one another.*

The value of a ***topei** CSR is not affected by an interrupt file's **eidelivery** register or by any of **mie**, **sie**, **hie**, **hgeie**, or **vsie**.

A write to a ***topei** CSR *claims* the reported interrupt identity by clearing its pending bit in the interrupt file. The value written is ignored; rather, the current readable value of the register determines which interrupt-pending bit is cleared. Specifically, when a ***topei** CSR is written, if the register value has interrupt identity i in bits 26:16, then the interrupt file's pending bit for interrupt i is cleared. When a ***topei** CSR's value is zero, a write to the register has no effect.

If a read and write of a ***topei** CSR are done together by a single CSR instruction (CSRRW, CSRRS, or CSRRC), the value returned by the read indicates the pending bit that is cleared.

*It is almost always a mistake to write to a **topei** CSR without a simultaneous read to learn which interrupt was claimed. Note especially, if a read of a **topei** register and a subsequent write to the register are done by two separate CSR instructions, then a higher-priority interrupt may become newly pending-and-enabled in the interrupt file between the two instructions, causing the write to clear the pending bit of the new interrupt and not the one reported by the read. Once the pending bit of the new interrupt is cleared, the interrupt is lost.*

*If it is necessary first to read a **topei** CSR and then subsequently claim the interrupt as a separate step, the claim can be safely done by clearing the pending bit in the **eip** array via **siselect** and **sireg**, instead of writing to **topei**.*

3.10. Interrupt delivery and handling

An IMSIC's interrupt files supply *external interrupt* signals to the attached hart, one interrupt signal per interrupt file. The interrupt signal from a machine-level interrupt file appears as bit MEIP in CSR

`mip`, and the interrupt signal from a supervisor-level interrupt file appears as bit SEIP in `mip` and `sip`. Interrupt signals from any guest interrupt files appear as the active bits in hypervisor CSR `hgeip`.

When interrupt delivery is disabled by an interrupt file's `eidelivery` register (`eidelivery` = 0), the interrupt signal from the interrupt file is held de-asserted (false). When interrupt delivery from an interrupt file is enabled (`eidelivery` = 1), its interrupt signal is asserted if and only if the interrupt file has a pending-and-enabled interrupt that also exceeds the priority threshold specified by `eithreshold`, if not zero.

Changes to the state of an interrupt file are guaranteed to be reflected in the relevant interrupt-pending bit in CSR `mip` or `hgeip` eventually, but not necessarily immediately.

A trap handler solely for external interrupts via an IMSIC could be written roughly as follows:

> save processor registers
>
> `i`=read CSR `mtopei` or `stopei`, and write simultaneously to claim the interrupt
>
> `i`= `i>>16`
>
> call the interrupt handler for external interrupt `i` (minor identity)
>
> restore processor registers
>
> return from trap

The combined read and write of `mtopei` or `stopei` in the second step can be done by a single CSRRW machine instruction,

`csrrw` *rd*, `mtopei/stopei`, `x0`

where *rd* is the destination register for value *i*.

Chapter 4. Advanced Platform-Level Interrupt Controller (APLIC)

In a RISC-V system, a Platform-Level Interrupt Controller (PLIC) handles external interrupts that are signaled through wires rather than by MSIs. When the RISC-V harts in a system do not have IMSICs, the harts themselves do not support MSIs, and all external interrupts to such harts must pass through a PLIC. But even in machines where harts have IMSICs and most interrupts are communicated via MSIs, it is not unusual for some device interrupts still to be signaled by dedicated wires. In particular, for devices (or device controllers) that do not otherwise need to initiate bus transactions in the system, the cost of supporting MSIs is especially high, so wired interrupts are a frugal alternative. Wired interrupts also continue to be universally supported by all current computer platforms, unlike MSIs, making another reason for many commodity devices or controllers to choose wired interrupts over MSIs, unless conforming to a standard like PCI Express that dictates MSIs.

This chapter specifies an *Advanced PLIC* (APLIC) that is not backward compatible with the earlier RISC-V PLIC. Full conformance to the Advanced Interrupt Architecture requires the APLIC. However, a workable system can be built substituting the older PLIC instead, assuming only wired interrupts to harts, not MSIs.

We intend eventually to provide a free example parameterized implementation of an APLIC, written in portable SystemVerilog, that we expect will be suitable for many RISC-V systems without modification.

A draft specification exists for a Duo-PLIC that is software-configurable to act as either an original RISC-V PLIC or an APLIC. However, at this time, it appears unlikely that RISC-V International will ever ratify the Duo-PLIC specification as a standard.

In a machine without IMSICs, every RISC-V hart accepts interrupts from exactly one PLIC or APLIC that is the *external interrupt controller* for that hart. A hart's external interrupt controller (the PLIC or APLIC) signals interrupts to the hart through a dedicated connection, usually a wire, for each privilege level that the hart may receive interrupts. (Recall Figure 1). A system without IMSICs will typically have only one PLIC or APLIC, serving as the external interrupt controller for all RISC-V harts.

Because every RISC-V hart without an IMSIC has exactly one PLIC or APLIC as its external interrupt controller, a system with multiple APLICs must partition the harts into disjoint subsets, making each APLIC the external interrupt controller for a separate subset of the harts. While not prohibited, this arrangement is likely to be less efficient than having all harts share a single APLIC.

RISC-V harts that employ IMSICs as their external interrupt controllers can receive external interrupts only in the form of MSIs. In that case, the role of an APLIC is to convert wired interrupts into MSIs for harts. (Recall Figure 2.) The APLIC is said to *forward* incoming wire-signaled interrupts to harts by sending MSIs to the harts.

When harts have IMSICs to support MSIs, a system may easily contain multiple APLICs for converting wired interrupts into MSIs, with each APLIC forwarding interrupts from a different subset of devices. Multiple APLICs are presumably more likely to arise when groups of devices are physically distant from one another, perhaps even on separate chips (including chiplets in a multi-chip module).

4.1. Interrupt sources and identities

An individual APLIC supports a fixed number of *interrupt sources*, corresponding exactly with the set of physical incoming interrupt wires at the APLIC. Most often, each source's incoming wire is connected to the output interrupt wire from a single device or device controller. (For level-sensitive interrupts, the interrupt outputs of multiple devices or controllers may be combined to drive the incoming wire of a single interrupt source at an APLIC. An interrupt source's incoming wire might also be simply tied high or low, if, for example, the source will always be configured as Detached. See Section 4.5.2 for a description of *source modes*.)

Each of an APLIC's interrupt sources has a fixed unique *identity number* in the range 1 to N, where N is the total number of sources at the APLIC. The number zero is not a valid interrupt identity number at an APLIC. The maximum number of interrupt sources an APLIC may support is 1023.

When an APLIC delivers interrupts directly to harts at a given privilege level (rather than forwarding interrupts as MSIs), the APLIC is the external interrupt controller for the harts at that privilege level, and the interrupt identities at the APLIC become directly the *minor identities* for external interrupts at the harts.

On the other hand, when an APLIC forwards interrupts by MSIs, software configures a new interrupt identity number for the outgoing MSIs of each source. Consequently, in this case, the source identity numbers at a given APLIC only distinguish the incoming interrupts at the APLIC and have no relevance outside the APLIC.

4.2. Interrupt domains

An APLIC supports one or more *interrupt domains*, each associated with a subset of RISC-V harts at one privilege level (machine or supervisor level). The harts within an interrupt domain are those that the domain can interrupt at the corresponding privilege level. Each domain has its own memory-mapped control region in the machine's address space that appears to control a complete, separate APLIC, though in fact all domain interfaces together access a single combined interrupt controller.

Figure 3 through Figure 5 depict some possible hierarchies of interrupt domains implemented by an APLIC in a RISC-V system.

The first figure represents a minimal system that has a single hart not supporting supervisor mode, with a single interrupt domain for machine level on that hart. The next figure, Figure 4, shows a basic arrangement for a larger system designed for symmetric multiprocessing (SMP), with multiple harts that all implement supervisor mode. In such cases, the APLIC will usually provide a separate interrupt domain for supervisor level, as the figure portrays. This supervisor-level interrupt domain allows an operating system, running in S-mode on the multiple harts, to have direct control over the interrupts it receives, avoiding the need to call upon M-mode to exercise that control.

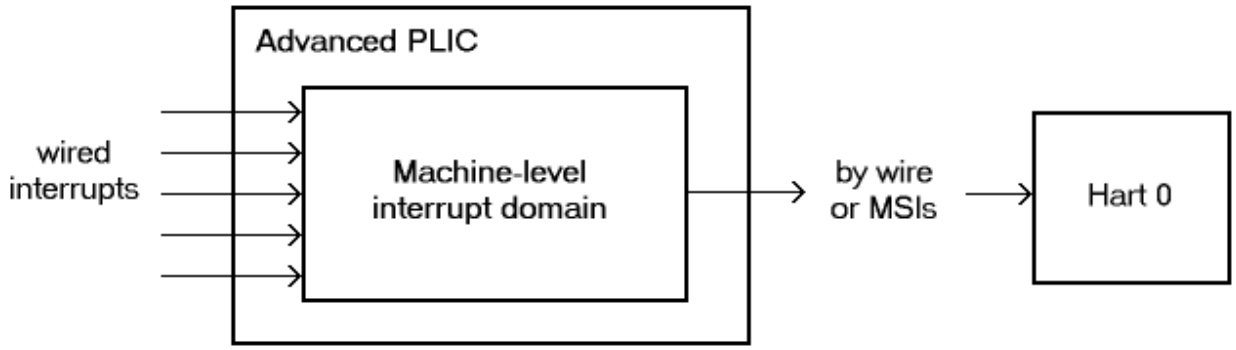

Figure 3. Example of a RISC-V system that has a single hart implementing only M-mode, with a single machine-level interrupt domain for that hart.

An APLIC's interrupt domains are arranged in a tree hierarchy, with the root domain always being at machine level. Incoming interrupt wires arrive first at the root domain. Each domain may then selectively delegate all or a subset of interrupt sources to its child domains in the hierarchy. Within a given APLIC, interrupt source numbers are invariant across all domains, so source identity number i always refers to the same source in every domain, corresponding to incoming wire number i. For an interrupt domain below the root, interrupt sources not delegated down to that domain appear to the domain as being not implemented.

Figure 5 shows a hierarchy of three interrupt domains, two at machine level and one at supervisor level. The arrangement in the figure, when combined with PMP (physical memory protection), allows machine-level software to isolate a selection of interrupts exclusively for hart 0, beyond the reach of the four application harts, even at machine level.

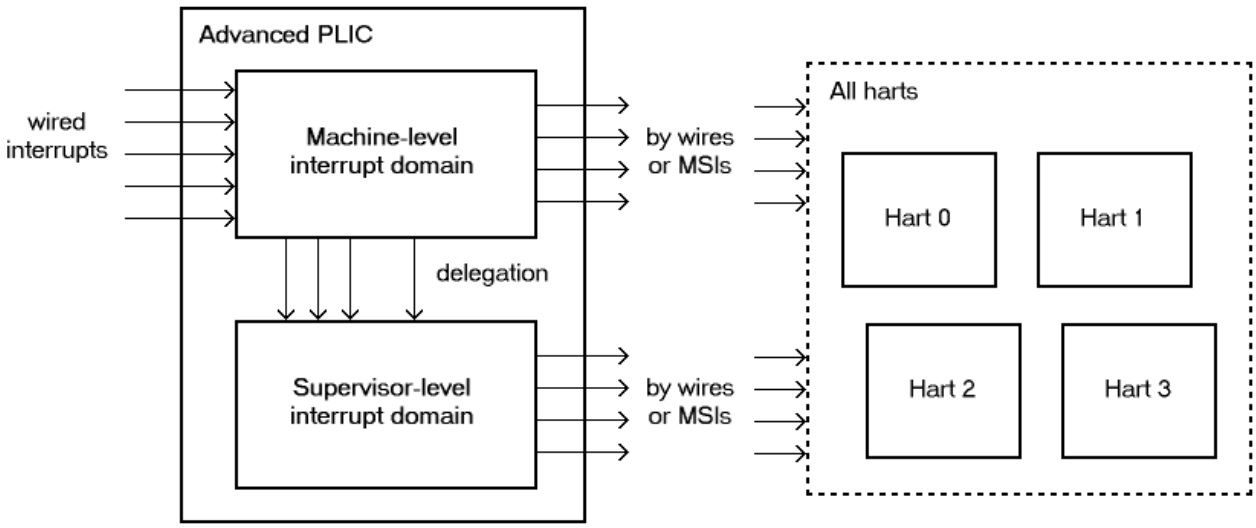

Figure 4. An example system with four harts that implement M-mode and S-mode, with two APLIC interrupt domains, one each for machine and supervisor levels.

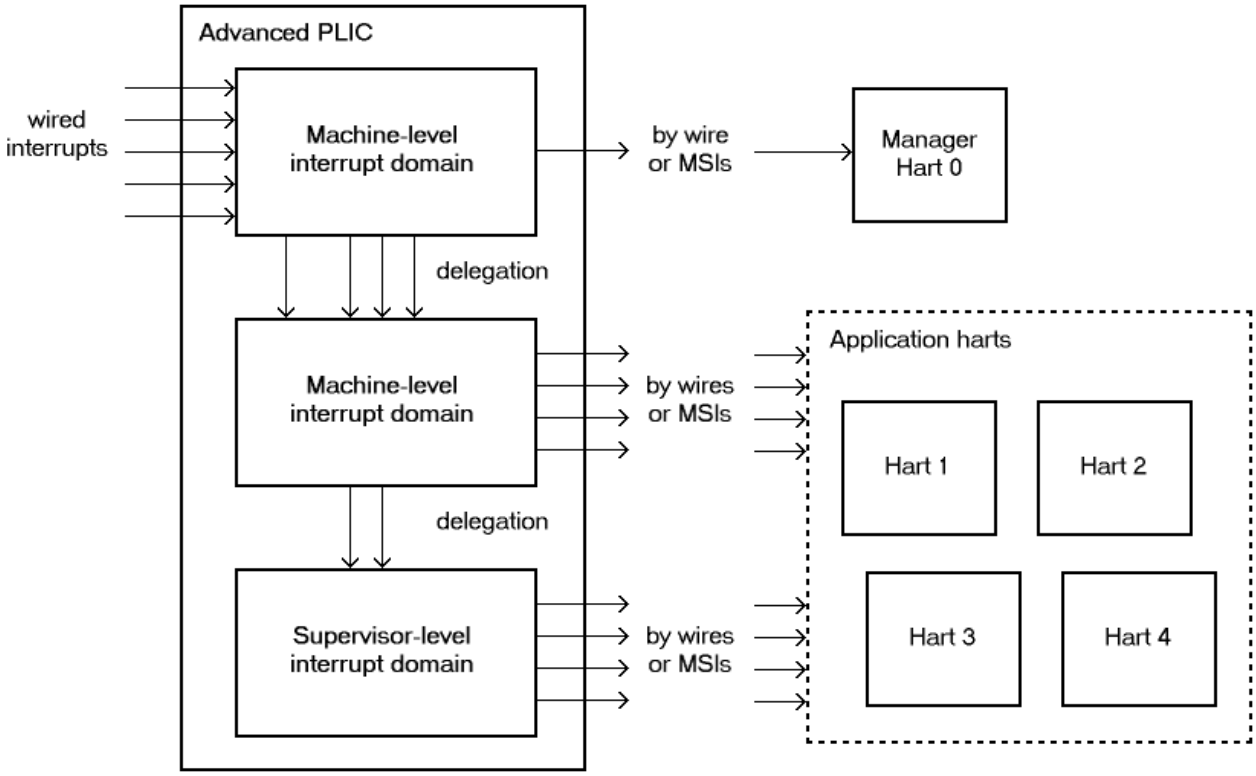

Figure 5. A RISC-V system that extends the example of Figure 4 with a fifth M-mode-only "manager" hart, with a separate machine-level interrupt domain above the other domains.

> *In order for the harts within an interrupt domain to have direct control over the interrupts from the domain, the harts must be cooperatively controlled by software at the same privilege level. In particular, a single operating system should control all of the harts associated with a supervisor-level interrupt domain. In the examples of Figure 4 and Figure 5, control of the APLIC's supervisor-level interrupt domain could not be safely split among multiple independent OSes.*
>
> *Given the domain hierarchies depicted in the figures, if it were necessary to partition the application harts for multiple OSes, machine-level software would need to prevent direct OS access to the supervisor-level interrupt domain and instead provide SBI services for controlling APLIC interrupts or, alternatively, emulate the control interfaces of separate supervisor-level interrupt domains, one for each OS. Note that such emulation might still make use of the APLIC's physical supervisor-level interrupt domain, but under the control of machine-level software.*

An APLIC's interrupt domain hierarchy satisfies these rules:

- The root domain is at machine level.
- The parent of any supervisor-level interrupt domain is a machine-level domain that includes at least the same harts (but at machine level, obviously). The parent domain may have a larger set of harts at machine level.
- For each interrupt domain, interrupts from the domain are signaled to harts all by the same method, either by wire or by MSIs, not by a mixture of methods among the harts.

When a RISC-V hart's external interrupt controller is an APLIC, not an IMSIC, the hart can be within only one interrupt domain of this APLIC at each privilege level.

On the other hand, a hart that has an IMSIC for its external interrupt controller may, at each privilege level, be in multiple APLIC interrupt domains, even those of the same APLIC, and may potentially receive MSIs from multiple different APLICs in the machine.

A platform might give software a way to choose between multiple interrupt domain hierarchies for any given APLIC. Any such configurability is outside the scope of this specification, but should be available to machine level only.

4.3. Hart index numbers

Within a given interrupt domain, each of the domain's harts has a unique *index number* in the range 0 to $2^{14}-1$ (= 16,383). The index number a domain associates with a hart may or may not have any relationship to the unique hart identifier ("hart ID") that the Privileged Architecture assigns to the hart. Two different interrupt domains may employ a different mapping of index numbers to the same set of harts. However, if any of an APLIC's interrupt domains can forward interrupts by MSI, then all machine-level domains of the APLIC share a common mapping of index numbers to harts.

For efficiency, implementations should prefer small integers for hart index numbers.

4.4. Overview of interrupt control for a single domain

Each interrupt domain implemented by an APLIC has its own separate physical control interface that is memory-mapped in the machine's address space, allowing access to each domain to be easily regulated by both PMP (physical memory protection) and page-based address translation. The control interfaces of all interrupt domains have a common structure. In most respects, every domain appears to software as though it were a root domain, without visibility of the domains above it in the hierarchy.

An individual interrupt domain has the following components for each interrupt source at the APLIC:

- Source configuration. This determines whether the specific source is active in the domain and, if so, how the incoming wire is to be interpreted, such as level-sensitive or edge-sensitive. For a source that is inactive in the domain, source configuration controls any delegation to a child domain.
- Interrupt-pending and interrupt-enable bits. For an inactive source, these two bits are read-only zeros. Otherwise, the pending bit records an interrupt that arrived and has not yet been signaled or forwarded, while the enable bit determines whether interrupts from this source should currently be delivered, or should remain pending.
- Target selection. For an active source, target selection determines the hart to receive the interrupt and either the interrupt's priority or the new interrupt identity when forwarding as an MSI.

For interrupt domains that deliver interrupts directly to harts rather than forwarding by MSIs, the domain has a final set of components for controlling interrupt delivery to harts, one instance per hart in the domain.

Although an APLIC with multiple interrupt domains may appear to duplicate the per-source state listed above (source configuration, etc.) by a factor equal to the number of domains, in fact, APLIC implementations can exploit the fact that each source is ultimately active in only one domain. In all domains to which a specific interrupt source has not been delegated, the state associated with the source appears as read-only zeros,

requiring no physical register bits.

4.5. Memory-mapped control region for an interrupt domain

For each interrupt domain that an APLIC supports, there is a dedicated memory-mapped control region for managing interrupts in that domain. This control region is a multiple of 4 KiB in size and aligned to a 4-KiB address boundary. The smallest valid control region is 16 KiB. An interrupt domain's control region is populated by a set of 32-bit registers. The first 16 KiB contains the registers listed in Table 6.

Table 6. The registers of the first 16 KiB of an interrupt domain's memory-mapped control region.

offset	size	register name	
0x0000	4 bytes	domaincfg	
0x0004	4 bytes	sourcecfg[1]	
0x0008	4 bytes	sourcecfg[2]	
...		...	
0x0FFC	4 bytes	sourcecfg[1023]	
0x1BC0	4 bytes	mmsiaddrcfg	(machine-level interrupt domains only)
0x1BC4	4 bytes	mmsiaddrcfgh	"
0x1BC8	4 bytes	smsiaddrcfg	"
0x1BCC	4 bytes	smsiaddrcfgh	"
0x1C00	4 bytes	setip[0]	
0x1C04	4 bytes	setip[1]	
...		...	
0x1C7C	4 bytes	setip[31]	
0x1CDC	4 bytes	setipnum	
0x1D00	4 bytes	in_clrip[0]	
0x1D04	4 bytes	in_clrip[1]	
...		...	
0x1D7C	4 bytes	in_clrip[31]	
0x1DDC	4 bytes	clripnum	
0x1E00	4 bytes	setie[0]	
0x1E04	4 bytes	setie[1]	
...		...	
0x1E7C	4 bytes	setie[31]	
0x1EDC	4 bytes	setienum	
0x1F00	4 bytes	clrie[0]	
0x1F04	4 bytes	clrie[1]	

...		...
0x1F7C	4 bytes	clrie[31]
0x1FDC	4 bytes	clrienum
0x2000	4 bytes	setipnum_le
0x2004	4 bytes	setipnum_be
0x3000	4 bytes	genmsi
0x3004	4 bytes	target[1]
0x3008	4 bytes	target[2]
...		...
0x3FFC	4 bytes	target[1023]

Starting at offset **0x4000**, an interrupt domain's control region may optionally have an array of *interrupt delivery control* (IDC) structures, one for each potential hart index number in the range 0 to some maximum that is at least as large as the maximum hart index number for the interrupt domain. IDC structures are used only when the domain is configured to deliver interrupts directly to harts instead of being forwarded by MSIs. An interrupt domain that supports only interrupt forwarding by MSIs and not the direct delivery of interrupts by the APLIC does not need IDC structures in its control region.

The first IDC structure, if any, is for the hart with index number 0; the second is for the hart with index number 1; and so forth. Each IDC structure is 32 bytes and has these defined registers:

offset	size	register name
0x00	4 bytes	idelivery
0x04	4 bytes	iforce
0x08	4 bytes	ithreshold
0x18	4 bytes	topi
ox1C	4 bytes	claimi

IDC structures are packed contiguously, 32 bytes per structure, so the offset from the beginning of an interrupt domain's control region to its second IDC structure (hart index 1), if it exists, is **0x4020**; the offset to the third IDC structure (hart index 2), if it exists, is **0x4040**; etc.

The array of IDC structures may include some for *potential* hart index numbers that are not *actual* hart index numbers in the domain. For example, the first IDC structure is always for hart index 0, but 0 is not necessarily a valid index number for any hart in the domain. For each IDC structure in the array that does not correspond to a valid hart index number in the domain, the IDC structure's registers may (or may not) be all read-only zeros.

Aside from the registers in Table 6 and those listed above for IDC structures, all other bytes in an interrupt domain's control region are reserved and are implemented as read-only zeros.

Only naturally aligned 32-bit simple reads and writes are supported within an interrupt domain's control region. Writes to read-only bytes are ignored. For other forms of accesses (other sizes, misaligned accesses, or AMOs), implementations should preferably report an access fault or bus error but must otherwise ignore the access.

The registers of the first 16 KiB of an interrupt domain's control region (all but the IDC structures) are documented individually below. IDC structures are documented later, in Section 4.8, "Interrupt delivery directly by the APLIC."

4.5.1. Domain configuration (`domaincfg`)

The **`domaincfg`** register has this format:

bits 31:24	read-only 0x80
bit 8	IE
bit 7	read-only 0
bit 2	DM (**WARL**)
bit 0	BE (**WARL**)

All other register bits are reserved and read as zeros.

Bit IE (Interrupt Enable) is a global enable for all active interrupt sources at this interrupt domain. Only when IE = 1 are pending-and-enabled interrupts actually signaled or forwarded to harts.

The value of bit IE affects only whether interrupts are delivered to harts. It has no effect on any other APLIC state, including the interrupt-enable and interrupt-pending bits of interrupt sources and IDC registers **`idelivery`**, **`topi`**, and **`claimi`**.

Field DM (Delivery Mode) is **WARL** and determines how this interrupt domain delivers interrupts to harts. The two possible values for DM are:

0 = direct delivery mode

1 = MSI delivery mode

In *direct delivery mode*, interrupts are prioritized and signaled directly to harts by the APLIC itself. In *MSI delivery mode*, interrupts are forwarded by the APLIC as MSIs to harts, presumably for further handling by IMSICs at those harts. A given APLIC implementation may support either or both of these delivery modes for each interrupt domain.

If the interrupt domain's harts have IMSICs, then unless the relevant interrupt files of those IMSICs support value **`0x40000000`** for register **`eidelivery`**, setting DM to zero (direct delivery mode) will have the same effect as setting IE to zero. See Section 3.8.1 and Section 4.8.2.

BE (Big-Endian) is a **WARL** field that determines the byte order for most registers in the interrupt domain's memory-mapped control region. If BE = 0, byte order is little-endian, and if BE = 1, it is big-endian. For RISC-V systems that support only little-endian, BE may be read-only zero, and for those that support only big-endian, BE may be read-only one. For bi-endian systems, BE is writable.

Field BE affects the byte order of accesses to the **`domaincfg`** register itself, just as for other registers in the interrupt domain's control region. To deal with this fact, the read-only value in **`domaincfg`**'s most-significant byte, bits 31:24, serves two purposes. First, for any read of **`domaincfg`**, the register's correct byte order is easily determined from the four-byte value obtained: When interpreted in the correct byte order, bit 31 is one, and in the wrong order, bit 31 is zero. Second, if the value of BE is uncertain (prior to software initializing the interrupt domain, presumably), an 8-bit value x can be safely written to **`domaincfg`** by writing $(x<<24)|x$, where <<24 represents shifting left by 24 bits, and the vertical bar

(|) represents bitwise logical OR. After **domaincfg** is written once, the value of BE should then be known, so subsequent writes should not need to repeat the same trick.

At system reset, all writable bits in **domaincfg** are initialized to zero, including IE. If an implementation supports additional forms of reset for the APLIC, it is implementation-defined (or possibly platform-defined) how these other resets may affect **domaincfg**.

4.5.2. Source configurations (sourcecfg[1]-sourcecfg[1023])

For each possible interrupt source i, register **sourcecfg**[i] controls the *source mode* for source i in this interrupt domain as well as any delegation of the source to a child domain. When source i is not implemented, or appears in this domain not to be implemented, **sourcecfg**[i] is read-only zero. If source i was not delegated to this domain and is then changed (at the parent domain) to become delegated to this domain, **sourcecfg**[i] remains zero until successfully written with a nonzero value.

Bit 10 of **sourcecfg**[i] is a 1-bit field called D (Delegate). If D = 1, source i is delegated to a child domain, and if D = 0, it is not delegated to a child domain. Interpretation of the rest of **sourcecfg**[i] depends on field D.

When interrupt source i is delegated to a child domain, **sourcecfg**[i] has this format:

bit 10	D, =1
bits 9:0	Child Index (**WLRL**)

All other register bits are reserved and read as zeros.

Child Index is a **WLRL** field that specifies the interrupt domain to which this source is delegated. For an interrupt domain with C child domains, this field must be able to hold integer values in the range 0 to $C-1$. Each interrupt domain has a fixed mapping from these index numbers to child domains.

If an interrupt domain has no children in the domain hierarchy, bit D cannot be set to one in any **sourcecfg** register for that domain. For such a leaf domain, attempting to write a **sourcecfg** register with a value that has bit 10 = 1 causes the entire register to be set to zero instead.

When interrupt source i is not delegated to a child domain, **sourcecfg**[i] has this format:

bit 10	D, =0
bits 2:0	SM (**WARL**)

All other register bits are reserved and read as zeros.

The SM (Source Mode) field is **WARL** and controls whether the interrupt source is active in this domain, and if so, what values or transitions on the incoming wire are interpreted as interrupts. The values allowed for SM and their meanings are listed in Table 7. Inactive (zero) is always supported for field SM. Implementations are free to choose, independently for each interrupt source, what other values are supported for SM.

Table 7. Encoding of the SM (Source Mode) field of a sourcecfg register when bit D = 0

Value	Name	Description
0	Inactive	Inactive in this domain (and not delegated)

Value	Name	Description
1	Detached	Active, detached from the source wire
2–3	—	*Reserved*
4	Edge1	Active, edge-sensitive; interrupt asserted on rising edge
5	Edge0	Active, edge-sensitive; interrupt asserted on falling edge
6	Level1	Active, level-sensitive; interrupt asserted when high
7	Level0	Active, level-sensitive; interrupt asserted when low

An interrupt source is inactive in the interrupt domain if either the source is delegated to a child domain (D = 1) or it is not delegated (D = 0) and SM is Inactive. Whenever interrupt source i is inactive in an interrupt domain, the corresponding interrupt-pending and interrupt-enable bits within the domain are read-only zeros, and register `target[`i`]` is also read-only zero. If source i is changed from inactive to an active mode, the interrupt source's pending and enable bits remain zeros, unless set automatically for a reason specified later in this section or in Section 4.7, and the defined subfields of `target[`i`]` obtain UNSPECIFIED values.

When a source is configured as Detached, its wire input is ignored; however, the interrupt-pending bit may still be set by a write to a `setip` or `setipnum` register. (This mode can be useful for receiving MSIs, for example.)

An edge-sensitive source can be configured to recognize an incoming interrupt on either a rising edge (low-to-high transition) or a falling edge (high-to-low transition). When configured for a falling edge (mode Edge0), the source is said to be *inverted*.

A level-sensitive source can be configured to interpret either a high level (1) or a low level (0) on the wire as the assertion of an interrupt. When configured for a low level (mode Level0), the source is said to be *inverted*.

For an interrupt source that is configured as edge-sensitive or level-sensitive, define

rectified input value = (incoming wire value) XOR (source is inverted).

For a source that is inactive or Detached, the *rectified input value* is zero.

Any write to a `sourcecfg` register might (or might not) cause the corresponding interrupt-pending bit to be set to one if the rectified input value is high (= 1) under the new source mode. A write to a `sourcecfg` register will not by itself cause a pending bit to be cleared except when the source is made inactive. (But see Section 4.7.)

4.5.3. Machine MSI address configuration (`mmsiaddrcfg` and `mmsiaddrcfgh`)

For machine-level interrupt domains, registers `mmsiaddrcfg` and `mmsiaddrcfgh` may optionally provide parameters used to determine the addresses to write outgoing MSIs.

If no interrupt domain of the APLIC supports MSI delivery mode (`domaincfg`.DM is read-only zero for all domains), these two registers are not implemented for any domain. Otherwise, they are implemented for the root domain, and may or may not be implemented for other machine-level domains. For domains not at machine level, they are never implemented. When a domain does not implement `mmsiaddrcfg` and `mmsiaddrcfgh`, the eight bytes at their locations are simply read-only zeros like other reserved bytes.

Registers `mmsiaddrcfg` and `mmsiaddrcfgh` are potentially writable only for the root domain. For all other machine-level domains that implement them, they are read-only.

When implemented, `mmsiaddrcfg` has this format:

bits 31:0	Low Base PPN (**WARL**)

and `mmsiaddrcfgh` has this format:

bit 31	L
bits 28:24	HHXS (**WARL**)
bits 22:20	LHXS (**WARL**)
bits 18:16	HHXW (**WARL**)
bits 15:12	LHXW (**WARL**)
bits 11:0	High Base PPN (**WARL**)

All other bits of `mmsiaddrcfgh` are reserved and read as zeros.

Fields High Base PPN from `mmsiaddrcfgh` and Low Base PPN from `mmsiaddrcfg` concatenate to form a 44-bit Base PPN (Physical Page Number). The use of this value and fields HHXS (High Hart Index Shift), LHXS (Low Hart Index Shift), HHXW (High Hart Index Width), and LHXW (Low Hart Index Width) for determining target addresses for MSIs is described later, in Section 4.9.1.

When `mmsiaddrcfg` and `mmsiaddrcfgh` are writable (root domain only), all fields other than L are **WARL**. An implementation is free to choose what values are supported. Typically, some bits are writable while others are read-only constants. In the extreme, the values of all fields may be entirely constant, fixed by the implementation.

If bit L in `mmsiaddrcfgh` is set to one, `mmsiaddrcfg` and `mmsiaddrcfgh` are *locked*, and writes to the registers are ignored, making the registers effectively read-only. When L = 1, the other fields in `mmsiaddrcfg` and `mmsiaddrcfgh` may optionally all read as zeros. In that case, if these other fields were given nonzero values when L was first set in the root domain, their values are retained internally by the APLIC but become no longer visible by reading `mmsiaddrcfg` and `mmsiaddrcfgh`.

Setting `mmsiaddrcfgh`.L to one also locks registers `smsiaddrcfg` and `smsiaddrcfgh` described in the next subsection, if those registers are implemented as well.

For the root domain, L is initialized at system reset to either zero or one, whichever is deemed appropriate for the specific APLIC implementation. If reset initializes L to one, either the other fields are hardwired by the APLIC to constants, or the APLIC has a different means, outside of this standard, for determining the addresses of outgoing MSI writes. In the latter case, the other fields in `mmsiaddrcfg` and `mmsiaddrcfgh` may all read as zeros, so registers `mmsiaddrcfg` and `mmsiaddrcfgh` have only read-only values zero and `0x80000000` respectively. Any time `mmsiaddrcfg` or `mmsiaddrcfgh` has a different value (not zero or `0x80000000` respectively), the addresses for outgoing MSI writes directed to machine level must be derivable from the visible values of these registers, as specified in Section 4.9.1.

For machine-level domains that are not the root domain, if these registers are implemented, bit L is always one, and the other fields either are read-only copies of `mmsiaddrcfg` and `mmsiaddrcfgh` from the root domain, or are all zeros.

Giving software the ability to arbitrarily determine the addresses to which MSIs are sent, even if allowed only for machine level, permits bypassing physical memory protection (PMP). For APLICs that support MSI delivery mode, it is recommended, if feasible, that the APLIC internally hardwire the physical addresses for all target IMSICs, putting those addresses beyond the reach of software to change. However, not all APLIC implementations will be able to follow that recommendation.

It is expected that most systems will arrange the physical addresses of target IMSICs in a simple linear correspondence with hart index numbers. (See Section 3.6.) Registers `mmsiaddrcfg` *and* `mmsiaddrcfgh` *(along with* `smsiaddrcfg` *and* `smsiaddrcfgh` *from the next subsection) allow sufficiently trusted machine-level software, early after system reset, to configure the pattern of physical addresses for target IMSICs and then lock this configuration against subsequent tampering.*

APLICs that actually hardwire the IMSIC addresses internally can implement these registers simply as read-only with values zero and `0x80000000`*. Or, if the IMSIC addresses must be configured by software but the formula is too complex for registers* `mmsiaddrcfg` *and* `mmsiaddrcfgh` *to handle, again the registers can be implemented simply as read-only with values zero and* `0x80000000`*, and a separate, custom mechanism supplied for configuring the IMSIC addresses.*

If an APLIC supports additional forms of reset besides system reset, it is implementation-defined (or possibly platform-defined) how these other resets may affect `mmsiaddrcfg` and `mmsiaddrcfgh` (as well as `smsiaddrcfg` and `smsiaddrcfgh`) in the root domain. However, it must not be possible for insufficiently privileged software to use a localized reset to unlock these registers by changing bit L back to zero. For this reason, it is likely that only a complete system reset affects these registers, and any other resets do not.

4.5.4. Supervisor MSI address configuration (`smsiaddrcfg` and `smsiaddrcfgh`)

For machine-level interrupt domains, registers `smsiaddrcfg` and `smsiaddrcfgh` may optionally provide parameters used by supervisor-level domains to determine the addresses to write outgoing MSIs.

Registers `smsiaddrcfg` and `smsiaddrcfgh` are implemented by a domain if the domain implements `mmsiaddrcfg` and `mmsiaddrcfgh` and the APLIC has at least one supervisor-level interrupt domain. If the registers are not implemented, the eight bytes at their locations are simply read-only zeros like other reserved bytes.

Like `mmsiaddrcfg` and `mmsiaddrcfgh`, registers `smsiaddrcfg` and `smsiaddrcfgh` are potentially writable only for the root domain. For all other machine-level domains that implement them, they are read-only.

When implemented, `smsiaddrcfg` has this format:

bits 31:0 Low Base PPN (**WARL**)

and `smsiaddrcfgh` has this format:

bits 22:20 LHXS (**WARL**)

bits 11:0 High Base PPN (**WARL**)

All other bits of **smsiaddrcfgh** are reserved and read as zeros.

Fields High Base PPN from **smsiaddrcfgh** and Low Base PPN from **smsiaddrcfg** concatenate to form a 44-bit Base PPN (Physical Page Number). The use of this value and field LHXS (Low Hart Index Shift) for determining target addresses for MSIs is described later, in Section 4.9.1.

When **smsiaddrcfg** and **smsiaddrcfgh** are writable (root domain only), all fields are **WARL**. An implementation is free to choose what values are supported, just as for **mmsiaddrcfg** and **mmsiaddrcfgh**.

If register **mmsiaddrcfgh** of the domain has bit L set to one, then **smsiaddrcfg** and **smsiaddrcfgh** are *locked* as read-only alongside **mmsiaddrcfg** and **mmsiaddrcfgh**. When **mmsiaddrcfgh.L** = 1, if the readable values of **mmsiaddrcfg** and **mmsiaddrcfgh** are zero and **0x80000000** respectively—because their other fields are hidden—then **smsiaddrcfg** and **smsiaddrcfgh** are hidden also and read as zeros.

For the root domain only, if **mmsiaddrcfgh.L** = 1 and the MSI-address-configuration fields are hidden (so **mmsiaddrcfgh** reads as **0x80000000** and registers **mmsiaddrcfg**, **smsiaddrcfg**, and **smsiaddrcfgh** all read as zeros), then whatever values **smsiaddrcfg** and **smsiaddrcfgh** had when **mmsiaddrcfgh**.L was first set are retained internally by the APLIC, though those values are no longer visible by reading the registers. Alternatively, if system reset initializes **mmsiaddrcfgh.L** = 1 in the root domain, and if all MSI-address-configuration fields never appear as anything other than zeros, then the APLIC implementation has some other, possibly nonstandard, means for determining the addresses of outgoing MSIs, as discussed in the previous subsection, Section 4.5.3.

Any time **mmsiaddrcfg** and **mmsiaddrcfgh** are not read-only zero and **0x80000000** respectively, the addresses for outgoing MSI writes directed to supervisor level must be derivable from the visible values of registers **mmsiaddrcfgh**, **smsiaddrcfg**, and **smsiaddrcfgh**, as specified in Section 4.9.1.

For machine-level domains that are not the root domain, if **smsiaddrcfg** and **smsiaddrcfgh** are implemented and are not read-only zeros, then they are read-only copies of the same registers from the root domain.

4.5.5. Set interrupt-pending bits (setip[0]-setip[31])

Reading or writing **setip**[k] register reads or potentially modifies the pending bits for interrupt sources $k \times 32$ through $k \times 32 + 31$. For an implemented interrupt source i within that range, the pending bit for source i corresponds with register bit (i mod 32).

A read of a **setip** register returns the pending bits of the corresponding interrupt sources. Bit positions in the result value that do not correspond to an implemented interrupt source (such as bit 0 of **setip[0]**) are zeros.

On a write to a **setip** register, for each bit that is one in the 32-bit value written, if that bit position corresponds to an active interrupt source, the interrupt-pending bit for that source is set to one if possible. See Section 4.7 for exactly when a pending bit may be set by writing to a **setip** register.

4.5.6. Set interrupt-pending bit by number (setipnum)

If i is an active interrupt source number in the domain, writing 32-bit value i to register **setipnum** causes the pending bit for source i to be set to one if possible. See Section 4.7 for exactly when a pending bit may be set by writing to **setipnum**.

A write to **setipnum** is ignored if the value written is not an active interrupt source number in the

domain. A read of `setipnum` always returns zero.

4.5.7. Rectified inputs, clear interrupt-pending bits (`in_clrip[0]`-`in_clrip[31]`)

Reading register `in_clrip[`k`]` returns the rectified input (Section 4.5.2) for interrupt sources $k \times 32$ through $k \times 32 + 31$, while writing `in_clrip[`k`]` potentially modifies the pending bits for the same sources. For an implemented interrupt source i within the specified range, source i corresponds with register bit (i mod 32).

A read of an `in_clrip` register returns the rectified input values of the corresponding interrupt sources. Bit positions in the result value that do not correspond to an implemented interrupt source (such as bit 0 of `in_clrip[0]`) are zeros.

On a write to an `in_clrip` register, for each bit that is one in the 32-bit value written, if that bit position corresponds to an active interrupt source, the interrupt-pending bit for that source is cleared if possible. See Section 4.7 for exactly when a pending bit may be cleared by writing to an `in_clrip` register.

4.5.8. Clear interrupt-pending bit by number (`clripnum`)

If i is an active interrupt source number in the domain, writing 32-bit value i to register `clripnum` causes the pending bit for source i to be cleared if possible. See Section 4.7 for exactly when a pending bit may be cleared by writing to `clripnum`.

A write to `clripnum` is ignored if the value written is not an active interrupt source number in the domain. A read of `clripnum` always returns zero.

4.5.9. Set interrupt-enable bits (`setie[0]`-`setie[31]`)

Reading or writing register `setie[`k`]` reads or potentially modifies the enable bits for interrupt sources $k \times 32$ through $k \times 32 + 31$. For an implemented interrupt source i within that range, the enable bit for source i corresponds with register bit i mod 32.

A read of a `setie` register returns the enable bits of the corresponding interrupt sources. Bit positions in the result value that do not correspond to an implemented interrupt source (such as bit 0 of `setie[0]`) are zeros.

On a write to a `setie` register, for each bit that is one in the 32-bit value written, if that bit position corresponds to an active interrupt source, the interrupt-enable bit for that source is set to one.

4.5.10. Set interrupt-enable bit by number (`setienum`)

If i is an active interrupt source number in the domain, writing 32-bit value i to register `setienum` causes the enable bit for source i to be set to one.

A write to `setienum` is ignored if the value written is not an active interrupt source number in the domain. A read of `setienum` always returns zero.

4.5.11. Clear interrupt-enable bits (`clrie[0]`-`clrie[31]`)

Writing register `clrie[`k`]` potentially modifies the enable bits for interrupt sources $k \times 32$ through $k \times 32 + 31$. For an implemented interrupt source i within that range, the enable bit for source i

corresponds with register bit $i \bmod 32$.

On a write to a **clrie** register, for each bit that is one in the 32-bit value written, the interrupt-enable bit for that source is cleared.

A read of a **clrie** register always returns zero.

4.5.12. Clear interrupt-enable bit by number (clrienum)

If i is an active interrupt source number in the domain, writing 32-bit value i to register **clrienum** causes the enable bit for source i to be cleared.

A write to **clrienum** is ignored if the value written is not an active interrupt source number in the domain. A read of **clrienum** always returns zero.

4.5.13. Set interrupt-pending bit by number, little-endian (setipnum_le)

Register **setipnum_le** acts identically to **setipnum** except that byte order is always little-endian, as though field BE (Big-Endian) of register **domaincfg** is zero.

For systems that are big-endian-only, with **domaincfg**.BE hardwired to one, **setipnum_le** need not be implemented, in which case the four bytes at this offset are simply read-only zeros like other reserved bytes.

setipnum_le may be used as a write port for MSIs.

4.5.14. Set interrupt-pending bit by number, big-endian (setipnum_be)

Register **setipnum_be** acts identically to **setipnum** except that byte order is always big-endian, as though field BE (Big-Endian) of register **domaincfg** is one.

For systems that are little-endian-only, with **domaincfg**.BE hardwired to zero, **setipnum_be** need not be implemented, in which case the four bytes at this offset are simply read-only zeros like other reserved bytes.

For systems built mainly for big-endian byte order, **setipnum_be** may be useful as a write port for MSIs from some devices.

4.5.15. Generate MSI (genmsi)

When the interrupt domain is configured in MSI delivery mode (**domaincfg**.DM = 1), register **genmsi** can be used to cause an *extempore* MSI to be sent from the APLIC to a hart. The main purpose for this function is to assist in establishing a temporary known ordering between a hart's writes to the APLIC's registers and the transmission of MSIs from the APLIC to the hart, as explained later in Section 4.9.3.

For other purposes, sending an MSI to a hart is usually better done by writing directly to the hart's IMSIC, rather than employing an APLIC as an intermediary. Use of the **genmsi** *register should be minimized to avoid it becoming a bottleneck.*

Register **genmsi** has this format:

bits 31:18 Hart Index (**WLRL**)

bits 12	Busy (read-only)
bits 10:0	EIID (**WARL**)

All other register bits are reserved and read as zeros.

The Busy bit is ordinarily zero (false), but a write to **genmsi** causes Busy to become one (true), indicating an extempore MSI is pending. The Hart Index field specifies the destination hart, and EIID (External Interrupt Identity) specifies the data value for the MSI. Fields Hart Index and EIID have the same formats and behavior as in a **target** register, documented in the next subsection, Section 4.5.16. For a machine-level interrupt domain, an extempore MSI is sent to the destination hart at machine level, and for a supervisor-level interrupt domain, an extempore MSI is sent to the destination hart at supervisor level.

A pending extempore MSI should be sent by the APLIC with minimal delay. Once it has left the APLIC and the APLIC is able to accept a new write to **genmsi** for another extempore MSI, Busy reverts to false. All MSIs previously sent from this APLIC to the same hart must be visible at the hart's IMSIC before the extempore MSI becomes visible at the hart's IMSIC.

While Busy is true, writes to **genmsi** are ignored.

Extempore MSIs are not affected by the IE bit of the domain's **domaincfg** register. An extempore MSI is sent even if **domaincfg**.IE = 0.

When the interrupt domain is configured in direct delivery mode (**domaincfg**.DM = 0), register **genmsi** is read-only zero.

4.5.16. Interrupt targets (target[1]-target[1023])

If interrupt source i is inactive in this domain, register **target[**i**]** is read-only zero. If source i is active, **target[**i**]** determines the hart to which interrupts from the source are signaled or forwarded. The exact interpretation of **target[**i**]** depends on the delivery mode configured by field DM of register **domaincfg**.

If **domaincfg**.DM is changed, the **target** registers for all active interrupt sources within the domain obtain UNSPECIFIED values in all fields defined for the new delivery mode.

4.5.16.1. Active source, direct delivery mode

For an active interrupt source i, if the domain is configured in direct delivery mode (**domaincfg**.DM = 0), then register **target[**i**]** has this format:

bits 31:18	Hart Index (**WLRL**)
bits 7:0	IPRIO (**WARL**)

All other register bits are reserved and read as zeros.

Hart Index is a **WLRL** field that specifies the hart to which interrupts from this source will be delivered.

Field IPRIO (Interrupt Priority) specifies the *priority number* for the interrupt source. This field is a **WARL** unsigned integer of *IPRIOLEN* bits, where IPRIOLEN is a constant parameter for the given APLIC, in the range of 1 to 8. Only values 1 through $2^{\text{IPRIOLEN}} - 1$ are allowed for IPRIO, not zero. A write

to a `target` register sets IPRIO equal to bits ($IPRIOLEN-1$):0 of the 32-bit value written, unless those bits are all zeros, in which case the priority number is set to 1 instead. (If IPRIOLEN = 1, these rules cause IPRIO to be effectively read-only with value 1.)

Smaller priority numbers convey higher priority. When interrupt sources have equal priority number, the source with the lowest identity number has the highest priority.

Interrupt priorities are encoded as integers, with smaller numbers denoting higher priority, to match the encoding of priorities by IMSICs.

4.5.16.2. Active source, MSI delivery mode

For an active interrupt source i, if the domain is configured in MSI delivery mode (`domaincfg`.DM = 1), then register `target[`i`]` has this format:

bits 31:18 Hart Index (**WLRL**)

bits 17:12 Guest Index (**WLRL**)

bits 10:0 EIID (**WARL**)

Bit 11 is reserved and reads as zero.

The Hart Index field specifies the hart to which interrupts from this source will be forwarded.

If the interrupt domain is at supervisor level and the domain's harts implement the H extension, then Guest Index is a **WLRL** field that must be able to hold all integer values in the range 0 through GEILEN. (Parameter *GEILEN* is defined by the H extension.) Otherwise, field Guest Index is read-only zero. For a supervisor-level interrupt domain, a nonzero Guest Index is the number of the target hart's guest interrupt file to which MSIs will be sent. When Guest Index is zero, MSIs from a supervisor-level domain are forwarded to the target hart at supervisor level. For a machine-level domain, Guest Index is read-only zero, and MSIs are forwarded to a target hart always at machine level.

Together, fields Hart Index and Guest Index of register `target[`i`]` determine the address for MSIs forwarded for interrupt source i. The remaining field EIID (External Interrupt Identity) specifies the data value for those MSIs, eventually becoming the minor identity for an external interrupt at the target hart.

If the interrupt domain's harts have IMSIC interrupt files that implement N distinct interrupt identities (Section 3.1), then EIID is a k-bit unsigned integer field, where $\lceil \log_2 N \rceil \le k \le 11$. EIID is thus able to hold at least values 0 through N. A write to a `target` register sets the k implemented bits of EIID equal to the least-significant k bits of the 32-bit value written.

4.6. Reset

Upon reset of an APLIC, all its state becomes valid and consistent but otherwise UNSPECIFIED, except for:

- the domaincfg register of each interrupt domain (Section 4.5.1);
- possibly the MSI address configuration registers of machine-level interrupt domains (Section 4.5.3 and Section 4.5.4); and
- the Busy bit of each interrupt domain's `genmsi` register, if it exists (Section 4.5.15).

4.7. Precise effects on interrupt-pending bits

An attempt to set or clear an interrupt source's pending bit by writing to a register in the interrupt domain's control region may or may not be successful, depending on the corresponding source mode, the interrupt domain's delivery mode, and the state of the source's rectified input value (defined in Section 4.5.2). The following enumerates all the circumstances when a pending bit is set or cleared for a given source mode.

If the source mode is Detached:

- The pending bit is set to one only by a relevant write to a `setip` or `setipnum` register.
- The pending bit is cleared when the interrupt is claimed at the APLIC or forwarded by MSI, or by a relevant write to an `in_clrip` register or to `clripnum`.

If the source mode is Edge1 or Edge0:

- The pending bit is set to one by a low-to-high transition in the rectified input value, or by a relevant write to a `setip` or `setipnum` register.
- The pending bit is cleared when the interrupt is claimed at the APLIC or forwarded by MSI, or by a relevant write to an `in_clrip` register or to `clripnum`.

If the source mode is Level1 or Level0 and the interrupt domain is configured in direct delivery mode (`domaincfg`.DM = 0):

- The pending bit is set to one whenever the rectified input value is high. The pending bit cannot be set by a write to a `setip` or `setipnum` register.
- The pending bit is cleared whenever the rectified input value is low. The pending bit is not cleared by a claim of the interrupt at the APLIC, nor can it be cleared by a write to an `in_clrip` register or to `clripnum`.

If the source mode is Level1 or Level0 and the interrupt domain is configured in MSI delivery mode (`domaincfg`.DM = 1):

- The pending bit is set to one by a low-to-high transition in the rectified input value. The pending bit may also be set by a relevant write to a `setip` or `setipnum` register when the rectified input value is high, but not when the rectified input value is low.
- The pending bit is cleared whenever the rectified input value is low, when the interrupt is forwarded by MSI, or by a relevant write to an `in_clrip` register or to `clripnum`.

When an interrupt domain is in direct delivery mode, the pending bit for a level-sensitive source is always just a copy of the rectified input value. Even in MSI delivery mode, the pending bit for a level-sensitive source is never set (= 1) when the rectified input value is low.

In addition to the rules above, a write to a `sourcecfg` register can cause the source's interrupt-pending bit to be set to one, as specified in Section 4.5.2.

4.8. Interrupt delivery directly by the APLIC

When an interrupt domain is in direct delivery mode (`domaincfg`.DM = 0), interrupts are delivered from the APLIC to harts by a unique signal to each hart, usually a dedicated wire. In this case, the

domain's memory-mapped control region contains at the end an array of interrupt delivery control (IDC) structures, one IDC structure per potential hart index. The first IDC structure is for the domain's hart with index 0; the second is for the hart with index 1; etc.

4.8.1. Interrupt delivery control (IDC) structure

Each IDC structure is 32 bytes (naturally aligned to a 32-byte address boundary) and has these defined registers:

offset	size	register name
`0x00`	4 bytes	`idelivery`
`ox04`	4 bytes	`iforce`
`0x08`	4 bytes	`ithreshold`
`0x18`	4 bytes	`topi`
`0x1C`	4 bytes	`claimi`

If the IDC structure is for a hart index number that is not valid for any actual hart in the interrupt domain, then these registers may optionally be all read-only zeros. Otherwise, the registers are documented individually below.

A particular APLIC might be built to support up to some maximum number of harts without complete knowledge of the set of hart index numbers the system will employ in each interrupt domain. In that case, for the hart index numbers that are unused, the APLIC may have IDC structures that are functional within the APLIC (not read-only zeros) but simply left unconnected to any physical harts.

4.8.1.1. Interrupt delivery enable (`idelivery`)

`idelivery` is a **WARL** register that controls whether interrupts that are targeted to the corresponding hart are delivered to the hart so they appear as a pending interrupt in the hart's `mip` CSR. Only two values are currently defined for `idelivery`:

0 = interrupt delivery is disabled

1 = interrupt delivery is enabled

The `idelivery` register affects only whether interrupts are delivered to the relevant hart. It has no effect on any other APLIC state, including IDC registers `topi` and `claimi`.

If an IDC structure is for a nonexistent hart (i.e., corresponding to a hart index number that is not valid for any actual hart in the interrupt domain), setting `idelivery` to 1 does not deliver interrupts to any hart.

4.8.1.2. Interrupt force (`iforce`)

`iforce` is a **WARL** register useful for testing. Only values 0 and 1 are allowed. Setting `iforce` = 1 forces an interrupt to be asserted to the corresponding hart whenever both the IE field of `domaincfg` is one and interrupt delivery is enabled to the hart by the `idelivery` register. When `topi` is zero, this creates a *spurious external interrupt* for the hart.

When a read of register `claimi` returns an interrupt identity of zero (indicating a spurious interrupt),

iforce is automatically cleared to zero.

4.8.1.3. Interrupt enable threshold (ithreshold)

ithreshold is a WLRL register that determines the minimum interrupt priority (maximum priority number) for an interrupt to be signaled to the corresponding hart. Register **ithreshold** implements exactly IPRIOLEN bits, and thus is capable of holding all priority numbers from 0 to $2^{\text{IPRIOLEN}} - 1$.

When **ithreshold** is a nonzero value P, interrupt sources with priority numbers P and higher do not contribute to signaling interrupts to the hart, as though those sources were not enabled, regardless of the settings of their interrupt-enable bits. When **ithreshold** is zero, all enabled interrupt sources can contribute to signaling interrupts to the hart.

4.8.1.4. Top interrupt (topi)

topi is a read-only register whose value indicates the current highest-priority pending-and-enabled interrupt targeted to this hart that also exceeds the priority threshold specified by **ithreshold**, if not zero.

A read of **topi** returns zero either if no interrupt that is targeted to this hart is both pending and enabled, or if **ithreshold** is not zero and no pending-and-enabled interrupt targeted to this hart has a priority number less than the value of **ithreshold**. Otherwise, the value returned from a read of **topi** has this format:

bits 25:16 Interrupt identity (source number)

bits 7:0 Interrupt priority

All other bit positions are zeros.

The interrupt identity reported in **topi** is the minor identity for an external interrupt at the target hart.

The value of **topi** is not affected by **domaincfg**.IE or by **idelivery**.

Writes to **topi** are ignored.

4.8.1.5. Claim top interrupt (claimi)

Register **claimi** has the same value as **topi**. When this value is not zero, reading **claimi** has the simultaneous side effect of clearing the pending bit for the reported interrupt identity, if possible. See Section 4.7 for exactly when the pending bit is cleared by a read of **claimi**.

A read from **claimi** that returns a value of zero has the simultaneous side effect of setting the **iforce** register to zero.

Writes to **claimi** are ignored.

4.8.2. Interrupt delivery and handling

When an interrupt domain is configured so the APLIC delivers interrupts directly to harts (field DM of **domaincfg** is zero), the APLIC supplies the *external interrupt* signals, at the domain's privilege level, for all harts of the domain, so long as one of the following is true: (a) the harts do not have IMSICs, or (b) the **eidelivery** registers of the relevant IMSIC interrupt files are set to **0x40000000** (Section 3.8.1). For

a machine-level domain, the interrupt signals from the APLIC appear as bit MEIP (Machine External Interrupt-Pending) in each hart's **mip** CSR. For a supervisor-level domain, the interrupt signals appear as bit SEIP (Supervisor External Interrupt-Pending) in each hart's **mip** and **sip** CSRs. Each interrupt signal may be arbitrarily delayed traveling from the APLIC to the proper hart.

At the APLIC, each interrupt signal to a hart is derived from the IE field of register **domaincfg** and the current state of the hart's IDC structure in the memory-mapped control region for the domain. If either **domaincfg**.IE = 0 or interrupt delivery to the hart is disabled by the **idelivery** register (**idelivery** = 0), the interrupt signal is held de-asserted. When **domaincfg**.IE = 1 and interrupt delivery is enabled (**idelivery** = 1), the interrupt signal is asserted whenever either register **iforce** or **topi** is not zero.

Due to likely delay in the communication between an APLIC and a hart, it may happen that an external interrupt trap is taken, yet no interrupt is pending and enabled for the hart when a read of the hart's **claimi** register actually occurs. In such a circumstance, the interrupt identity reported by the claim will be zero, resulting in an apparent *spurious interrupt* from the APLIC. Portable software must be prepared for the possibility of spurious interrupts at the APLIC, which can safely be ignored and should be rare. For testing purposes, a spurious interrupt can be triggered for a hart by setting an IDC structure's **iforce** register to 1.

A trap handler solely for external interrupts via an APLIC could be written roughly as follows:

> save processor registers
>
> i = read register **claimi** from the hart's IDC structure at the APLIC
>
> i = i>>16
>
> call the interrupt handler for external interrupt *i* (minor identity)
>
> restore processor registers
>
> return from trap

To account for spurious interrupts, this pseudocode assumes there is an interrupt handler for "external interrupt 0" which does nothing.

4.9. Interrupt forwarding by MSIs

In MSI delivery mode (**domaincfg**.DM = 1), an interrupt domain forwards interrupts to target harts by MSIs.

An MSI is sent for a specific source only when the source's corresponding pending and enable bits are both one and the IE field of register **domaincfg** is also one. If and when an MSI is sent, the source's interrupt pending bit is cleared.

4.9.1. Addresses and data for outgoing MSIs

To forward interrupts by MSIs, an APLIC must know the MSI target address for each hart. For any given system, these addresses are fixed and should be hardwired into the APLIC if possible. However, some APLIC implementations may require that software supply the MSI target addresses. In that case, the root domain's registers **mmsiaddrcfg**, **mmsiaddrcfgh**, **smsiaddrcfg**, and **smsiaddrcfgh** (Section 4.5.3 and Section 4.5.4) may be used to configure the MSI addresses for all interrupt domains. Alternatively MSI addresses may be configured by some custom means outside this standard. If MSI

target addresses must be configured by software, this should be done only from a suitably privileged execution mode, typically just once, early after system reset.

For a machine-level interrupt domain, if MSI target addresses are determined by **mmsiaddrcfg** and **mmsiaddrcfgh**, then the address for an outgoing MSI for interrupt source i is computed from those registers and from the Hart Index field of register **target[i]** as follows:

$$g = (\text{Hart Index}>>\text{LHXW}) \;\&\; (2^{\text{HHXW}} - 1)$$

$$h = \text{Hart Index} \;\&\; (2^{\text{LHXW}} - 1)$$

$$\text{MSI address} = (\text{Base PPN} \mid (g<<(\text{HHXS}+12)) \mid (h<<\text{LHXS}))<<12$$

Here, $<<k$ and $>>k$ represent shifting left and right by k bits, an ampersand (&) represents bitwise logical AND, and a vertical bar (|) represents bitwise logical OR. Assuming the recommendations of Section 3.6 are followed for the arrangement of IMSIC interrupt files in the machine's address space, value g is intended to be the number of a hart group (always zero if HHXW = 0), while h is the number of the target hart within that group. Represented in the terms of Section 3.6, HHXW = j, LHXW = k, HHXS = $E-24$, LHXS = $C-12$, and Base PPN = A>>12.

For a supervisor-level domain, if MSI target addresses are determined by the root domain's configuration registers (**smsiaddrcfg** and others), then to construct the address for an outgoing MSI for interrupt source i, the Hart Index from register **target[i]** must first be converted into the index number that machine-level domains use for the same hart. (These numbers are often the same, but they may not be.) The address for the MSI is then computed using this machine-level hart index together with the Base PPN and LHXS values from **smsiaddrcfg** and **smsiaddrcfgh**, the other fields (HHXW, LHXW, and HHXS) from **mmsiaddrcfgh**, and the Guest Index from **target[i]**, as follows:

$$g = (\text{machine-level hart index}>>\text{LHXW}) \;\&\; (2^{\text{HHXW}} - 1)$$

$$h = \text{machine-level hart index} \;\&\; (2^{\text{LHXW}} - 1)$$

$$\text{MSI address} = (\text{Base PPN} \mid (g<<(\text{HHXS} + 12)) \mid (h<<\text{LHXS}) \mid \text{Guest Index})<<12$$

Represented in the terms of Section 3.6, HHXW = j, LHXW = k, HHXS = $E-24$, LHXS = $D-12$, and Base PPN = B>>12.

The data for an outgoing MSI write is taken from the EIID field of **target[i]**, zero-extended to 32 bits. An MSI's 32-bit data is always written in little-endian byte order, regardless of the BE field of the domain's **domaincfg** register.

4.9.2. Special consideration for level-sensitive interrupt sources

As soon as a level-sensitive interrupt is forwarded by MSI, the APLIC clears the pending bit for the interrupt source and then ignores the source until its incoming signal has been de-asserted. Clearing the pending bit when an MSI is sent is obviously necessary to avoid a constant stream of repeated MSIs from the APLIC to the target hart for the same interrupt. However, after an interrupt service routine has addressed a cause found for the interrupt, the incoming interrupt wire might remain asserted at the APLIC for another reason, despite that the interrupt's pending bit at the APLIC was cleared and will remain so without intervention from software. If the interrupt service routine then exits without further action, a continued interrupt from this source might never receive attention.

To avoid dropping interrupts in this way, the interrupt service routine for a level-sensitive interrupt may do one of the following before exiting:

The first option is to test whether the interrupt wire into the APLIC is still asserted, by reading the appropriate `in_clrip` register at the APLIC. If the incoming interrupt is still asserted, the body of the interrupt service routine may be repeated to find and address an additional interrupt cause before the source wire is tested again. Once the incoming wire is observed not asserted, the interrupt service routine may safely exit, as any new interrupt assertion will cause the pending bit to become set and a new MSI sent to the hart.

A second option is for the interrupt service routine to write the APLIC's source identity number for the interrupt to the domain's `setipnum` register just before exiting. This will cause the interrupt's pending bit to be set to one again if the source is still asserting an interrupt, but not if the source is not asserting an interrupt.

4.9.3. Synchronizing interactions between a hart and the APLIC

When an APLIC sends an MSI to a hart, there is an unspecified travel delay before the MSI is observed at the hart's IMSIC. Consequently, after an APLIC's configuration is changed by writing to an APLIC register, harts may continue to see MSIs arrive from the APLIC from the time before the write, for an unspecified amount of time.

It is sometimes necessary to know when no more of these late MSIs can arrive. For example, if a hart will be turned off ("powered down"), all interrupts directed to it must be redirected to other harts, which may involve reconfiguring one or more APLICs. Even after the APLICs are reconfigured, the hart still cannot be safely turned off until it is known no more MSIs are destined for it.

The `genmsi` register (Section 4.5.15) exists to allow software to determine when all earlier MSIs have arrived at a hart. To use `genmsi` for this purpose, software can dedicate one external interrupt identity at each hart's IMSIC interrupt file solely for APLIC synchronization. Assuming there are multiple harts, an APLIC's `genmsi` register should also be protected by a standard mutual-exclusion lock. The following sequence can then be used to synchronize between an APLIC and a specific hart:

1. At the hart's IMSIC, clear the pending bit for the specific minor interrupt identity i used exclusively for APLIC synchronization.
2. Acquire the shared lock for the APLIC's `genmsi` register.
3. Write `genmsi` to generate an MSI to the hart with interrupt identity i.
4. Repeatedly read `genmsi` until bit Busy is false.
5. Release the lock for `genmsi`.
6. Repeatedly read the pending bit for minor interrupt identity i at the hart's IMSIC until it is found set.

The loops of steps 4 and 6 are expected normally to succeed very quickly, often on the first or second attempt. When this sequence is complete, all earlier MSIs from the APLIC must also have arrived at the hart's IMSIC.

Chapter 5. Interrupts for Machine and Supervisor Levels

The RISC-V Privileged Architecture defines several major identities in the range 0-15 for interrupts at a hart, including machine-level and supervisor-level external interrupts (numbers 11 and 9), machine- and supervisor-level timer interrupts (7 and 5), and machine- and supervisor-level software interrupts (3 and 1). Beyond these major labels, the *external* interrupts at each privilege level are given secondary, minor identities by an external interrupt controller such as an APLIC or IMSIC, distinguishing interrupts from different devices or causes. These minor identities for external interrupts were covered in Chapter 3 and Chapter 4 specifying the IMSIC and APLIC components.

The Advanced Interrupt Architecture reserves another 24 major interrupt identities for additional *local interrupts* that arise within or in close proximity to the hart, often for reporting errors. A mechanism is also defined that allows software to selectively delegate both local and custom interrupts to the next less-privileged level, or in some cases to inject entirely virtual interrupts into a less-privileged level.

Lastly, an optional facility lets software assign priorities to major interrupts (such as the timer and software interrupts, and any local interrupts) such that they may mix with the priorities set for external interrupts by a PLIC, APLIC, or IMSIC.

5.1. Defined major interrupts and default priorities

Table 8 lists all the major interrupts currently defined for RISC-V harts that conform to this Advanced Interrupt Architecture (AIA). Besides the major interrupts specified by the RISC-V Privileged Architecture, the AIA adds interrupt numbers 35 and 43 as local interrupts for low- and high-priority *RAS events*.

Table 8. The standard major interrupt codes, listed in default priority order

Default priority order	Major interrupt numbers	Description
Highest	43	Local interrupt: high-priority RAS event
	11, 3, 7 9, 1, 5 12 10, 2, 6 13	Machine interrupts: external, software, timer Supervisor interrupts: external, software, timer Supervisor guest external interrupt VS interrupts: external, software, timer Local interrupt: counter overflow
Lowest	35	Local interrupt: low-priority RAS event

The default priority order in Table 8 is applicable only when multiple major interrupts would trap to the same privilege mode. Interrupt traps to a more-privileged mode always have priority over traps to a less-privileged mode.

Table 9. Categorization of current and future major interrupts.

Major interrupt numbers	Category	
0-12 13-15	Not Local interrupts Local interrupts	Assigned by the Privileged Architecture
16-23 24-31 32-47 ≥ 48	Local interrupts *Designated for custom use* Local interrupts *Designated for custom use*	

Of the major interrupts controlled by the base Privileged Architecture (numbers 0-15), the AIA categorizes the counter overflow interrupt (code 13) as a *local interrupt*. It is assumed furthermore that any future definitions for reserved interrupt numbers 14 and 15 will also be local interrupts. Besides the two RAS interrupts, the AIA additionally reserves major interrupt numbers in the ranges 16-23 and 32-47 for standard local interrupts that other RISC-V extensions may define. The remaining major interrupts allocated to the Privileged Architecture, numbers 0-12, are categorized as not local interrupts. Taken altogether, Table 9 summarizes the AIA's categorization of all major interrupt identities.

RAS is an abbreviation for *Reliability, Availability, and Serviceability*. Typically a RAS event corresponds to the detection of corrupted data (e.g. as a result of a soft or hard error) and/or the use of such data. The high-priority RAS event local interrupt may, for example, signal an occurrence of an urgent uncorrected error that needs action from a RAS error handler to contain the error and, if possible, to recover from it. The low-priority RAS event local interrupt may, for example, be triggered by non-urgent deferred or corrected errors.

The AIA does not itself require that detected RAS events trigger one of the two local interrupts defined for this purpose. Systems are free to report any or all RAS events another way, such as by external interrupts routed through an APLIC or IMSIC, or by custom interrupts.

In all likelihood, the method for reporting a particular RAS event will depend on where in the system the event is detected. The AIA defines local interrupt numbers for RAS events so systems have a standard way to report such events when detected locally at a hart, without depending solely on external or custom interrupts.

As always, platform standards may further constrain how a system reports events, whether RAS events or other.

For the standard local interrupts not defined by the base RISC-V Privileged Architecture (numbers 16-23 and 32-47), the current plan is to assign default priorities in the order shown in this table:

Highest	*47, 23, 46, 45, 22, 44, 43, 21, 42, 41, 20, 40*	
	11, 3, 7 *9, 1, 5* *12* *10, 2, 6* *13*	*Machine interrupts: external, software, timer* *Supervisor interrupts: external, software, timer* *Supervisor guest external interrupt* *VS interrupts: external, software, timer* *Counter overflow interrupt*
Lowest	*39, 19, 38, 37, 18, 36, 35, 17, 34, 33, 16, 32*	

Among interrupts 16-23, a higher interrupt number conveys higher default priority, and likewise for interrupts 32-47. These two groups are interleaved together in the complete order, and the Privileged Architecture's standard interrupts, 0-15, are inserted into the middle of the sequence. This proposed default priority order is arranged so that interrupts 0-31 can potentially be an adequate subset on their own for 32-bit RISC-V systems.

In actuality, future RISC-V extensions may or may not stick to this plan for the default priority order of interrupts they define.

In addition to the existing major interrupts of Table 8, the following local interrupts are tentatively proposed, listed in order of decreasing default priority:

23 Bus or system error

45 Per-core high-power or over-temperature event

17 Debug/trace interrupt

These local interrupts are expected to be specified by other RISC-V extensions. Be aware, this list is not final and may change as the relevant extensions are developed and ratified.

If a future version of the RISC-V Privileged Architecture defines interrupt 0, the Advanced Interrupt Architecture needs it to have a default priority lower than certain external interrupts. See Section 5.2.2 and Section 5.4.2 on CSRs `mtopi` *and* `stopi`.

Interrupt numbers 24-31 and 48 and higher are all designated for custom use. If a hart implements any custom interrupts, their positions in the default priority order must be documented for the hart.

While many of the standard registers such as `mip` *and* `mie` *have space for major interrupts only in the range 0-63, custom interrupts with numbers 64 and above are conceivable with added custom support. CSRs* `mtopi` *(Section 5.2.2) and* `stopi` *(Section 5.4.2) allow for major interrupt numbers potentially as large as 4095.*

When a hart supports the arbitrary configuration of interrupt priorities by software (described in later sections), the default priority order still remains relevant for breaking ties when two interrupt sources are assigned the same priority number.

5.2. Interrupts at machine level

For whichever standard local interrupts are implemented, the corresponding bits in CSRs `mip` and `mie`

must be writable, and the corresponding bits in **mideleg** (if that CSR exists because supervisor mode is implemented) must each either be writable or be hardwired to zero. An occurrence of a local interrupt event causes the interrupt-pending bit in **mip** to be set to one. This bit then remains set until cleared by software.

As established by the base RISC-V Privileged Architecture, an interrupt traps to M-mode whenever all of the following are true: (a) either the current privilege mode is M-mode and machine-level interrupts are enabled by the MIE bit of **mstatus**, or the current privilege mode has less privilege than M-mode; (b) matching bits in **mip** and **mie** are both one; and (c) if **mideleg** exists, the corresponding bit in **mideleg** is zero.

When multiple interrupt causes are ready to trigger simultaneously, the interrupt taken first is determined by priority order, which may be the default order specified in the previous section Section 5.1, or may be a modified order configured by software.

5.2.1. Configuring priorities of major interrupts at machine level

The machine-level priorities for major interrupts 0-63 may be configured by a set of registers accessed through the **miselect** and **mireg** CSRs introduced in Chapter 2. When XLEN = 32, sixteen of these registers are defined, listed below with their **miselect** addresses:

```
0x30 iprio0
0x31 iprio1
...  ...
0x3F iprio15
```

Each register controls the priorities of four interrupts, with one 8-bit byte per interrupt. For a number k in the range 0-15, register `iprio`k controls the priorities of interrupts $k \times 4$ through $k \times 4 + 3$, formatted as follows:

```
bits 7:0   Priority number for interrupt k × 4
bits 15:8  Priority number for interrupt k × 4 + 1
bits 23:16 Priority number for interrupt k × 4 + 2
bits 31:24 Priority number for interrupt k × 4 + 3
```

When XLEN = 64, only the even-numbered registers exist:

```
0x30 iprio0
0x32 iprio2
 ...    ...
0x3E iprio14
```

Each register controls the priorities of eight interrupts. For even k in the range 0-14, register `iprio`k controls the priorities of interrupts $k \times 4$ through $k \times 4 + 7$, formatted as follows:

```
bits 7:0   Priority number for interrupt k × 4
bits 15:8  Priority number for interrupt k × 4 + 1
```

bits 23:16 Priority number for interrupt $k \times 4 + 2$

bits 31:24 Priority number for interrupt $k \times 4 + 3$

bits 39:32 Priority number for interrupt $k \times 4 + 4$

bits 47:40 Priority number for interrupt $k \times 4 + 5$

bits 55:48 Priority number for interrupt $k \times 4 + 6$

bits 63:56 Priority number for interrupt $k \times 4 + 7$

When XLEN = 64 and **miselect** is an odd value in the range **0x31**-**0x3F**, attempting to access **mireg** raises an illegal instruction exception.

The valid registers **iprio0**-**iprio15** are known collectively as the iprio *array* for machine level.

The width of priority numbers for external interrupts is *IPRIOLEN*. This parameter is affected by the main external interrupt controller for the hart, whether a PLIC, APLIC, or IMSIC.

For an APLIC, IPRIOLEN is in the range 1-8 as specified in Chapter 4 on the APLIC.

For an IMSIC, IPRIOLEN is 6, 7, or 8. IPRIOLEN may be 6 only if the number of external interrupt identities implemented by the IMSIC is 63. IPRIOLEN may be 7 only if the number of external interrupt identities implemented by the IMSIC is no more than 127. IPRIOLEN may be 8 for any IMSIC, regardless of the number of external interrupt identities implemented.

Each byte of a valid **iprio**k register is either a read-only zero or a **WARL** unsigned integer field implementing exactly IPRIOLEN bits. For a given interrupt number, if the corresponding bit in **mie** is read-only zero, then the interrupt's priority number in the **iprio** array must be read-only zero as well. The priority number for a machine-level external interrupt (bits 31:24 of register **iprio2**) must also be read-only zero. Aside from these two restrictions, implementations may freely choose which priority number fields are settable and which are read-only zeros. If all bytes in the **iprio** array are read-only zeros, priorities can be configured only for external interrupts, not for any other interrupts.

Platform standards may require that priorities be configurable for certain interrupt causes.

The **iprio** array accessed via **miselect** and **mireg** affects the prioritization of interrupts only when they trap to M-mode. When an interrupt's priority number in the array is zero (either read-only zero or set to zero), its priority is the default order from Section 5.1. Setting an interrupt's priority number instead to a nonzero value p gives that interrupt nominally the same priority as a machine-level external interrupt with priority number p. For a major interrupt that defaults to a higher priority than machine external interrupts, setting its priority number to a nonzero value *lowers* its priority. For a major interrupt that defaults to a lower priority than machine external interrupts, setting its priority number to a nonzero value *raises* its priority. When two interrupt causes have been assigned the same nominal priority, ties are broken by the default priority order. Table 10 summarizes the effect of priority numbers on interrupt priority.

When a hart has an IMSIC supporting more than 255 minor identities for external interrupts, the only non-default priorities that can be configured for other interrupts are those corresponding to external interrupt identities 1-255, not those of identities 256 or higher.

Table 10. Effect of the machine-level iprio array on the priorities of interrupts taken in M-mode. For interrupts with the same priority number, the default order of Section 5.1 prevails.

	Interrupts with default priority above machine external interrupts	Machine external interrupts	Interrupts with default priority below machine external interrupts
Priority order	Priority number in machine-level `iprio` array	Priority number from interrupt controller (APLIC or IMSIC)	Priority number in machine-level `iprio` array
Highest	0		
	1 2 ... 254 255	1 2 ... 254 255	1 2 ... 254 255
		256 and above (IMSIC only)	
Lowest			0

Implementing the priority configurability of this section requires that a RISC-V hart's external interrupt controller communicate to the hart not only the existence of a pending-and-enabled external interrupt but also the interrupt's priority number. Typically this implies that the width of the connection for signaling an external interrupt to the hart is not just a single wire as usual but now $IPRIOLEN+1$ *wires.*

It is expected that many systems will forego priority configurability of major interrupts and simply have the array be all read-only zeros. Systems that need this priority configurability can try to arrange for each hart's external interrupt controller to be relatively close to the hart, by, for example, limiting the system to at most a few small cores connected to an APLIC, or alternatively by giving every hart its own IMSIC.

If supported, setting the priority number for supervisor-level external interrupts (bits 15:8 of `iprio2`) to a nonzero value p has the effect of giving the entire category of supervisor external interrupts nominally the same priority as a machine external interrupt with priority number p. But note that this applies only to the case when supervisor external interrupts trap to M-mode.

(Because supervisor guest external interrupts and VS-level external interrupts are required to be delegated to supervisor level when the H extension is implemented, the machine-level priority numbers for these interrupts are always ignored and should be read-only zeros.)

If the system has an original PLIC for backward compatibility with older software, reset should initialize the machine-level `iprio` array to all zeros.

5.2.2. Machine top interrupt CSR (`mtopi`)

Machine-level CSR `mtopi` is read-only with width MXLEN. A read of `mtopi` returns information about the highest-priority pending-and-enabled interrupt for machine level, in this format:

```
bits 27:16 IID
bits 7:0   IPRIO
```

All other bits of `mtopi` are reserved and read as zeros.

The value of `mtopi` is zero unless there is an interrupt pending in `mip` and enabled in `mie` that is not delegated to a less-privileged level. When there is a pending-and-enabled major interrupt for machine level, field IID (Interrupt Identity) is the major identity number of the highest-priority interrupt, and field IPRIO indicates its priority.

If all bytes of the machine-level `iprio` array are read-only zeros, a simplified implementation of field IPRIO is allowed in which its value is always 1 whenever `mtopi` is not zero.

Otherwise, when `mtopi` is not zero, if the priority number for the reported interrupt is in the range 1 to 255, IPRIO is simply that number. If the interrupt's priority number is zero or greater than 255, IPRIO is set to either 0 or 255 as follows:

- If the interrupt's priority number is greater than 255, then IPRIO is 255 (lowest representable priority).
- If the interrupt's priority number is zero and interrupt number IID has a default priority higher than a machine external interrupt, then IPRIO is 0 (highest priority).
- If the interrupt's priority number is zero and interrupt number IID has a default priority lower than a machine external interrupt, then IPRIO is 255 (lowest representable priority).

To ensure that `mtopi` is never zero when an interrupt is pending and enabled for machine level, if major interrupt 0 can trap to M-mode, it must have a default priority lower than a machine external interrupt.

The value of `mtopi` is not affected by the global interrupt enable MIE in CSR `mstatus`.

The RISC-V Privileged Architecture ensures that, when the value of `mtopi` is not zero, a trap is taken to M-mode for the interrupt indicated by field IID if either the current privilege mode is M and `mstatus`.MIE is one, or the current privilege mode has less privilege than M-mode. The trap itself does not cause the value of `mtopi` to change.

The following pseudocode shows how a machine-level trap handler might read `mtopi` to avoid redundant restoring and saving of processor registers when an interrupt arrives during the handling of another trap (either a synchronous exception or an earlier interrupt):

```
save processor registers
i = read CSR mcause
if (i >= 0) {
    handle synchronous exception i
    restore mstatus if necessary
}
if (mstatus.MPIE == 1) {
    loop {
        i = read CSR mtopi
        if (i == 0) exit loop
        i = i>>16
        call the interrupt handler for major interrupt i
    }
}
restore processor registers
return from trap
```

(This example can be further optimized, but with an increase in complexity.)

In order for this algorithm to function correctly, `mstatus`.MPIE must be set to 1 before executing an MRET that changes the privilege mode.

Assuming `mstatus` *is saved and restored by trap handlers at entry and exit as is common, it is sufficient to set* `mstatus`*.MPIE = 1 only once, before the first use of MRET that changes privilege mode. After an MRET, a trap back to M-mode will restore* `mstatus`*.MPIE = 1; and if the trap handler preserves* `mstatus`*, it will still be true before the next MRET that ends the handler.*

5.3. Interrupt filtering and virtual interrupts for supervisor level

When supervisor mode is implemented, the Advanced Interrupt Architecture adds a facility for software filtering of interrupts and for virtual interrupts, making use of new CSRs `mvien` (Machine Virtual Interrupt Enables) and `mvip` (Machine Virtual Interrupt-Pending bits). *Interrupt filtering* permits a supervisor-level interrupt (SEI or SSI) or local or custom interrupt to trap to M-mode and then be selectively delegated by software to supervisor level, even while the corresponding bit in `mideleg` remains zero. The same hardware may also, under the right circumstances, allow machine level to assert *virtual interrupts* to supervisor level that have no connection to any real interrupt events.

Just as with CSRs `mip`, `mie`, and `mideleg`, each bit of registers `mvien` and `mvip` corresponds with an interrupt number in the range 0-63. When a bit in `mideleg` is zero and the matching bit in `mvien` is one, then the same bit position in `sip` is an alias for the corresponding bit in `mvip`. A bit in `sip` is read-only zero when the corresponding bits in `mideleg` and `mvien` are both zero. The combined effects of `mideleg` and `mvien` on `sip` and `sie` are summarized in Table 11.

Table 11. The effects of `mideleg` *and* `mvien` *on* `sip` *and* `sie` *(except for the H extension's VS-level interrupts, which appear in* `hip` *and* `hie` *instead of* `sip` *and* `sie`*). A bit in* `mvien` *can be set to 1 only for major interrupts 1, 9, and 13-63. For interrupts 0-12, some aliases of* `mip` *bits in* `sip` *may be read-only copies, as specified by the base Privileged Architecture.*

mideleg[n]	mvien[n]	sip[n]	sie[n]
0	0	Read-only 0	Read-only 0
0	1	Alias of `mvip`[n]	Writable
1	-	Alias of `mip`[n]	Alias of `mie`[n]

The name of CSR `mvien` *is not* `"mvie"` *because the function of this register is more analogous to* `mcounteren` *than to* `mie`*. The bits of* `mvien` *control whether the virtual interrupt-pending bits in register* `mvip` *are active and visible at supervisor level. This is different than how the usual interrupt-enable bits (such as in* `mie`*) mask pending interrupts.*

A bit in `sie` is writable if and only if the corresponding bit is set in either `mideleg` or `mvien`. When an interrupt is delegated by `mideleg`, the writable bit in `sie` is an alias of the corresponding bit in `mie`; else it is an independent writable bit. As usual, bits that are not writable in `sie` must be read-only zeros.

If a bit of `mideleg` is zero and the corresponding bit in `mvien` is changed from zero to one, then the value of the matching bit in `sie` becomes UNSPECIFIED. Likewise, if a bit of `mvien` is one and the corresponding bit in `mideleg` is changed from one to zero, the value of the matching bit in `sie` again becomes UNSPECIFIED.

For interrupt numbers 13-63, implementations may freely choose which bits of `mvien` are writable and which bits are read-only zero or one. If such a bit in `mvien` is read-only zero (preventing the virtual

interrupt from being enabled), the same bit should be read-only zero in **mvip**. All other bits for interrupts 13-63 must be writable in **mvip**.

Platform standards or other extensions may require that bits of **mvien** *for certain interrupt causes be writable, or be read-only zero or one.*

The bits of **mvien** for supervisor software interrupts (code 1) and supervisor external interrupts (code 9) are each either writable or read-only zero; they cannot be read-only ones. The other bits of **mvien** for interrupts 0-12 are reserved and must be read-only zeros.

It is strongly recommended that bit 9 of **mvien** be writable. Furthermore, if bit 1 (SSIP) of **mip** can be set automatically by an interrupt controller and not just by explicit writes to **mip** or **sip**, it is strongly recommended that bit 1 of **mvien** also be writable.

When bit 1 of **mvien** is zero, bit 1 of **mvip** is an alias of the same bit (SSIP) of **mip**. But when bit 1 of **mvien** is one, bit 1 of **mvip** is a separate writable bit independent of **mip**.SSIP. When the value of bit 1 of **mvien** is changed from zero to one, the value of bit 1 of **mvip** becomes UNSPECIFIED.

Bit 5 of **mvip** is an alias of the same bit (STIP) in **mip** when that bit is writable in **mip**. When STIP is not writable in **mip** (such as when **menvcfg**.STCE = 1), bit 5 of **mvip** is read-only zero.

When bit 9 of **mvien** is zero, bit 9 of **mvip** is an alias of the software-writable bit 9 of **mip** (SEIP). But when bit 9 of **mvien** is one, bit 9 of **mvip** is a writable bit independent of **mip**.SEIP. Unlike for bit 1, changing the value of bit 9 of **mvien** does not affect the value of bit 9 of **mvip**.

The base Privileged Architecture specifies unusual read/write behavior for what it calls the software-writable SEIP bit of register **mip**. *When bit 9 of* **mvien** *is zero, bit 9 of* **mvip** *makes the software-writable SEIP bit of* **mip** *directly accessible by itself. Furthermore, as explained below, setting bit 9 of* **mvien** *to one separates the software-writable SEIP bit from* **mip** *entirely, so it is then just a writable bit in* **mvip**.

Except for bits 1, 5, and 9 as specified above, the bits of **mvip** in the range 12:0 are reserved and must be read-only zeros.

The value of bit 9 of **mvien** has some additional consequences for supervisor external interrupts:

- When bit 9 of **mvien** is zero, the software-writable SEIP bit (bit 9 of **mvip**) interacts with reads and writes of **mip** in the way specified by the base RISC-V Privileged Architecture. In particular, for most purposes, the value of bit 9 of **mvip** is logically ORed into the readable value of **mip**.SEIP. But when bit 9 of **mvien** is one, bit SEIP in **mip** is read-only and does not include the value of bit 9 of **mvip**. Rather, the value of **mip**.SEIP is simply the supervisor external interrupt signal from the hart's external interrupt controller (APLIC or IMSIC).
- If the hart has an IMSIC, then when bit 9 of **mvien** is one, attempts from S-mode to explicitly access the supervisor-level interrupt file raise an illegal instruction exception. The exception is raised for attempts to access CSR **stopei**, or to access **sireg** when **siselect** has a value in the range **0x70**-**0xFF**. Accesses to guest interrupt files (through **vstopei** or **viselect**+**vsireg**) are not affected.

When the H extension is implemented, if a bit is zero in the same position in both **mideleg** and **mvien**, then that bit is read-only zero in **hideleg** (in addition to being read-only zero in **sip**, **sie**, **hip**, and **hie**). But if a bit for one of interrupts 13-63 is a one in either **mideleg** or **mvien**, then the same bit in **hideleg** may be writable or may be read-only zero, depending on the implementation. No bits in **hideleg** are ever read-only ones. The H extension further constrains bits 12:0 of **hideleg**.

When supervisor mode is implemented, the minimal required implementation of **mvien** and **mvip** has all bits being read-only zeros except for **mvip** bits 1 and 9, and sometimes bit 5, each of which is an alias of an existing writable bit in **mip**. (Although, as noted, it is strongly recommended that bit 9 of **mvien** also be writable.) When supervisor mode is not implemented, registers **mvien** and **mvip** do not exist.

5.4. Interrupts at supervisor level

If a standard local interrupt becomes pending (= 1) in **sip**, the bit in **sip** is writable and will remain set until cleared by software.

Just as for machine level, the taking of interrupt traps at supervisor level remains essentially the same as specified by the base RISC-V Privileged Architecture. An interrupt traps into S-mode (or HS-mode) whenever all of the following are true: (a) either the current privilege mode is S-mode and supervisor-level interrupts are enabled by the SIE bit of **sstatus**, or the current privilege mode has less privilege than S-mode; (b) matching bits in **sip** and **sie** are both one, or, if the H extension is implemented, matching bits in **hip** and **hie** are both one; and (c) if the H extension is implemented, the corresponding bit in **hideleg** is zero.

5.4.1. Configuring priorities of major interrupts at supervisor level

Supervisor-level priorities for major interrupts 0-63 are optionally configurable in an array of supervisor-level **iprio***k* registers accessed through **siselect** and **sireg**. This array has the same structure when XLEN = 32 or 64 as does the machine-level **iprio** array. To summarize, when XLEN = 32, there are sixteen 32-bit registers with these **siselect** addresses:

```
0x30 iprio0
0x31 iprio1
 ...    ...
0x3F iprio15
```

Each register controls the priorities of four interrupts, one 8-bit byte per interrupt. When XLEN = 64, only the even-numbered registers exist:

```
0x30 iprio0
0x32 iprio2
 ...    ...
0x3E iprio14
```

Each register controls the priorities of eight interrupts. If XLEN = 64 and **siselect** is an odd value in the range **0x31**-**0x3F**, attempting to access **sireg** raises an illegal instruction exception.

The valid registers **iprio0**-**iprio15** are known collectively as the iprio *array* for supervisor level. Each byte of a valid **iprio***k* register is either a read-only zero or a **WARL** unsigned integer field implementing exactly IPRIOLEN bits.

For a given interrupt number, if the corresponding bit is not writable either in **sie** or, if the H extension is implemented, in **hie**, then the interrupt's priority number in the supervisor-level **iprio**

array must be read-only zero as well. The priority number for a supervisor-level external interrupt (bits 15:8 of `iprio2`) must also be read-only zero. Aside from these two restrictions, implementations may freely choose which priority number fields are settable and which are read-only zeros.

As always, platform standards may require that priorities be configurable for certain interrupt causes.

It is expected that many higher-end systems will not support the ability to configure the priorities of major interrupts at supervisor level as described in this section. Linux in particular is not designed to take advantage of such facilities if provided. The `iprio` *array must be accessible but may simply be all read-only zeros.*

The supervisor-level `iprio` array accessed via `siselect` and `sireg` affects the prioritization of interrupts only when they trap to S-mode. When an interrupt's priority number in the array is zero (either read-only zero or set to zero), its priority is the default order from Section 5.1. Setting an interrupt's priority number instead to a nonzero value p gives that interrupt nominally the same priority as a supervisor-level external interrupt with priority number p. For an interrupt that defaults to a higher priority than supervisor external interrupts, setting its priority number to a nonzero value lowers its priority. For an interrupt that defaults to a lower priority than supervisor external interrupts, setting its priority number to a nonzero value raises its priority. When two interrupt causes have been assigned the same nominal priority, ties are broken by the default priority order. Table 12 summarizes the effect of priority numbers on interrupt priority.

If supported, setting the priority number for VS-level external interrupts (bits 23:16 of `iprio2`) to a nonzero value p has the effect of giving the entire category of VS external interrupts nominally the same priority as a supervisor external interrupt with priority number p, when VS external interrupts trap to S-mode.

Table 12. Effect of the supervisor-level iprio array on the priorities of interrupts taken in S-mode. For interrupts with the same priority number, the default order of Section 5.1 prevails.

	Interrupts with default priority above supervisor external interrupts	Supervisor external interrupts	Interrupts with default priority below supervisor external interrupts
Priority order	Priority number in supervisor-level `iprio` array	Priority number from interrupt controller (APLIC or IMSIC)	Priority number in supervisor-level `iprio` array
Highest	0		
	1 2 ... 254 255	1 2 ... 254 255	1 2 ... 254 255
		256 and above (IMSIC only)	
Lowest			0

If bit 9 for a supervisor external interrupt (SEI) is one in `mideleg` or `mvien` and in `mvip`, causing `sip`.SEIP to be one, but there is no supervisor-level interrupt from the hart's external interrupt controller (APLIC or IMSIC), then a priority number for the SEI is not supplied by the external

interrupt controller as usual. In that case, the SEI is assigned a priority number of 256.

If the system has an original PLIC for backward compatibility with older software, reset should initialize the supervisor-level `iprio` array to all zeros.

5.4.2. Supervisor top interrupt CSR (`stopi`)

Supervisor-level CSR `stopi` is read-only with width SXLEN. A read of `stopi` returns information about the highest-priority pending-and-enabled interrupt for supervisor level, in this format:

```
bits 27:16 IID
bits 7:0   IPRIO
```

All other bits of `stopi` are reserved and read as zeros.

The value of `stopi` is zero unless: (a) there is an interrupt that is both pending in `sip` and enabled in `sie`, or, if the H extension is implemented, both pending in `hip` and enabled in `hie`; and (b) the interrupt is not delegated to a less-privileged level (by `hideleg`, if the H extension is implemented). When there is a pending-and-enabled major interrupt for supervisor level, field IID is the major identity number of the highest-priority interrupt, and field IPRIO indicates its priority.

If all bytes of the supervisor-level `iprio` array are read-only zeros, a simplified implementation of field IPRIO is allowed in which its value is always 1 whenever `stopi` is not zero.

Otherwise, when `stopi` is not zero, if the priority number for the reported interrupt is in the range 1 to 255, IPRIO is simply that number. If the interrupt's priority number is zero or greater than 255, IPRIO is set to either 0 or 255 as follows:

- If the interrupt's priority number is greater than 255, then IPRIO is 255 (lowest representable priority).
- If the interrupt's priority number is zero and interrupt number IID has a default priority higher than a supervisor external interrupt, then IPRIO is 0 (highest priority).
- If the interrupt's priority number is zero and interrupt number IID has a default priority lower than a supervisor external interrupt, then IPRIO is 255 (lowest representable priority).

To ensure that `stopi` is never zero when an interrupt is pending and enabled for supervisor level, if major interrupt 0 can trap to S-mode, it must have a default priority lower than a supervisor external interrupt.

The value of `stopi` is not affected by the global interrupt enable SIE in CSR `sstatus`.

The RISC-V Privileged Architecture ensures that, when the value of `stopi` is not zero, a trap is taken to S-mode for the interrupt indicated by field IID if either the current privilege mode is S and `sstatus`.SIE is one, or the current privilege mode has less privilege than S-mode. The trap itself does not cause the value of `stopi` to change.

The following pseudocode shows how a supervisor-level trap handler might read `stopi` to avoid redundant restoring and saving of processor registers when an interrupt arrives during the handling of another trap (either a synchronous exception or an earlier interrupt):

```
save processor registers
i = read CSR scause
```

```
if (i >= 0) {
    handle synchronous exception i
    restore sstatus if necessary
}
if (sstatus.SPIE == 1) {
    loop {
        i = read CSR stopi
        if (i == 0) exit loop
        i = i>>16
        call the interrupt handler for major interrupt i
    }
}
restore processor registers
return from trap
```

(This example can be further optimized, but with an increase in complexity.)

In order for this algorithm to function correctly, `sstatus`.SPIE must be set to 1 before executing an SRET that changes the privilege mode.

Assuming `sstatus` *is saved and restored by trap handlers at entry and exit as is common, it is sufficient to set* `sstatus`*.SPIE = 1 only once, before the first use of SRET that changes privilege mode. After an SRET, a trap back to S-mode will restore* `sstatus`*.SPIE = 1; and if the trap handler preserves* `sstatus`*, it will still be true before the next SRET that ends the handler.*

5.5. WFI (Wait for Interrupt) instruction

The RISC-V Privileged Architecture specifies that instruction WFI (Wait for Interrupt) may suspend execution at a hart until an interrupt is pending for the hart. The Advanced Interrupt Architecture (AIA) redefines when execution must resume following a WFI.

According to the base RISC-V Privileged Architecture, instruction execution must resume from a WFI whenever any interrupt is both pending and enabled in CSRs `mip` and `mie`, ignoring any delegation indicated by `mideleg`. With the AIA, this succinct rule is no longer appropriate, due to the mechanisms the AIA adds for virtual interrupts. Instead, execution must resume from a WFI whenever an interrupt is pending at any privilege level (regardless of whether the interrupt privilege level is higher or lower than the hart's current privilege mode).

An interrupt is pending at machine level if register `mtopi` is not zero. If S-mode is implemented, an interrupt is pending at supervisor level if `stopi` is not zero. And if the H extension is implemented, an interrupt is pending at VS level if `vstopi` (Section 6.3.3) is not zero.

The AIA's rule for WFI gives the same behavior as the base Privileged Architecture's rule when `mvien` *= 0 and, if the H extension is implemented, also* `hvien` *= 0 and* `hvictl`*.VTI = 0, thus disabling all virtual interrupts not visible in* `mip`*. (The AIA's hypervisor registers are covered in the next chapter, "Interrupts for Virtual Machines (VS Level)".)*

Chapter 6. Interrupts for Virtual Machines (VS Level)

When the H extension is implemented, a hart's set of possible privilege modes includes the *virtual supervisor* (VS) and *virtual user* (VU) modes for hosting virtual harts. The Advanced Interrupt Architecture adds to the H extension new interrupt facilities aligned with those described earlier for supervisor-level interrupts.

As introduced in Chapter 2, several hypervisor and VS CSRs are added: `hvien`, `hvictl`, `hviprio1`, `hviprio2`, `vsiselect`, `vsireg`, `vstopei`, and `vstopi`. (And for RV32, the following high-half CSRs are also added: `hidelegh`, `hvienh`, `hviph`, `hviprio1h`, `hviprio2h`, `vsiph` and `vsieh`.) As always, when executing in VS-mode or VU-mode, the VS CSRs substitute for the corresponding supervisor CSRs.

To give software that runs in a virtual machine the appearance of executing on a real machine that implements the Advanced Interrupt Architecture at supervisor level, responsibility is shared between hypervisor software and the hardware facilities described in this chapter. While some behaviors can be handled directly by hardware, others require significant emulation by the hypervisor, sometimes with hardware assistance.

6.1. VS-level external interrupts with a guest interrupt file

When a hart implements the H extension, it is recommended that the hart also have an IMSIC with guest interrupt files. Assuming guest interrupt files are available, each can be assigned to a virtual hart at the physical hart to act as the supervisor-level interrupt file for that virtual hart. If there are N guest interrupt files, then N virtual harts at that physical hart may each have a physical guest interrupt file to serve as its (virtual) supervisor-level interrupt file. The guest interrupt file for the current virtual hart is always indicated by the VGEIN field of CSR `hstatus`. When VGEIN is not the valid number of a guest interrupt file, the current virtual hart has no guest interrupt file to act as its supervisor-level interrupt file.

When `hstatus`.VGEIN is the valid number of a guest interrupt file, values of `vsiselect` in the range `0x70`-`0xFF` select registers of this guest interrupt file, just as values of `siselect` in the same range select registers of the IMSIC's true supervisor-level interrupt file. The registers of an interrupt file that are accessed indirectly through `vsiselect` and `vsireg` are documented in Chapter 3 on the IMSIC, along with IMSIC-only CSR `vstopei`. Because all IMSIC interrupt files act identically, the guest interrupt file that a virtual hart accesses through CSRs `siselect`, `sireg`, and `stopei` is indistinguishable from a true supervisor-level interrupt file as seen from S-mode (or HS-mode).

In addition to an IMSIC at each hart, a virtual machine may also need to see a PLIC or APLIC. However, unlike an IMSIC's ability to provide physical guest interrupt files for virtual harts, a PLIC or APLIC must be emulated for a virtual machine by the hypervisor.

The Advanced Interrupt Architecture does not currently include hardware assistance for virtualizing an APLIC. For small numbers of harts, such hardware would be substantially larger than that required to implement guest interrupt files for an IMSIC. It is assumed that most high-performance I/O can be done through devices that can send MSIs directly to guest interrupt files (such as devices attached through a PCI Express interconnect). For the types of devices whose interrupts must go through a (virtual) APLIC, the overhead cost

of emulating the APLIC is expected to be less significant.

When a virtual hart appears to have an IMSIC because a guest interrupt file is assigned to it, all external interrupts, real or emulated, destined for the virtual hart must go through this perceived IMSIC. A hypervisor can easily inject an emulated external interrupt into the guest interrupt file selected by `hstatus`.VGEIN by setting a bit in the interrupt-pending array indirectly accessed through `vsiselect` and `vsireg`. When a virtual hart has a guest interrupt file, a hypervisor is not normally expected to set bit VSEIP in CSR `hvip`.

In the special case that an emulated APLIC for a virtual machine has a wired interrupt source that equates to an actual interrupt source of a real APLIC, if software running in this virtual machine configures its virtual APLIC to forward interrupts from that source as MSIs to a specific virtual hart, the hypervisor can configure the real APLIC to forward the actual interrupts directly as MSIs to the virtual hart's guest interrupt file. In this way, although the hypervisor must trap and emulate the virtual machine's memory accesses that configure the forwarding of interrupts at the virtual APLIC, the interrupts themselves can be converted automatically into real MSIs for the guest interrupt file, without the hypervisor being invoked for each arriving interrupt.

6.1.1. Direct control of a device by a guest OS

To ensure proper support for interrupts, two conditions must be met before a hypervisor may allow a guest OS running in a virtual machine to directly control a physical device that sends MSIs: First, each virtual hart must have a guest interrupt file assigned to it, giving each its own apparent IMSIC within the virtual machine. Second, interrupts from the device must be signaled by wire through an APLIC that can translate these interrupts into MSIs, or the system must have an IOMMU that can translate the addresses of MSI memory writes made by the device itself.

If a guest OS directly controls a device capable of sending MSIs, it will naturally configure MSIs at the device with the guest physical addresses the OS sees for the IMSICs of its virtual harts, not knowing that these addresses are only virtual. When the device performs a memory write for an MSI, the destination address of this write must be translated by the IOMMU from the guest physical address assigned by the guest OS into the true physical address of the target guest interrupt file, using a translation table supplied by the hypervisor.

By design, the translation an IOMMU must do for device MSIs is fundamentally no different than the address translation the IOMMU already must perform for other memory accesses from the same device, converting guest physical addresses into true physical addresses. Because each virtual hart is assigned a dedicated, physical guest interrupt file that is indistinguishable from a true supervisor-level interrupt file, no translation is needed for the data of an MSI write, which specifies the interrupt's identity number in the target interrupt file.

6.1.2. Migrating a virtual hart to a different guest interrupt file

When it is necessary to move a virtual hart from one physical hart to another, if the virtual hart uses a guest interrupt file, the specific guest interrupt file assigned to it must change from the one in use at the old physical hart to a different one at the new physical hart. Because each guest interrupt file is physically tied to a single physical hart, a virtual hart cannot bring its guest interrupt file with it when it moves.

The process of migrating a virtual hart from one guest interrupt file to another is more complex than moving most other state held by the virtual hart. After the destination guest interrupt file has been chosen at the new physical hart, the following steps are recommended:

1. At the old interrupt file, save to memory the values of registers **eidelivery** and **eithreshold**, and set **eidelivery** = 0.
2. At the new interrupt file, set **eidelivery** = 0, and zero all implemented interrupt-pending bits (the **eip** array).
3. Modify the relevant translation tables at all IOMMUs so that MSIs for this virtual interrupt file are now sent to the new physical interrupt file. Likewise, if any interrupts at an APLIC are forwarded by MSIs to the old interrupt file, reconfigure the APLIC to send them to the new interrupt file. As needed, synchronize with all IOMMUs and APLICs to ensure that no straggler MSIs will arrive at the old interrupt file after this step. Synchronizing with an APLIC can be accomplished using the algorithm of Section 4.9.3.
4. At the old interrupt file, dump to memory all implemented interrupt-pending and interrupt-enable bits (the **eip** and **eie** arrays). After this step is done, the old interrupt file is no longer in use.
5. At the new interrupt file, logically OR the interrupt-pending bits that were saved in step 4 into the new interrupt file, using instruction CSRS to write to the **eip** array. Also, load the interrupt-enable bits that were saved in step 4 into the **eie** array.
6. At the new interrupt file, load registers **eithreshold** and **eidelivery** with the values that were saved in step 1.

Resuming execution of the virtual hart at the new physical hart is not recommended until the entire interrupt file has been fully migrated.

Resuming execution of the virtual hart before the interrupt file is fully migrated could allow software running in the virtual machine to see multiple MSIs arriving from a single device in an order that should not happen. While this would rarely matter in practice, it runs the risk of wedging a device driver that depends (perhaps inadvertently) on a valid ordering of events.

6.2. VS-level external interrupts without a guest interrupt file

Although it is recommended that harts implementing the hypervisor extension also have IMSICs with guest interrupt files, this is not a requirement. Even assuming guest interrupt files exist, it may happen that there are more virtual harts at a physical hart than guest interrupt files, leaving some virtual harts without one. In either case, a hypervisor must emulate an external interrupt controller for a virtual hart without the benefit of a guest interrupt file allocated to the virtual hart.

When emulating an external interrupt controller for a virtual hart, if configurable interrupt priority is not supported for the virtual hart other than for external interrupts, then external interrupts may be asserted to VS level simply by setting bit VSEIP in **hvip**, as defined by the H extension. However, to emulate both an external interrupt controller and priority configurability for non-external interrupts, a hypervisor must make use of CSR **hvictl** (Hypervisor Virtual Interrupt Control), described later in the next section.

6.3. Interrupts at VS level

6.3.1. Configuring priorities of major interrupts at VS level

Like for supervisor level, the Advanced Interrupt Architecture optionally allows major VS-level

interrupts to be configured by software to intermix in priority with VS-level external interrupts. As documented in Section 5.4, interrupt priorities for supervisor level are configured by the `iprio` array accessed indirectly through CSRs `siselect` and `sireg`. The `siselect` addresses for the `iprio` array registers are `0x30`-`0x3F`.

VS level has its own `vsiselect` and `vsireg`, but unlike supervisor level, there are no registers at `vsiselect` addresses `0x30`-`0x3F`. When `vsiselect` has a value in the range `0x30`-`0x3F`, an attempt from VS-mode to access `sireg` (really `vsireg`) causes a virtual instruction exception. To give a virtual hart the illusion of an array of `iprio` registers accessed through `siselect` and `sireg`, a hypervisor must emulate the VS-level `iprio` array when accesses to `sireg` from VS-mode cause virtual instruction traps.

Instead of a physical VS-level `iprio` array, a separate hardware mechanism is provided for configuring the priorities of a subset of interrupts for VS level, using hypervisor CSRs `hviprio1` and `hviprio2`. The subset of major interrupt numbers whose priority may be configured in hardware are these:

1	Supervisor software interrupt
5	Supervisor timer interrupt
13	Counter overflow interrupt
14-23	*Reserved for standard local interrupts*

For interrupts directed to VS level, software-configurable priorities are not supported in hardware for standard local interrupts in the range 32-48.

For custom interrupts, priority configurability may be supported in hardware by custom CSRs, expanding upon `hviprio1` *and* `hviprio2` *for standard interrupts.*

Registers `hviprio1` and `hviprio2` have these formats:

`hviprio1`:

bits 7:0	*Reserved for priority number for interrupt 0; reads as zero*
bits 15:8	Priority number for interrupt 1, supervisor software interrupt
bits 23:16	*Reserved for priority number for interrupt 4; reads as zero*
bits 31:24	Priority number for interrupt 5, supervisor timer interrupt
bits 39:32	*Reserved for priority number for interrupt 8; reads as zero*
bits 47:40	Priority number for interrupt 13, counter overflow interrupt
bits 55:48	Priority number for interrupt 14
bits 63:56	Priority number for interrupt 15

`hviprio2`:

bits 7:0	Priority number for interrupt 16
bits 15:8	Priority number for interrupt 17
bits 23:16	Priority number for interrupt 18
bits 31:24	Priority number for interrupt 19

bits 39:32	Priority number for interrupt 20
bits 47:40	Priority number for interrupt 21
bits 55:48	Priority number for interrupt 22
bits 63:56	Priority number for interrupt 23

Each priority number in **hviprio1** and **hviprio2** is a **WARL** unsigned integer field that is either read-only zero or implements a minimum of IPRIOLEN bits or 6 bits, whichever is larger, and preferably all 8 bits. Implementations may freely choose which priority number fields are read-only zeros, but all other fields must implement the same number of integer bits. A minimal implementation of these CSRs has them both be read-only zeros.

A hypervisor can choose to employ registers **hviprio1** and **hviprio2** when emulating the (virtual) supervisor-level **iprio** array accessed indirectly through **siselect** and **sireg** (really **vsiselect** and **vsireg**) for a virtual hart. For interrupts not in the subset supported by **hviprio1** and **hviprio2**, the priority number bytes in the emulated **iprio** array can be read-only zeros.

Providing hardware support for configurable priority for only a subset of major interrupts at VS level is a compromise. The utility of being able to control interrupt priorities at VS level is arguably illusory when all traps to M-mode and HS-mode—both interrupts and synchronous exceptions—have absolute priority, and when each virtual hart may also be competing for resources against other virtual harts well beyond its control. Nevertheless, priority configurability has been made possible for the most likely subset of interrupts, while minimizing the number of added CSRs that must be swapped on a virtual hart switch.

Major interrupts outside the priority-configurable subset can still be directed to VS level, but their priority will simply be the default order defined in Section 5.1.

If a hypervisor really must emulate configurability of priority for interrupts beyond the subset supported by **hviprio1** and **hviprio2**, it can do so with extra effort by setting bit VTI of CSR **hvictl**, described in the next subsection.

6.3.2. Virtual interrupts for VS level

Assuming a virtual hart does not need configurable priority for major interrupts beyond the subset supported in hardware by **hviprio1** and **hviprio2**, a hypervisor can assert interrupts to the virtual hart using CSRs **hvien** (Hypervisor Virtual-Interrupt-Enable) and **hvip** (Hypervisor Virtual-Interrupt-Pending bits). These CSRs affect interrupts for VS level much the same way that **mvien** and **mvip** do for supervisor level, as explained in Section 5.3.

Each bit of registers **hvien** and **hvip** corresponds with an interrupt number in the range 0-63. Bits 12:0 of **hvien** are reserved and must be read-only zeros, while bits 12:0 of **hvip** are defined by the H extension. Specifically, bits 10, 6, and 2 of **hvip** are writable bits that correspond to VS-level external interrupts (VSEIP), VS-level timer interrupts (VSTIP), and VS-level software interrupts (VSSIP), respectively.

The following applies only to the CSR bits for interrupt numbers 13-63: When a bit in **hideleg** is one, then the same bit position in **vsip** is an alias for the corresponding bit in **sip**. Else, when a bit in **hideleg** is zero and the matching bit in **hvien** is one, the same bit position in **vsip** is an alias for the corresponding bit in **hvip**. A bit in **vsip** is read-only zero when the corresponding bits in **hideleg** and **hvien** are both zero. The combined effects of **hideleg** and **hvien** on **vsip** and **vsie** are summarized in

Table 13.

Table 13. The effects of `hideleg` *and* `hvien` *on* `vsip` *and* `vsie` *for major interrupts 13-63.*

`hideleg`[n]	`hvien`[n]	`vsip`[n]	`vsie`[n]
0	0	Read-only 0	Read-only 0
0	1	Alias of `hvip`[n]	Writable
1	-	Alias of `sip`[n]	Alias of `sie`[n]

For interrupt numbers 13-63, a bit in `vsie` is writable if and only if the corresponding bit is set in either `hideleg` or `hvien`. When an interrupt is delegated by `hideleg`, the writable bit in `vsie` is an alias of the corresponding bit in `sie`; else it is an independent writable bit. The H extension specifies when bits 12:0 of `vsie` are aliases of bits in `hie`. As usual, bits that are not writable in `vsie` must be read-only zeros.

If a bit of `hideleg` is zero and the corresponding bit in `hvien` is changed from zero to one, then the value of the matching bit in `vsie` becomes UNSPECIFIED. Likewise, if a bit of `hvien` is one and the corresponding bit in `hideleg` is changed from one to zero, the value of the matching bit in `vsie` again becomes UNSPECIFIED.

For interrupt numbers 13-63, implementations may freely choose which bits of `hvien` are writable and which bits are read-only zero or one. If such a bit in `hvien` is read-only zero (preventing the virtual interrupt from being enabled), the same bit should be read-only zero in `hvip`. All other bits for interrupts 13-63 must be writable in `hvip`.

CSR `hvictl` (Hypervisor Virtual Interrupt Control) provides further flexibility for injecting interrupts into VS level in situations not fully supported by the facilities described thus far, but only with more active involvement of the hypervisor. A hypervisor must use `hvictl` for any of the following:

- asserting for VS level a major interrupt not supported by `hvien` and `hvip`;
- implementing configurability of priorities at VS level for major interrupts beyond those supported by `hviprio1` and `hviprio2`; or
- emulating an external interrupt controller for a virtual hart without the use of an IMSIC's guest interrupt file, while also supporting configurable priorities both for external interrupts and for major interrupts to the virtual hart.

Among the possible uses, `hvictl` *is needed for a hypervisor to fully emulate HS-mode in VS-mode, which is a requirement for the hosting of nested hypervisors without paravirtualization.*

The format of `hvictl` is:

bit 30	VTI
bits 27:16	IID (WARL)
bit 9	DPR
bit 8	IPRIOM
bits 7:0	IPRIO

All other bits of `hvictl` are reserved and read as zeros.

When bit VTI (Virtual Trap Interrupt control) = 1, attempts from VS-mode to explicitly access CSRs `sip` and `sie` (or, for RV32 only, `siph` and `sieh`) cause a virtual instruction exception. Furthermore, for any given CSR, if there is some circumstance in which a write to the register may cause a bit of `vsip` to change from one to zero, excluding bit 9 for external interrupts (SEIP), then when VTI = 1, a virtual instruction exception is raised also for any attempt by the guest to write this register. Both the value being written to the CSR and the value of `vsip` (before or after) are ignored for determining whether to raise the exception. (Hence a write would not actually need to change a bit of `vsip` from one to zero for the exception to be raised.) In particular, if register `vstimecmp` is implemented (from extension Sstc), then attempts from VS-mode to write to `stimecmp` (or, for RV32 only, `stimecmph`) cause a virtual instruction exception when VTI = 1.

For the standard local interrupts (major identities 13-23 and 32-47), and for software interrupts (SSI), the corresponding interrupt-pending bits in `vsip` *are defined as "sticky," meaning a guest can clear them only by writing directly to* `sip` *(really* `vsip`*). Among the standard-defined interrupts, that leaves only timer interrupts (STI), which can potentially be cleared in* `vsip` *by writing a new value to* `vstimecmp`*.*

All `hvictl` fields together can affect the value of CSR `vstopi` (Virtual Supervisor Top Interrupt) and therefore the interrupt identity reported in `vscause` when an interrupt traps to VS-mode. IID is a **WARL** unsigned integer field with at least 6 implemented bits, while IPRIO is always the full 8 bits. If k bits are implemented for IID, then all values 0 through $2^k - 1$ are supported, and a write to `hvictl` sets IID equal to bits $(15 + k)$:16 of the value written.

For a virtual interrupt specified for VS level by `hvictl`, if VTI = 1 and $IID \neq 9$, field DPR (Default Priority Rank) determines the interrupt's presumed default priority order relative to a (virtual) supervisor external interrupt (SEI), major identity 9, as follows:

0 = interrupt has higher default priority than an SEI

1 = interrupt has lower default priority than an SEI

When `hvictl`.IID = 9, DPR is ignored.

Register `hvictl` *has no effect on any of* `mip`*,* `sip`*,* `hip`*, or* `vsip`*; it affects only* `vstopi` *and the trapping of some instructions.*

6.3.3. Virtual supervisor top interrupt CSR (`vstopi`)

Read-only CSR `vstopi` is VSXLEN bits wide and has the same format as `stopi`:

bits 27:16 IID

bits 7:0 IPRIO

`vstopi` returns information about the highest-priority interrupt for VS level, found from among these candidates (prefixed by + signs):

- if bit 9 is one in both `vsip` and `vsie`, `hstatus`.VGEIN is the valid number of a guest interrupt file, and `vstopei` is not zero:
  - + a supervisor external interrupt (code 9) with the priority number indicated by `vstopei`;
- if bit 9 is one in both `vsip` and `vsie`, `hstatus`.VGEIN = 0, and `hvictl` fields IID = 9 and $IPRIO \neq 0$:
  - + a supervisor external interrupt (code 9) with priority number `hvictl`.IPRIO;

- if bit 9 is one in both `vsip` and `vsie`, and neither of the first two cases applies:
  - + a supervisor external interrupt (code 9) with priority number 256;
- if `hvictl`.VTI = 0:
  - + the highest-priority pending-and-enabled major interrupt indicated by `vsip` and `vsie` other than a supervisor external interrupt (code 9), using the priority numbers assigned by `hviprio1` and `hviprio2`;
- if `hvictl` fields VTI = 1 and $IID \neq 9$:
  - + the major interrupt specified by `hvictl` fields IID, DPR, and IPRIO.

In the list above, all "supervisor" external interrupts are virtual, directed to VS level, having major code 9 at VS level.

The list of candidate interrupts can be reduced to two finalists relatively easily by observing that the first three list items are mutually exclusive of one another, and the remaining two items are also mutually exclusive of one another.

When `hvictl`*.VTI = 1, the absence of an interrupt for VS level can be indicated only by setting* `hvictl`*.IID = 9. Software might want to use the pair IID = 9, IPRIO = 0 generally to represent no interrupt in* `hvictl`.

When no interrupt candidates satisfy the conditions of the list above, `vstopi` is zero. Else, `vstopi` fields IID and IPRIO are determined by the highest-priority interrupt from among the candidates. The usual priority order for supervisor level applies, as specified by Table 12, except that priority numbers are taken from the candidate list above, not from the supervisor-level `iprio` array. Ties in nominal priority are broken as usual by the default priority order from Table 8, unless `hvictl` fields VTI = 1 and $IID \neq 9$ (last item in the candidate list above), in which case default priority order is determined solely by `hvictl`.DPR. If bit IPRIOM (IPRIO Mode) of `hvictl` is zero, IPRIO in `vstopi` is 1; else, if the priority number for the highest-priority candidate is within the range 1 to 255, IPRIO is that value; else, IPRIO is set to either 0 or 255 in the manner documented for `stopi` in Section 5.4.2.

6.3.4. Interrupt traps to VS-mode

The Advanced Interrupt Architecture modifies the H extension such that an interrupt is pending at VS level if and only if `vstopi` is not zero. CSRs `vsip` and `vsie` do not by themselves determine whether a VS-level interrupt is pending, though they may do so indirectly through their effect on `vstopi`.

Whenever `vstopi` is not zero, if either the current privilege mode is VS-mode and the SIE bit in CSR `vsstatus` is one, or the current privilege mode is VU-mode, a trap is taken to VS-mode for the interrupt indicated by field IID of `vstopi`.

The Exception Code field of `vscause` must implement at least as many bits as needed to represent the largest value that field IID of `vstopi` can have for the given hart.

Chapter 7. Interprocessor Interrupts (IPIs)

By default, unless a platform has a different mechanism for interprocessor interrupts (IPIs), the base RISC-V Privileged Architecture specifies that a machine with multiple harts must provide for each hart an implementation-defined memory address that can be written to signal a machine-level *software interrupt* (major code 3) at that hart. IPIs at machine level can thus be sent to any hart as machine-level software interrupts.

A RISC-V software interrupt acts only as a minimal "doorbell" signal. Software at the receiving hart is responsible for recognizing an incoming software interrupt as an IPI and decoding its purpose further, usually making use of additional data stored by the sender in ordinary memory.

The same kind of mechanism (but with a different set of memory addresses) may or may not exist for signaling supervisor-level software interrupts (major code 1) at remote harts as well. If not directly supported in this way, a supervisor-level software interrupt is typically sent to another hart instead through an environment call from supervisor mode to machine mode. An operating system running in S-mode thus invokes a specific SBI function for delivering a software interrupt to another hart, causing machine-level software at the originating hart to send a machine-level IPI to the destination hart, where software then sets the supervisor-level software interrupt-pending bit (SSIP) in CSR `mip`.

When harts have IMSICs, instead of using the base Privileged Architecture's mechanism for signaling software interrupts at remote harts, an IPI can be sent to a hart by writing to the destination hart's IMSIC, the same as a regular message-signaled interrupt (MSI). In that case, an incoming IPI appears at the destination hart as an *external interrupt* routed through the IMSIC, rather than as a software interrupt as before. However, so long as the same software (e.g. an operating system or machine monitor) is in control at both endpoints of an IPI, source and destination, there should be no reason for a destination hart to misinterpret the purpose of an incoming external interrupt that represents an IPI.

If harts do not have IMSICs, then the method specified by the base Privileged Architecture is assumed to be used for IPIs, signaling software interrupts at destination harts. On the other hand, when harts have IMSICs, the machinery for triggering software interrupts at remote harts is redundant with the capabilities of the IMSICs, so it is downgraded from a requirement to an option, useful perhaps only to provide software compatibility across a range of RISC-V systems, with and without IMSICs. If a machine implements IMSICs and not the earlier software-interrupt mechanism, then the bits of CSRs `mip` and `mie` for machine-level software interrupts, MSIP and MSIE, are hardwired to zero in harts.

If a machine implements IMSICs but not the software-interrupt mechanism, the latter can still be fully emulated at supervisor level for S-mode or VS-mode, by trapping on writes to the special memory addresses that should signal supervisor-level software interrupts at remote harts. On such a trap, software can send a higher-level IPI via IMSIC to the destination hart, where the higher-level software then can set the SSIP bit in `sip` at the intended privilege level, S or VS.

Similarly, SBI environment calls for sending IPIs can easily continue to be supported without clients being at all aware of a change in the underlying hardware for delivering IPIs between harts.

When software sends IPIs by writing MSIs to the IMSICs of other harts, programmers should consider also the need to execute a FENCE instruction before each store instruction that writes such an MSI. In the absence of FENCEs, many systems guarantee

> *to preserve the order of a hart's loads and stores only to/from individual devices, not among multiple devices, and not at all for accesses to main memory. With such a system, it must be remembered that each IMSIC is likely to be considered a separate device among the many. For example, if hart A wants to notify hart B that it has completed a task involving accesses to some I/O device, hart A may need to execute a FENCE before sending an MSI to B's IMSIC, to ensure that all of A's accesses to the device have actually completed before the MSI could arrive at B. Similarly, if hart A stores anything to memory that should be visible at hart B, a FENCE is likely needed before a subsequent store sending an MSI to B's IMSIC.*

Chapter 8. IOMMU Support for MSIs to Virtual Machines

The existence of an IOMMU in a system makes it possible for a guest operating system, running in a virtual machine, to be given direct control of an I/O device with only minimal hypervisor intervention. A guest OS with direct control of a device will program the device with guest physical addresses, because that is all the OS knows. When the device then performs memory accesses using those addresses, an IOMMU is responsible for translating those guest physical addresses into machine physical addresses, referencing address-translation data structures supplied by the hypervisor.

To handle MSIs from a device controlled by a guest OS, an IOMMU must be able to redirect those MSIs to a guest interrupt file in an IMSIC. Systems that do not have IMSICs with guest interrupt files do not need to implement the facilities described in this chapter.

Because MSIs from devices are simply memory writes, they would naturally be subject to the same address translation that an IOMMU applies to other memory writes. However, the Advanced Interrupt Architecture requires that IOMMUs treat MSIs directed to virtual machines specially, in part to simplify software, and in part to allow optional support for *memory-resident interrupt files*.

This chapter uses the term *IOMMU* in a generic sense that encompasses all translation and transaction processing services required to virtualize device accesses and is concerned only with how an IOMMU recognizes and processes MSIs directed to virtual machines. Most other functions and details of an IOMMU are beyond the scope of this standard, and must be specified elsewhere.

The RISC-V IOMMU Architecture Specification provides a detailed description of the IOMMU architecture, dividing translation and transaction processing functionality into blocks such as IOMMU, IO Bridge, etc. and describing how those blocks are integrated into a system.

If a single physical I/O device can be subdivided for control by multiple separate device drivers, each sub-device is referred to here as one device.

8.1. Device contexts at an IOMMU

The following assumptions are made about the IOMMUs in a system:

- For each I/O device connected to the system through an IOMMU, software can configure at the IOMMU a *device context*, which associates with the device a specific virtual address space and any other per-device parameters the IOMMU may support. By giving devices each their own separate device context at an IOMMU, each device can be individually configured for a separate operating system, which may be a guest OS or the main (host) OS. On every memory access initiated by a device, hardware indicates to the IOMMU the originating device by some form of unique device identifier, which the IOMMU uses to locate the appropriate device context within data structures supplied by software. For PCI, for example, the originating device may be identified by the unique triple of PCI bus number, device number, and function number.
- An IOMMU optionally translates the addresses of a device's memory accesses using address-translation data structures—typically page tables—specified by software via the corresponding device context. The smallest granularity of address translation implemented by all IOMMUs is not larger than a 4-KiB page, matching that of standard RISC-V address-translation page tables. (An IOMMU may in fact employ page tables in the same format as the page-based address translation

defined by the RISC-V Privileged Architecture, but this is not required.)

The Advanced Interrupt Architecture adds to device contexts these fields, as needed:

- an *MSI address mask* and *address pattern*, used together to identify pages in the guest physical address space that are the destinations of MSIs; and
- the real physical address of an *MSI page table* for controlling the translation and/or conversion of MSIs from the device.

The MSI address mask and address pattern are each unsigned integers with the same width as guest physical page numbers, i.e., 12 bits narrower than the maximum supported width of a guest physical address. Their use is explained in Section 8.4.

A device context's MSI page table is separate from the usual address-translation data structures used to translate other memory accesses from the same device. The form and function of MSI page tables are the subject of most of the rest of this chapter.

A device context is given an independent page table for MSIs for two reasons:

First, hypervisors running under Linux or a similar OS can benefit from separate control of MSI translations to help simplify the case when virtual harts are migrated from one physical hart to another. As noted in Section 6.1.2, when a virtual hart's interrupt files are mapped to guest interrupt files in the real machine, migration of the virtual hart causes the physical guest interrupt files underlying those virtual interrupt files to change. However, because on other systems (not RISC-V) the migration of a virtual hart does not affect the mapping from guest physical addresses to real physical addresses, the internal functions of Linux that perform this migration are not set up to modify an IOMMU's address-translation tables to adjust for the changing physical locations of RISC-V virtual interrupt files. Giving a hypervisor control of a separate MSI translation table at an IOMMU bypasses this limitation. The MSI page table can be modified at will by the hypervisor and/or by the subsystem that manages interrupts without coordinating with the many other OS components concerned with regular address translation.

Second, specifying a separate MSI page table facilitates the use of memory-resident interrupt files (MRIFs), which are introduced in Section 8.3. A dedicated MSI page table can easily support a special table entry format for MRIFs (Section 8.5.2) that would be entirely foreign and difficult to retrofit to any other address-translation data structures.

8.2. Translation of addresses for MSIs from devices

To support the delivery of MSIs from I/O devices directly to RISC-V virtual machines without hypervisor intervention, an IOMMU must be able to translate the guest physical address of an MSI to the real physical address of an IMSIC's guest interrupt file in the machine, as illustrated in Figure 6. This address translation is controlled by the MSI page table configured in the appropriate device context at the IOMMU. Because every interrupt file, real or virtual, occupies a naturally aligned 4-KiB page of address space, the required address translation is from a virtual (guest) page address to a physical page address, the same as supported by regular RISC-V page-based address translation.

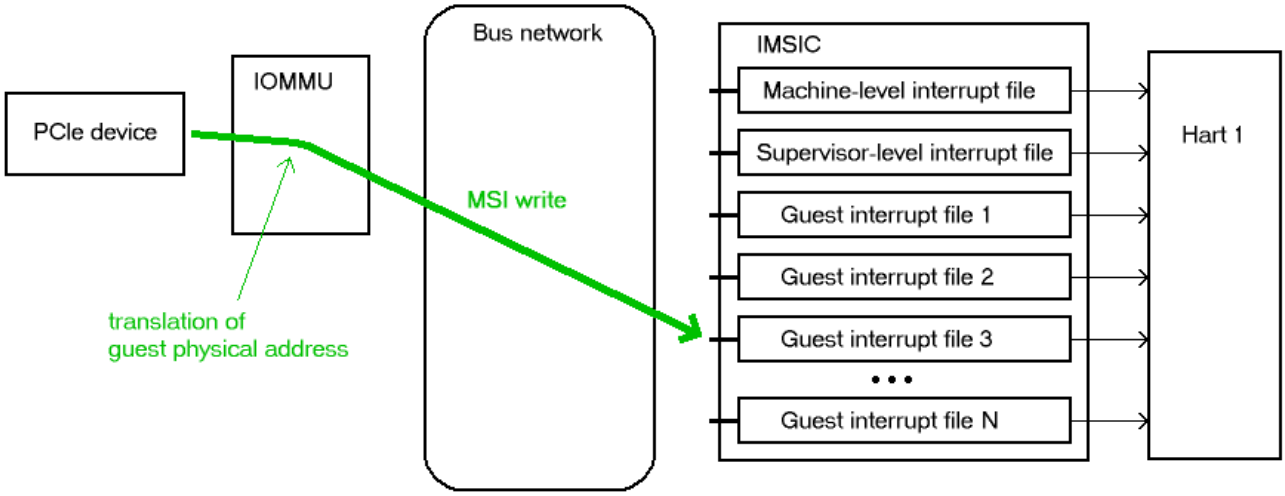

Figure 6. Translation of a device-sourced MSI that a guest OS intended to go to a (virtual) IMSIC interrupt file in the OS's virtual machine. Referencing an MSI page table supplied by the controlling hypervisor, the IOMMU redirects the MSI to a guest interrupt file of the real machine.

Memory writes from a device are recognized as MSIs by the destination address of the write. If an IOMMU determines that a 32-bit write is to the location of a (virtual) interrupt file in the relevant virtual machine, the write is considered an MSI within the VM, else not. The exact formula for recognizing MSIs is documented in Section 8.4.

Although the translation of MSIs is controlled by its own separate page table, the fact that MSI translations are at the same page granularity as regular RISC-V address translations implies that an address translation cache within an IOMMU requires little modification to also cache MSI translations. Only on a translation cache miss does the IOMMU need to treat MSIs significantly differently than other memory accesses from the same device, to choose the correct translation table and to access and interpret the table properly.

8.3. Memory-resident interrupt files

An IOMMU may optionally support memory-resident interrupt files (MRIFs). If implemented, the use of memory-resident interrupt files can greatly increase the number of virtual harts that can be given direct control of one or more physical devices in a system, assuming the rest of the system can still handle the added load.

Without memory-resident interrupt files, the number of virtual RISC-V harts that can directly receive MSIs from devices is limited by the total number of guest interrupt files implemented by all IMSICs in the system, because all MSIs to RISC-V harts must go through IMSICs. For a single RISC-V hart, the number of guest interrupt files is the *GEILEN* parameter defined by the H extension, which can be at most 31 for RV32 and 63 for RV64.

With the use of memory-resident interrupt files, on the other hand, the total number of virtual RISC-V harts able to receive device MSIs is almost unbounded, constrained only by the amount of real physical memory and the additional processing time needed to handle them. As its name implies, a memory-resident interrupt file is located in memory instead of within an IMSIC. Figure 7 depicts how an IOMMU can record an incoming MSI in an MRIF. When properly configured by a hypervisor, an IOMMU recognizes certain incoming MSIs as intended for a specific virtual interrupt file, and records each such MSI by setting an interrupt-pending bit stored within the MRIF data structure in ordinary

memory. After each MSI is recorded in an MRIF, the IOMMU also sends a *notice MSI* to the hypervisor to inform it that the MRIF contents may have changed.

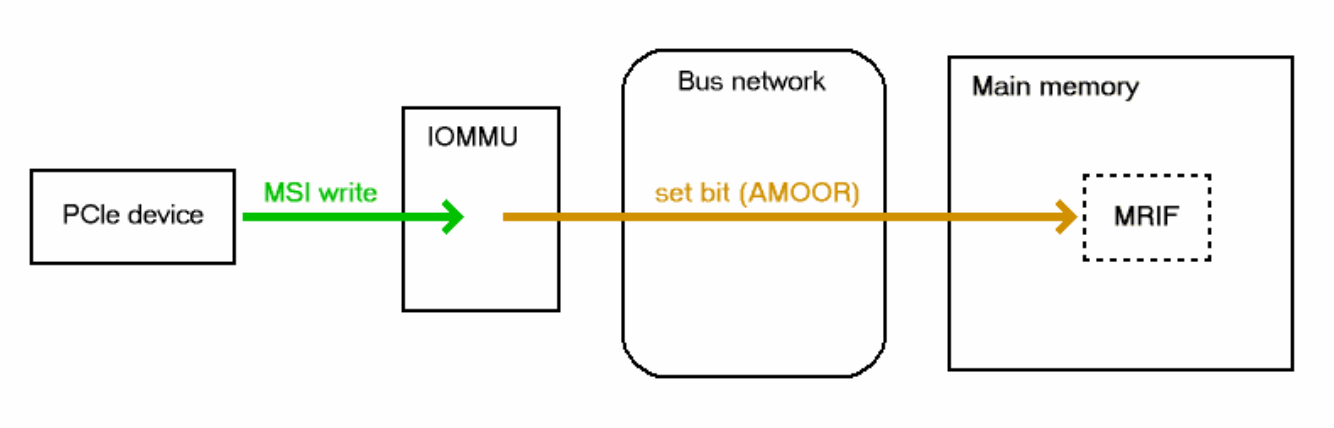

Figure 7. Recording an incoming MSI into a memory-resident interrupt file (MRIF) instead of sending it to a guest interrupt file as in Figure 6.

While a memory-resident interrupt file provides a place to record MSIs, it cannot interrupt a hart directly the way an IMSIC's guest interrupt files can. The notice MSIs that hypervisors receive only indicate that a virtual hart *might* need interrupting; a hypervisor is responsible for examining the MRIF contents each time to determine whether actually to interrupt the virtual hart. Furthermore, whereas an IMSIC's guest interrupt file can directly act as a supervisor-level interrupt file for a virtual hart, keeping a virtual hart's interrupt file in an MRIF while the virtual hart executes requires that the hypervisor emulate a supervisor-level interrupt file for the virtual hart, hiding the underlying MRIF. Depending on how often the virtual hart touches its interrupt file and the implementation's level of support for MRIFs, the cost of this emulation may be significant.

Consequently, MRIFs are expected most often to be used for virtual harts that are more-or-less "swapped out" of a physical hart due to being idle, or nearly so. When a hypervisor determines that an MSI that landed in an MRIF should wake up a particular virtual hart that was idle, the virtual hart can be assigned a guest interrupt file in an IMSIC and its interrupt file moved from the MRIF into this guest interrupt file before the virtual hart is resumed. The process of allocating a guest interrupt file for the newly wakened virtual hart may of course force the interrupt file of another virtual hart to be evicted to its own MRIF.

Not all systems need to accommodate large numbers of idle virtual harts. Many batch-processing servers, for example, strive to keep all virtual worker threads as busy as possible from start to finish, throttled only by I/O delays and limits on processing resources. In such environments, support for MRIFs may not be useful, so long as parameter GEILEN is not too small.

An IOMMU can have one of these three levels of support for memory-resident interrupt files:

- no memory-resident interrupt files;
- memory-resident interrupt files without atomic update; or
- memory-resident interrupt files with atomic update.

Memory-resident interrupt files are most efficient when the memory system supports logical atomic memory operations (AMOs) corresponding to RISC-V instructions AMOAND and AMOOR, for memory accesses made both from harts and from the IOMMU. The AMOAND and AMOOR operations are required for *atomic update* of a memory-resident interrupt file. A reduced level of support is possible without AMOs, relying solely on basic memory reads and writes.

8.3.1. Format of a memory-resident interrupt file

A memory-resident interrupt file occupies 512 bytes of memory, naturally aligned to a 512-byte address boundary. The 512 bytes are organized as an array of 32 pairs of 64-bit doublewords, 64 doublewords in all. Each doubleword is in little-endian byte order (even for systems where all harts are big-endian-only).

Big-endian-configured harts that make use of MRIFs are expected to implement the REV8 byte-reversal instruction defined by standard RISC-V extension Zbb, or pay the cost of endianness conversion using a sequence of instructions.

The pairs of doublewords contain the interrupt-pending and interrupt-enable bits for external interrupt identities 1-2047, in this arrangement:

```
offset   size   contents
 0x000  8 bytes interrupt-pending bits for (minor) identities 1-63
 0x008  8 bytes interrupt-enable bits for identities 1-63
 0x010  8 bytes interrupt-pending bits for identities 64-127
 0x018  8 bytes interrupt-enable bits for identities 64-127
  ...            ...
 0x1F0  8 bytes interrupt-pending bits for identities 1984-2047
 0x1F8  8 bytes interrupt-enable bits for identities 1984-2047
```

In general, the pair of doublewords at address offsets $k \times 16$ and $k \times 16+8$ for integer k contain the interrupt-pending and interrupt-enable bits for external interrupt minor identities in the range $k \times 64$ to $k \times 64+63$. For identity i in this range, bit $(i \bmod 64)$ of the first (even) doubleword is the interrupt-pending bit, and the same bit of the second (odd) doubleword is the interrupt-enable bit.

The interrupt-pending and interrupt-enable bits are stored interleaved by doublewords within an MRIF to facilitate the possibility of an IOMMU examining the relevant enable bit to determine whether to send a notice MSI after updating a pending bit, rather than the default behavior of always sending a notice MSI after an update without regard for the interrupt-enable bits. The memory arrangement matters only when MRIFs are supported without atomic update.

Bit 0 of the first doubleword of an MRIF stores a faux interrupt-pending bit for nonexistent interrupt 0. If a write from an I/O device appears to be an MSI that should be stored in an MRIF, yet the data to write (the interrupt identity) is zero, the IOMMU acts as though zero were a valid interrupt identity, setting bit 0 of the target MRIF's first doubleword and sending a notice MSI as usual.

All MRIFs are the size to accommodate 2047 valid interrupt identities, the maximum allowed for an IMSIC interrupt file. If a system's actual IMSICs have interrupt files that implement only N interrupt identities, $N<2047$, then the contents of MRIFs for identities greater than N may be ignored by software. IOMMUs, however, treat every MRIF as though all interrupt identities in the range 0-2047 are valid, even as software ignores invalid identity 0 and all identities greater than N.

There is no need to specify to an IOMMU a desired size N for an MRIF smaller than 2047 valid interrupt identities. The only use an IOMMU would make of this information would be to discard any MSIs indicating an interrupt identity greater than N. If devices are

> *properly configured by software, such errant MSIs should not occur; but even if they do, it is just as effective for software to ignore spurious interrupt identities after they have been recorded in an MRIF as for an IOMMU to discard them before recording them in the MRIF.*
>
> *It is likewise unnecessary for IOMMUs to check for and discard MSIs indicating an invalid interrupt identity of zero.*

8.3.2. Recording of incoming MSIs to memory-resident interrupt files

The data component of an MSI write specifies the interrupt identity to raise in the destination interrupt file. (Recall Section 3.2.) This data may be in little-endian or big-endian byte order. If an IOMMU supports memory-resident interrupt files, it can store to an MRIF MSIs of the same endianness that the machine's IMSICs accept. All IMSIC interrupt files are required to accept MSIs in little-endian byte order written to memory-mapped register `seteipnum_le` (Section 3.5). IMSIC interrupt files may also accept MSIs in big-endian byte order if register `seteipnum_be` is implemented alongside `seteipnum_le`.

If the interrupt identity indicated by an MSI's data (when interpreted in the correct byte order) is in the range 0-2047, an IOMMU stores the MSI to an MRIF by setting to one the interrupt-pending bit in the MRIF for that identity. If atomic update is supported for MRIFs, the pending bit is set using an AMOOR operation, else it is set using a non-atomic read-modify-write sequence. After the interrupt-pending bit is set in the MRIF, the IOMMU sends the notice MSI that software has configured for the MRIF.

The exact process of storing an MSI to an MRIF is specified more precisely in Section 8.5.2, which covers MSI page table entries configured in *MRIF mode*.

> *It is an open question whether an IOMMU might optionally examine the matching interrupt-enable bit within a destination MRIF to decide whether to send a notice MSI after setting an interrupt-pending bit. Currently, an IOMMU is required always to send a notice MSI after storing an MSI to an MRIF, even when the corresponding enable bit for the interrupt identity is zero.*

8.3.3. Use of memory-resident interrupt files with atomic update

To make use of a memory-resident interrupt file with support for atomic update, software must have memory locations to save an IMSIC interrupt file's `eidelivery` and `eithreshold` registers, in addition to the MRIF structure itself from Section 8.3.1.

Moving a virtual hart's interrupt file from an IMSIC into an MRIF involves these steps:

1. Prepare the MRIF by zeroing all of its interrupt-pending bits (the even doublewords) and by copying the IMSIC interrupt file's `eie` array to the MRIF's interrupt-enable bits (the odd doublewords).
2. Save to memory the existing values of the IMSIC interrupt file's registers `eidelivery` and `eithreshold`, and set `eidelivery` = 0.
3. Modify all relevant translation tables at IOMMUs so that MSIs for this virtual interrupt file are now stored in the MRIF. If necessary, synchronize with all IOMMUs to ensure that no straggler MSIs will arrive at the IMSIC interrupt file after this step.
4. Logically OR the contents of the IMSIC interrupt file's `eip` array into the interrupt-pending bits of

the MRIF, using AMOOR operations.

Once this sequence is complete, the IMSIC interrupt file is no longer in use.

Each time a notice MSI arrives indicating that an MSI has been stored in the MRIF, the controlling hypervisor should scan the MRIF's interrupt-pending and interrupt-enable bits to determine if any enabled interrupt is now both pending and enabled and thus should interrupt the virtual hart.

With atomic update of MRIFs, a virtual hart may continue executing with its interrupt file contained in an MRIF, so long as the hypervisor emulates for the virtual hart a proper IMSIC interrupt file to hide the underlying MRIF. Hypervisor software can safely set and clear the interrupt-pending and interrupt-enable bits of the MRIF using AMOOR and AMOAND operations, even as an IOMMU may be storing incoming MSIs into the same MRIF.

If an IOMMU is ever configured to examine an MRIF's interrupt-enable bits to decide whether to send notice MSIs, then modifying those enable bits will generally require coordination with the IOMMU. But so long as IOMMUs ignore the interrupt-enable bits as is currently assumed, the bits can be changed by software without risk.

To move the same interrupt file from the MRIF back to an IMSIC:

1. At the new IMSIC interrupt file, set **`eidelivery`** = 0, and zero the **`eip`** array.
2. Modify all relevant translation tables at IOMMUs so that MSIs for this virtual interrupt file are now sent to the IMSIC interrupt file. If necessary, synchronize with all IOMMUs to ensure that no straggler MSIs will be stored in the MRIF after this step.
3. Logically OR the interrupt-pending bits from the MRIF into the IMSIC interrupt file, using instruction CSRS to write to the **`eip`** array. Also, copy the interrupt-enable bits from the MRIF to the IMSIC interrupt file's **`eie`** array.
4. Load the IMSIC interrupt file's registers **`eithreshold`** and **`eidelivery`** with the values that were earlier saved.

8.3.4. Use of memory-resident interrupt files without atomic update

Without support for atomic update, the use of memory-resident interrupt files is similar to the atomic-update case of the previous subsection, but with some added complexities.

First, if the I/O devices that a virtual hart controls are behind multiple IOMMUs, then multiple MRIF structures are needed, one per IOMMU, not just a single MRIF structure. Furthermore, in addition to locations for storing **`eidelivery`** and **`eithreshold`**, software needs a place for a complete copy of the interrupt file's implemented **`eip`** array, apart from the MRIFs. While a virtual interrupt file is in memory, its interrupt-pending bits will be split across all the MRIFs and the saved **`eip`** array. The interrupt-enable bits may exist only in the MRIFs.

To move a virtual hart's interrupt file from an IMSIC into memory, with one MRIF per IOMMU:

1. Prepare all MRIFs by zeroing their interrupt-pending bits (the even doublewords) and by copying the IMSIC interrupt file's **`eie`** array to the MRIFs' interrupt-enable bits (the odd doublewords).
2. Save to memory the existing values of the IMSIC interrupt file's registers **`eidelivery`** and **`eithreshold`**, and set **`eidelivery`** = 0.
3. At each IOMMU, modify all relevant translation tables so that MSIs for this virtual interrupt file are now stored in the individual MRIF matched to the IOMMU. If necessary, synchronize with all

IOMMUs to ensure that no straggler MSIs will arrive at the IMSIC interrupt file after this step.

4. Dump the IMSIC interrupt file's `eip` array to its separate location outside the MRIFs.

Once this sequence is complete, the IMSIC interrupt file is no longer in use.

While a virtual hart's interrupt file remains in memory, an interrupt identity's true pending bit is the logical OR of its bit in all MRIFs and its bit in the saved `eip` array. All pending bits in the MRIFs start as zeros, but interrupts may become pending there as MSIs for this virtual hart arrive at IOMMUs and are stored in the corresponding MRIFs.

Without atomic update of MRIFs, an interrupt-pending bit is not easily cleared in an MRIF. (Clearing a single pending bit in one MRIF requires that a new MRIF be allocated and initialized and the corresponding IOMMU reconfigured to store MSIs into the new MRIF.) For this reason, it may or may not be practical to have a virtual hart execute while keeping one of its interrupt files in memory. When an MRIF records an interrupt that should wake a virtual hart, the simplest strategy is to always move the interrupt file back into an IMSIC's guest interrupt file before resuming execution of the virtual hart.

To transfer an interrupt file from memory back to an IMSIC:

1. At the new IMSIC interrupt file, set `eidelivery` = 0, and zero the `eip` array.
2. Modify all relevant translation tables at IOMMUs so that MSIs for this virtual interrupt file are now sent to the IMSIC interrupt file. If necessary, synchronize with all IOMMUs to ensure that no straggler MSIs will be stored in MRIFs after this step.
3. Merge by bitwise logical OR the interrupt-pending bits of all MRIFs and the saved `eip` array, and logically OR these merged bits into the IMSIC interrupt file, using instruction CSRS to write to the `eip` array. Also, copy the interrupt-enable bits from one of the MRIFs to the IMSIC interrupt file's `eie` array.
4. Load the IMSIC interrupt file's registers `eithreshold` and `eidelivery` with the values that were earlier saved.

8.3.5. Allocation of guest interrupt files for receiving notice MSIs

The processing a hypervisor does in response to notice MSIs can be minimized by assigning a separate interrupt identity for each MRIF, so the identity encoded in a notice MSI always indicates which one MRIF may have changed. However, if there are very many MRIFs (potentially in the thousands), a hypervisor may run short of interrupt identities within the supervisor-level interrupt files available in IMSICs. In that case, the hypervisor can increase its supply of interrupt identities by allocating one or more of the IMSICs' guest interrupt files to itself for the purpose of receiving notice MSIs.

Although guest interrupt files exist primarily to act as supervisor-level interrupt files for virtual harts, the IMSIC hardware does not police exactly how they are used by software.

8.4. Identification of page addresses of a VM's interrupt files

When an I/O device is configured directly by a guest operating system, MSIs from the device are expected to be targeted to virtual IMSICs within the guest OS's virtual machine, using guest physical addresses that are inappropriate and unsafe for the real machine. An IOMMU must recognize certain incoming writes from such devices as MSIs and convert them as needed for the real machine. (Recall

Figure 6.)

MSIs originating from a single device that require conversion are expected to have been configured at the device by a single guest OS running within one RISC-V virtual machine. Assuming the VM itself conforms to the Advanced Interrupt Architecture, MSIs are sent to virtual harts within the VM by writing to the memory-mapped registers of the interrupt files of virtual IMSICs. Each of these virtual interrupt files occupies a separate 4-KiB page in the VM's guest physical address space, the same as real interrupt files do in a real machine's physical address space. A write to a guest physical address can thus be recognized as an MSI to a virtual hart if the write is to a page occupied by an interrupt file of a virtual IMSIC within the VM.

The MSI address mask and address pattern specified in a device context (Section 8.1) are used to identify the 4-KiB pages of virtual interrupt files in the guest physical address space of the relevant VM. An incoming 32-bit write made by a device is recognized as an MSI write to a virtual interrupt file if the destination guest physical page matches the supplied address pattern in all bit positions that are zeros in the supplied address mask. In detail, a memory access to guest physical address A is an access to a virtual interrupt file's memory-mapped page if

((A >> 12) & ~address mask) = (address pattern & ~address mask)

where >> 12 represents shifting right by 12 bits, an ampersand (&) represents bitwise logical AND, and "~address mask" is the bitwise logical complement of the address mask.

When a memory access is found to be to a virtual interrupt file, an *interrupt file number* is extracted from the original guest physical address as

interrupt file number = extract(A >> 12, address mask)

Here, extract(x, y) is a "bit extract" that discards all bits from x whose matching bits in the same positions in the mask y are zeros, and packs the remaining bits from x contiguously at the least-significant end of the result, keeping the same bit order as x and filling any other bits at the most-significant end of the result with zeros. For example, if the bits of x and y are

x = a b c d e f g h

y = 1 0 1 0 0 1 1 0

then the value of extract(x, y) has bits 0 0 0 0 a c f g.

8.5. MSI page tables

When an IOMMU determines that a memory access is to a virtual interrupt file as specified in the previous section, the access is translated or converted by consulting the MSI page table configured for the device, instead of using the regular translation data structures that apply to all other memory accesses from the same device.

An MSI page table is a flat array of MSI page table entries (MSI PTEs), each 16 bytes. MSI page tables have no multi-level hierarchy like regular RISC-V page tables do. Rather, every MSI PTE is a leaf entry specifying the translation or conversion of accesses made to a particular 4-KiB guest physical page that a virtual interrupt file occupies (or may occupy) in the relevant virtual machine. To select an individual MSI PTE from an MSI page table, the PTE array is indexed by the interrupt file number extracted from the destination guest physical address of the incoming memory access by the formula of the previous section. Each MSI PTE may specify either the address of a real guest interrupt file that

substitutes for the targeted virtual interrupt file (as in Figure 6), or a memory-resident interrupt file in which to store incoming MSIs for the virtual interrupt file (as in Figure 7).

The number of entries in an MSI page table is 2^k where k is the number of bits that are ones in the MSI address mask used to extract the interrupt file number from the destination guest physical address. If an MSI page table has 256 or fewer entries, the start of the table is aligned to a 4-KiB page address in real physical memory. If an MSI page table has $2^k > 256$ entries, the table must be naturally aligned to a $2^k \times 16 - byte$ address boundary. If an MSI page table is not aligned as required, all entries in the table appear to an IOMMU as UNSPECIFED, and any address an IOMMU may compute and use for reading an individual MSI PTE from the table is also UNSPECIFIED.

Every 16-byte MSI PTE is interpreted as two 64-bit doublewords. If an IOMMU also references standard RISC-V page tables, defined by the RISC-V Privileged Architecture, for regular address translation, then the byte order for each of the two doublewords in memory, little-endian or big-endian, should be the same as the endianness of the regular RISC-V page tables configured for the same device context. Otherwise, the endianness of the doublewords of an MSI PTE is implementation-defined.

Bit 0 of the first doubleword of an MSI PTE is field V (Valid). When V = 0, the PTE is invalid, and all other bits of both doublewords are ignored by an IOMMU, making them free for software to use.

If V = 1, bit 63 of the first doubleword is field C (Custom), designated for custom use. If an MSI PTE has V = 1 and C = 1, interpretation of the rest of the PTE is implementation-defined.

If V = 1 and the custom-use bit C = 0, then bits 2:1 of the first doubleword contain field M (Mode). If M = 3, the MSI PTE specifies *basic translate mode* for accesses to the page, and if M = 1, it specifies *MRIF mode*. Values of 0 and 2 for M are reserved. The interpretation of an MSI PTE for each of the two defined modes is detailed further in the next two subsections.

8.5.1. MSI PTE, basic translate mode

When an MSI PTE has fields V = 1, C = 0, and M = 3 (basic translate mode), the PTE's complete format is:

```
First doubleword:  bit 63     C, = 0
                   bits 53:10 PPN
                   bits 2:1   M, = 3
                   bit 0      V, = 1
Second doubleword: ignored
```

All other bits of the first doubleword are reserved and must be set to zeros by software. The second doubleword is ignored by an IOMMU so is free for software to use.

A memory access within the page covered by the MSI PTE is translated by replacing the access's original address bits 12 and above (the guest physical page number) with field PPN (Physical Page Number) from the PTE, while retaining the original address bits 11:0 (the page offset). This translated address is either zero-extended or clipped at the upper end as needed to make it the width of a real physical address for the machine. The original memory access from the device is then passed onward to the memory system with the new address.

An MSI PTE in basic translate mode allows a hypervisor to route an MSI write intended for a virtual

interrupt file to go instead to a guest interrupt file of a real IMSIC in the machine.

An IOMMU that also employs standard RISC-V page tables for regular address translation can maximize the overlap between the handling of MSI PTEs and regular RISC-V leaf PTEs as follows:

For RV64, the first doubleword of an MSI PTE in basic translate mode has the same encoding as a regular RISC-V leaf PTE for Sv39, Sv48, Sv57, Sv39x4, Sv48x4, or Sv57x4 page-based address translation, with PTE fields D, A, G, U, and X all zeros and W = R = 1. Hence, the MSI PTE's first doubleword appears the same as a regular PTE that grants read and write permission (R = W = 1) but not execute permissions (X = 0). This same-encoded regular PTE would translate an MSI write the same as the actual MSI PTE, except that what would be the PTE's accessed (A), dirty (D), and user (U) bits are all zeros. An IOMMU needs to treat only these three bits differently for an MSI PTE versus a regular RV64 leaf PTE.

The address computation used to select a PTE from a regular RISC-V page table must be modified to select an MSI PTE's first doubleword from an MSI page table. However, the extraction of an interrupt file number from a guest physical address to obtain the index for accessing the MSI page table already creates an unavoidable difference in PTE addressing.

For RV32, the lower 32-bit word of an MSI PTE's first doubleword has the same format as a leaf PTE for Sv32 or Sv32x4 page-based address translation, except again for what would be PTE bits A, D, and U, which must be treated differently.

8.5.2. MSI PTE, MRIF mode

If memory-resident interrupt files are supported and an MSI PTE has fields V = 1, C = 0, and M = 1 (MRIF mode), the PTE's complete format is:

```
First doubleword:  bit 63     C, = 0
                   bits 53:7  MRIF Address[55:9]
                   bits 2:1   M, = 1
                   bit 0      V, = 1
Second doubleword: bit 60     NID[10]
                   bits 53:10 NPPN
                   bits 9:0   NID[9:0]
```

All other PTE bits are reserved and must be set to zeros by software.

The PTE's MRIF Address field provides bits 55:9 of the physical address of a memory-resident interrupt file in which to store incoming MSIs, referred to as the *destination MRIF*. As every memory-resident interrupt file is naturally aligned to a 512-byte address boundary, bits 8:0 of the destination MRIF's address must be zero and are not specified in the PTE.

Field NPPN (Notice Physical Page Number) and the two NID (Notice Identifier) fields together specify a destination and value for a *notice MSI* that is sent after each time the destination MRIF is updated as a result of consulting this PTE to store an incoming MSI.

Typically, NPPN will be the page address of an IMSIC's interrupt file in the real machine, and NID will be the interrupt identity to make pending in that interrupt file to indicate that the destination MRIF may have changed. However, NPPN is not required to be a valid interrupt file address, and an IOMMU must not attempt to restrict it to only such addresses. Any page address must be accepted for NPPN.

Memory accesses by I/O devices to addresses within a page covered by an MRIF-mode PTE are handled by the IOMMU instead of being passed through to the memory system. If a memory access, read or write, is not for 32 bits of data, or if the access address is not aligned to a 4-byte boundary (including accesses that straddle the page boundary), the access should be aborted as unsupported. For a naturally aligned 32-bit read, the IOMMU should preferably return zero as the read value but may alternatively abort the access. A naturally aligned 32-bit write is either interpreted as an MSI, resulting in an update of the destination MRIF, or is discarded.

When the IMSIC interrupt files in the system implement memory-mapped register `seteipnum_be` for receiving MSIs in big-endian byte order (Section 3.5), then an IOMMU must be able to store MSIs in both little-endian and big-endian byte orders to the destination MRIF. If the IMSIC interrupt files in the system do not implement register `seteipnum_be`, an IOMMU should ordinarily store only little-endian MSIs to the destination MRIF. The data of an incoming MSI is assumed to be in little-endian byte order if bit 2 of the destination address is zero, and in big-endian byte order if bit 2 of the destination address is one.

If a naturally aligned 32-bit write is to guest physical address A within a page covered by an MRIF-mode PTE, and if the write data is D when interpreted in the byte order indicated by bit 2 of A, then the write is processed as follows: If either A[11:3] or D[31:11] is not zero, or if bit 2 of A is one and big-endian MSIs are not supported, then the incoming write is accepted but discarded. Else, the original incoming write is recognized as an MSI and is replaced by one of the following memory accesses, setting the interrupt-pending bit that corresponds to the interrupt identity D in the destination MRIF to one:

- an atomic AMOOR operation, if atomic updates are supported; or
- a non-atomic read-modify-write sequence, if atomic updates are not supported.

Once the MRIF update operation is visible to all agents in the system, the 11-bit NID value is zero-extended to 32 bits, and this value is written to the address NPPN<<12 (i.e., physical page number NPPN, page offset zero) in little-endian byte order.

While IOMMUs are expected typically to cache MSI PTEs that are configured in basic translate mode (M = 3), they might not cache PTEs configured in MRIF mode (M = 1). Two reasons together justify not caching MSI PTEs in MRIF mode: First, the information and actions required to store an MSI to an MRIF are far different than normal address translation; and second, by their nature, MSIs to MRIFs should occur less frequently. Hence, an IOMMU might perform MRIF-mode processing solely as an extension of cache-miss page table walks, leaving its address translation cache oblivious to MRIF-mode MSI PTEs.