RISC-V S-level Physical Memory Protection (SPMP)

Editor - Dong Du, Bicheng Yang, RISC-V SPMP Task Group

Version 1.0.0-rc5, 11/2025: This document is in development. Assume everything can change. See http://riscv.org/spec-state for details.

Table of Contents

Preamble

This document is in the Development state

Assume everything can change. This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Copyright and license information

Contributors

The proposed SPMP specifications (non-ratified, under discussion) has been contributed to directly or indirectly by:

- Dong Du, Editor <dd_nirvana@sjtu.edu.cn>
- Bicheng Yang, Editor <bichengyang@sjtu.edu.cn>
- José Martins
- Thomas Roecker
- Sandro Pinto
- Manuel Rodriguez
- Luís Cunha
- Rongchuan Liu
- Nick Kossifidis
- Andy Dellow
- Manuel Offenberg
- Allen Baum
- Bill Huffman
- Xu Lu
- Wenhao Li
- Yubin Xia
- Joe Xie
- Paul Ku
- Jonathan Behrens
- Robin Zheng
- Zeyu Mi
- Fabrice Marinet
- Noureddine Ait Said
- Xi Zhao

Chapter 1. Introduction

This document introduces the RISC-V S-level Physical Memory Protection (SPMP) proposal. SPMP is designed to enforce memory isolation in environments where virtual memory, managed by a Memory Management Unit (MMU), is not employed. Processors based on the RISC-V architecture are gaining significant traction in the burgeoning Internet of Things (IoT) and automotive sectors. In these domains, however, virtual memory is frequently avoided to comply with stringent constraints on system resources and execution latency. Consequently, in the absence of an MMU, effectively isolating S-mode operating systems (such as an RTOS) from U-mode applications presents a significant challenge. The SPMP extension adopts an architecture and programming model analogous to the existing Physical Memory Protection (PMP) standard. It equips an S-mode OS with the capability to guarantee secure processing and fault isolation for U-mode software. This protection is achieved by restricting the physical memory regions that U-mode software can access on a specific hart.

Chapter 2. "Smpmpdeleg" Extension for Sharing Hardware Resources between PMP and SPMP

The similar architecture of PMP and SPMP registers, including their shared address-matching logic, makes hardware reuse a practical approach for resource conservation. This chapter introduces the `Smpmpdeleg` extension, a mechanism that allows hardware resources to be dynamically allocated between PMP and SPMP.

This extension is mandatory for implementations that support Sspmp (Chapter 3) in conjunction with M-mode (i.e., `Sm1p13`). To streamline the specification and reduce optional features, the `Smpmpdeleg` extension mandates a total of 64 PMP entries. However, an implementation retains the flexibility to provide fewer physical entries; any unimplemented entries behave as read-only zero.

2.1. Resource Sharing between PMP and SPMP

The `Smpmpdeleg` extension facilitates the delegation of PMP entries for use by S-level, thereby creating S-level PMP (SPMP) entries. This delegation is managed by an MXLEN-bit M-mode CSR named `mpmpdeleg`, whose layout is detailed in Figure 1.

1. A key component of this CSR is the `pmpnum` WARL field, which defines the starting index for delegation. All PMP entries with an index equal to or greater than `pmpnum` are delegated as SPMP entries.
2. If a write to `pmpnum` specifies a value exceeding the number of physically implemented PMP entries, the field subsequently reads back the total count of implemented entries.
3. Setting `pmpnum` to zero delegates all PMP entries to SPMP, while setting it to the total number of entries delegates none.
4. By default, unless hardwired, `pmpnum` resets to the total number of implemented PMP entries.
5. If no entries are delegated to SPMP, the `Sspmp` extension is effectively disabled, and any attempt to access SPMP-related registers results in reads returning zero, and writes being ignored.

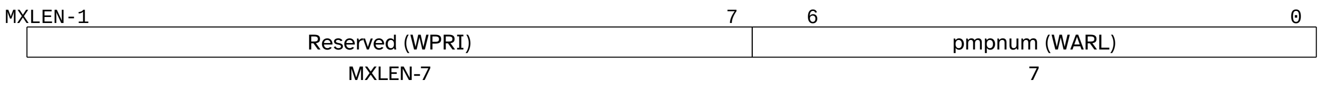


Figure 1. mpmpdeleg CSR format.

The `mpmpdeleg.pmpnum` field is a WARL field, which allows an implementation to hardwire the partition between PMP and SPMP.

Addressing:

Both PMP and SPMP entries are indexed starting from zero. For example, in an implementation with 64 total entries where `pmpnum` is configured to 16:

1. PMP entries `0` through 15 act as PMP (i.e., `PMP[0..15]`) and are accessible via standard PMP CSRs (i.e., `pmpcfg[0..3]` and `pmpaddr[0..15]` for RV32; `pmpcfg[0,2]` and `pmpaddr[0..15]` for RV64).
2. The remaining 48 entries are delegated as SPMP (i.e., `SPMP[0..47]`) and are indirectly accessed via `xiselect` (see Section 2.2 and Section 3.7).
3. Accesses to out-of-range indices, such as reading `PMP[16]` or writing to `SPMP[48]` in this scenario, are handled as follows: reads return zero, and writes are ignored.

Reconfiguration:

1. M-mode software can dynamically adjust the allocation between PMP and SPMP by writing to the `mpmpdeleg` CSR.
2. The `pmpnum` value cannot be set to an index that is less than or equal to that of any locked PMP entry. For instance, if `PMP[7]` is locked, any attempt to write a value less than 8 to `pmpnum` is ignored, and the field retains its prior value.
3. The `pmpnum` value can be set to override locked SPMP entries. For example, if `SPMP[0]` is locked, M-mode software can still increment `pmpnum`.

2.2. The Access Methods for SPMP CSRs in M-mode

M-mode employs different methods to access PMP and SPMP entries. PMP entries are accessed directly through their dedicated CSRs (i.e., `pmpcfg` and `pmpaddr`). Delegated SPMP entries, however, are accessed indirectly using the `xiselect` CSR (i.e., `siselect` and `miselect`).

For these indirect accesses, `miselect` selects the target SPMP entry, `mireg` accesses its `spmpaddr` register, and `mireg2` accesses its `spmpcfg` register. The `mireg3` through `mireg6` are read-only `0`.

The lock bit (`spmpcfg[i].L`) of an SPMP entry can only be cleared by the execution environment (M-mode in this case) through an indirect access using `miselect`.

The view provided by `miselect` is identical to that of `siselect` (see Section 3.7). For instance, if 48 out of 64 entries are delegated, both S-mode (via `siselect` indices `0`-47) and M-mode (via `siselect` or `miselect` indices `0`-47) can access `SPMP[0..47]`. Any access attempt by either mode to an index outside this range (i.e., `i >= 48`) results in a read of zero, and writes are ignored.

`miselect` **value**	**indirect CSR access of** `mireg`
`0x100`	`mireg` → `spmpaddr[0]`, `mireg2` → `spmpcfg[0]`
`0x101`	`mireg` → `spmpaddr[1]`, `mireg2` → `spmpcfg[1]`
…	…
`0x13F`	`mireg` → `spmpaddr[63]`, `mireg2` → `spmpcfg[63]`

Indirect accesses to SPMP CSRs are not ordered with respect to each other or with subsequent memory accesses. To enforce ordering, software must execute an `SFENCE.VMA` instruction with `rs1=x0` and `rs2=x0`, which synchronizes subsequent memory accesses with all preceding SPMP CSR writes.

Allowing indirect accesses to SPMP CSRs to be not ordered with respect to each other or to subsequent memory accesses enables fast context switching of SPMP registers. While `SFENCE.VMA` *instructions normally order preceding stores and subsequent implicit accesses to memory management structures, SPMP entries are also effectively regarded as memory management structures.*

Chapter 3. "Sspmp" Extension for S-level Physical Memory Protection (SPMP)

The RISC-V S-level Physical Memory Protection (SPMP) mechanism provides per-hart control registers in supervisor mode. These registers define physical memory access privileges—read, write, and execute—for discrete physical memory regions.

A memory access is successful only when permission checks for both PMP/ePMP and SPMP pass. SPMP checks can be performed by the hardware in parallel with PMA and PMP checks. SPMP exceptions take priority over PMP or PMA exceptions. Consequently, if a memory access violates both SPMP and PMP/PMA rules, only the SPMP exception is reported.

SPMP checks are enforced on all memory accesses with effective privilege modes less privileged than M-mode. SPMP can be configured to grant permissions to U-mode, which has none by default, and to revoke permissions from S-mode.

If the Hypervisor Extension is implemented in conjunction with Sspmp, when V = 1 and hgatp.MODE = Bare, SPMP enforces access checks on all memory accesses from VS and VU-modes, effectively applying SPMP protections to guest execution contexts. Additionally, the Hypervisor Virtual Machine Load and Store instructions (HLV, HLVX, HSV), when executed from M-mode, HS-mode, or U-mode (when hstatus.HU = 1), are also subject to SPMP checks under the V = 1 condition.

3.1. SPMP and Paging

SPMP and paged virtual memory are mutually exclusive and cannot be enabled concurrently for two primary reasons:

1. Enabling both introduces a redundant layer of permission checks for each memory access.
2. Paged virtual memory, by itself, offers a sufficient level of protection.

The following table dictates which isolation mechanism is active based on the satp configuration, assuming the hardware implements both.

satp	Isolation mechanism
satp.mode == Bare	SPMP only
satp.mode != Bare	Paged Virtual Memory only

3.2. Extension Dependencies

1. The `Sscsrind` extension for indirect CSR access must be implemented.
2. The `sstatus.SUM` (permit Supervisor User Memory access) bit must be **writable**, deviating from the Privileged Architecture (i.e., `Svbare`). In SPMP, this writability is essential because the bit alters how S-mode load and store operations access user memory.
3. The `sstatus.MXR` (Make eXecutable Readable) bit must be **writable**, deviating from the Privileged Architecture (i.e., `Svbare`). This writability supports M-mode emulation handlers that require reading instructions with `MXR=1 and MPRV=1`.
4. The Sspmp extension can be virtualized. If the Hypervisor Extension is implemented in conjunction with Sspmp, the only mandatory translation mode in both hgatp and satp is Bare.

3.3. S-level Physical Memory Protection CSRs

Each SPMP entry is composed of an SXLEN-bit configuration register and an associated SXLEN-bit address register. From S-mode, the SPMP registers are accessible only via the indirect access mechanism (see Section 3.7). In TOR mode, an entry also uses the address register of the preceding entry to define its range (see Section 3.5). Implementations support up to 64 SPMP entries.

> *An SPMP entry refers to the register pair* `spmpcfg[i]` *and* `spmpaddr[i]`.
>
> *An SPMP rule is defined by the contents of an* `spmpcfg` *register and its corresponding* `spmpaddr` *register(s). These registers collectively define a protected physical memory region and its access constraints.*
>
> *For a rule to be valid, its* `spmpcfg[i].A` *field must not be OFF. Furthermore, if* `spmpcfg[i].A` *is set to TOR, the condition* `spmpaddr[i-1] < spmpaddr[i]` *must hold. Particularly, if* `spmpcfg[0].A` *is set to TOR, zero is used for the lower bound.*

The SPMP address registers, named `spmpaddr0` through `spmpaddr63`, share the same layout as PMP architecture. On RV32 systems, each `spmpaddr` register encodes a 34-bit physical address from bit 33 down to bit 2, as illustrated in Figure 2. On RV64 systems, each `spmpaddr` register encodes a 56-bit physical address from bit 55 down to bit 2, as shown in Figure 3. An implementation may support fewer address bits, particularly on systems with a smaller physical address space. All writable SPMP entries should implement a consistent number of address bits. Since not all physical address bits must be implemented, the SPMP address registers are considered WARL, with exceptions defined by granularity rules. Refer to the "RISC-V Privileged Architecture", Section 3.7: Physical Memory Protection, Address Matching.

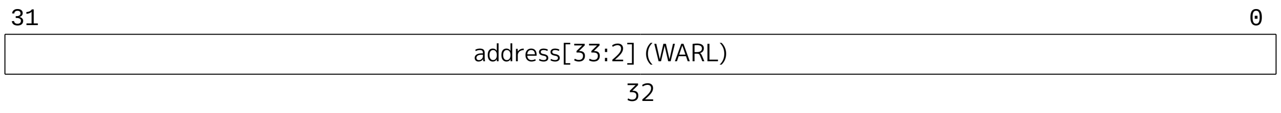


Figure 2. SPMP address register format, RV32.

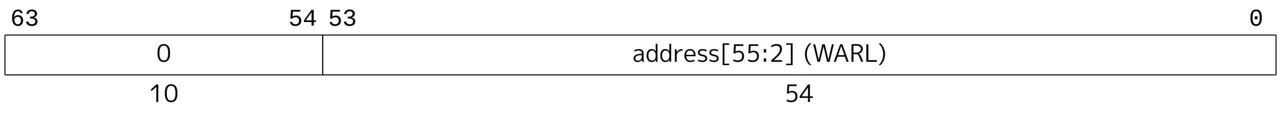


Figure 3. SPMP address register format, RV64.

Every SPMP entry contains an SXLEN-bit configuration register, `spmpcfg[i]`. Its lower 8 bits are an alias for the 8-bit field of the corresponding PMP configuration register. Figure 4 illustrates the layout of `spmpcfg[i]`. Permission rules and their encodings are detailed in Section 3.4.

1. The R, W, and X bits govern permissions for read, write, and instruction execution, respectively.
2. The A field, which defines the address-matching mode, is detailed in Section 3.5.
3. Bits 5 and 6 are reserved for future standard use.
4. The L (lock) bit designates an entry as locked. Setting the L bit locks the SPMP entry, regardless of the A field’s setting (even OFF). Attempts to write to locked `spmpcfg[i]` and `spmpaddr[i]` registers using the `siselect` CSR are ignored, irrespective of privilege level. If a locked entry has `spmpcfg[i].A` set to TOR, writes to the preceding `spmpaddr[i-1]` via `siselect` are also ignored.
5. If locking is not a required feature, an implementation can hardwire the L bit to `0`.
6. For any rule not designated as a `Shared-Region`, the U bit determines if it is `U-mode` (when set) or `S-`

mode-only (when clear), as explained in Section 3.4.

7. The SHARED bit identifies a rule as a `Shared-Region` rule.

SXLEN-1 10	9	8	7	6 5	4 3	2	1	0
Reserved (WPRI)	SHARED (WARL)	U (WARL)	L (WARL)	Reserved (WPRI)	A (WARL)	X (WARL)	W (WARL)	R (WARL)
SXLEN-10	1	1	1	2	2	1	1	1

Figure 4. SPMP configuration register format.

M-mode can leverage the L bit to create a sandbox for S-mode software. This is achieved by setting and locking high-priority SPMP entries where `spmpcfg[i].U` *is 1. This mechanism effectively thwarts privilege escalation attacks that might try to reconfigure SPMP entries to bypass S-mode restrictions. While PMP/ePMP entries could offer a similar function, the resulting configuration is not identical because PMP does not distinguish between S-mode and U-mode. Moreover, if resource sharing is statically defined (e.g.,* `mpmpdeleg.pmpnum` *is hardwired, see Section 2.1), there might not be enough PMP/ePMP entries to enforce the intended isolation policy.*

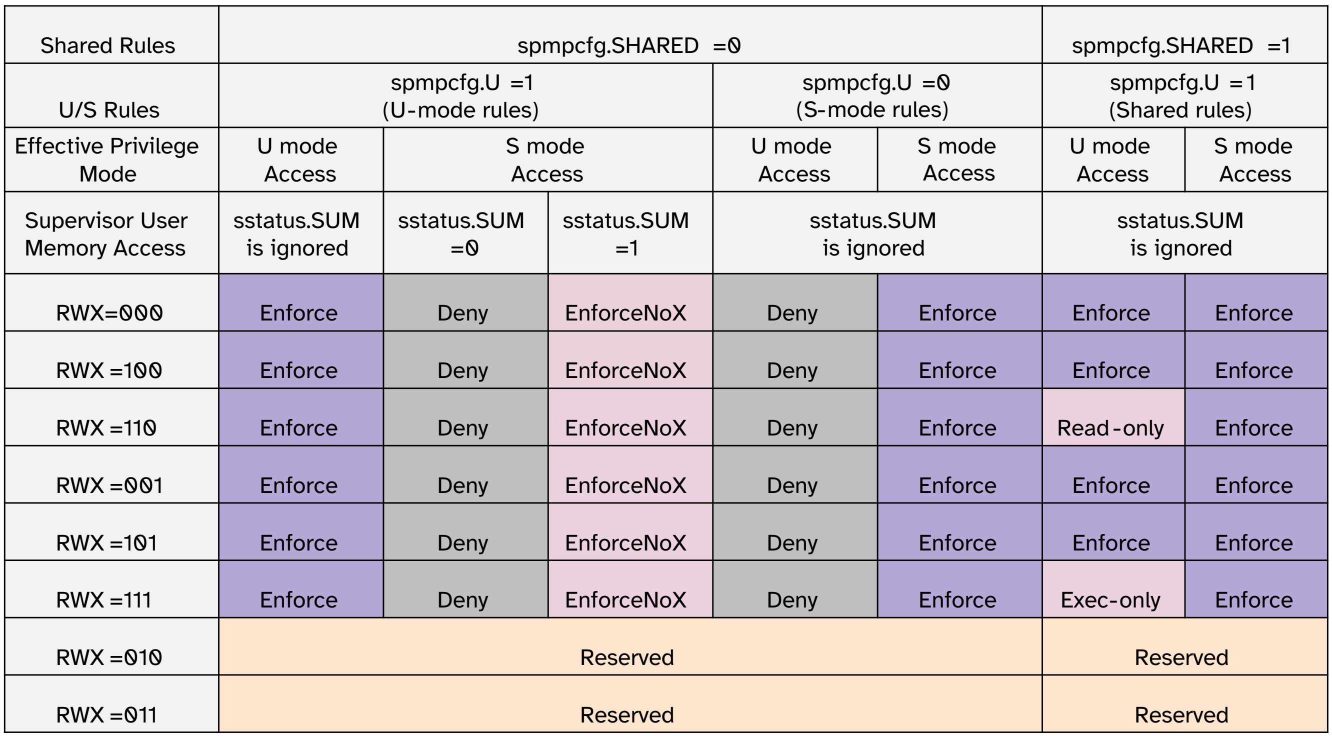
3.4. Encoding of Permissions

SPMP supports three distinct rule types: **S-mode-only**, **U-mode** and **Shared-Region**.

1. An **S-mode-only** rule is **enforced** for accesses from Supervisor mode and **denied** for accesses from User mode.
2. A **U-mode** rule is always enforced for User mode accesses. Its behavior for Supervisor mode accesses depends on the `sstatus.SUM` bit.
   - With `sstatus.SUM` set, the rule is enforced for Supervisor mode data accesses, but execution permission is denied (termed **EnforceNoX** in Figure 5). This prevents the OS from executing the memory of an unprivileged process at all times.
   - With `sstatus.SUM` clear, the rule is denied for any Supervisor mode access. This prevents the OS from accessing the memory of an unprivileged process unless a specific code path is followed.
3. The encoding `spmpcfg.SHARED == 1 and spmpcfg.U == 1` defines a **Shared-Region** rule. For shared regions, the state of the `sstatus.SUM` bit is irrelevant.
4. A **Shared-Region** rule is **enforced** for both Supervisor and User modes, but it imposes the constraint that read and write permissions for User mode are mutually exclusive.
5. The R, W, and X bits collectively form a WARL field.
6. The encoding `spmpcfg.RWX=010`, `spmpcfg.RWX=011`, and the combination `spmpcfg.SHARED == 1 and spmpcfg.U == 0` are all reserved for future standard use.

If the Hypervisor Extension is implemented, when V = 1 and hgatp.MODE = Bare, the permission encodings remain consistent with those when spmpcfg.U = 1, but are applied to VS/VU rather than U-mode accesses.

The complete table of encodings and their outcomes is presented in Figure 5:



Shared Rules	spmpcfg.SHARED =0					spmpcfg.SHARED =1	
U/S Rules	spmpcfg.U =1 (U-mode rules)			spmpcfg.U =0 (S-mode rules)		spmpcfg.U =1 (Shared rules)	
Effective Privilege Mode	U mode Access	S mode Access		U mode Access	S mode Access	U mode Access	S mode Access
Supervisor User Memory Access	sstatus.SUM is ignored	sstatus.SUM =0	sstatus.SUM =1	sstatus.SUM is ignored		sstatus.SUM is ignored	
RWX=000	Enforce	Deny	EnforceNoX	Deny	Enforce	Enforce	Enforce
RWX =100	Enforce	Deny	EnforceNoX	Deny	Enforce	Enforce	Enforce
RWX =110	Enforce	Deny	EnforceNoX	Deny	Enforce	Read-only	Enforce
RWX =001	Enforce	Deny	EnforceNoX	Deny	Enforce	Enforce	Enforce
RWX =101	Enforce	Deny	EnforceNoX	Deny	Enforce	Enforce	Enforce
RWX =111	Enforce	Deny	EnforceNoX	Deny	Enforce	Exec-only	Enforce
RWX =010	Reserved					Reserved	
RWX =011	Reserved					Reserved	

Figure 5. SPMP Encoding Table

Deny: The memory access is blocked and an exception is raised.

Enforce: The R/W/X permissions defined in `spmpcfg` are applied to the access.

EnforceNoX: The R/W permissions from `spmpcfg` are applied, but execute permission is denied.

Reserved: This encoding is reserved for future standard use.

SUM bit: The SPMP mechanism uses the `sstatus.SUM` (permit Supervisor User Memory access) bit to alter the privilege of S-mode load and store operations on physical memory. The semantics of `sstatus.SUM` within SPMP are consistent with its definition in the Machine-Level ISA (for details, refer to the "Memory Privilege in mstatus Register" subsection of the "RISC-V Privileged Architecture").

3.5. Address Matching

The A field within an SPMP entry's configuration register determines the address-matching mode for its associated spmpaddr register. The following table details the A field's encoding.

spmpcfg[i].A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥8 bytes

This encoding is consistent with the PMP/ePMP. For comprehensive details, refer to the "Address Matching" subsection for PMP in the "RISC-V Privileged Architecture".

Software can probe the minimum SPMP granularity. This is done by clearing `spmpcfg[i]`, *writing all ones to* `spmpaddr[i]`, *and then reading back the resulting* *`spmpaddr[i]` value. If G is the index of the least-significant bit set in the result, the granularity is 2^{G+2} bytes.*

Software can also determine the implemented physical address bits in `spmpaddr`. This involves setting `spmpcfg[i].A` to `0b11`, writing all ones to `spmpaddr[i]`, and reading back the result. (Consult the "NAPOT range encoding in PMP address and configuration registers" table in the "RISC-V Privileged Architecture" for interpretation.)

Because the `spmpcfg[i].A` field is WARL, an implementation is free to hardwire a specific address-matching mode.

3.6. Matching Logic

1. SPMP entries are statically prioritized.
2. The lowest-numbered SPMP entry that matches any byte of an access (indicated by an address and the accessed length) determines whether that access is allowed or denied.
3. This matching SPMP entry must match **all** bytes of the access, or the access fails and an instruction, load, or store page-fault exception is generated (see Section 3.8).
4. This matching is done irrespective of the SHARED, U, R, W, and X bits.
5. If the effective privilege mode of the access is M, the access is `allowed`.
6. If the effective privilege mode of the access is S/U and no SPMP entry matches, but at least one SPMP entry is implemented, the access is `denied`.
7. Otherwise, each access is checked according to the permission bits in the matching SPMP entry. That access is allowed if it satisfies the permission checking with the encoding corresponding to the access type.

Certain implementations may decompose misaligned memory operations into multiple accesses. In such cases, some of these sub-accesses might succeed before another triggers an exception. Notably, a portion of a misaligned store that passes an SPMP check could become architecturally visible, even if another portion of the same store fails. This behavior may also occur for stores wider than XLEN (e.g., FSD instruction in RV32D), even if the store address is naturally aligned.

SPMP rules are checked for all memory accesses, both implicit and explicit, that originate from S-mode or any less-privileged mode.

The execution environment is expected to configure one or more SPMP entries to grant S-mode its necessary baseline permissions. S-mode software can subsequently constrain these permissions by refining the SPMP entries.

3.7. The Access Method for SPMP CSRs in S-mode

Each value of `siselect` maps to a corresponding set of SPMP CSRs. `sireg` is used to access the `spmpaddr` register, while `sireg2` is used for the `spmpcfg` register. The registers `sireg3` through `sireg6` are read-only `0`.

SPMP entries are indexed starting from zero. In a system with 48 SPMP entries, S-mode can address `SPMP[0..47]` using `siselect#0..47`. An access to an out-of-bounds index via `siselect` will return zero on read and be ignored on write.

`siselect` **value**	**indirect CSR access of** `sireg`
`0x100`	`sireg` → `spmpaddr[0]`, `sireg2` → `spmpcfg[0]`
`0x101`	`sireg` → `spmpaddr[1]`, `sireg2` → `spmpcfg[1]`
…	…
`0x13F`	`sireg` → `spmpaddr[63]`, `sireg2` → `spmpcfg[63]`

The rationale for disallowing `siselect` writes to clear a lock bit is to isolate this capability within the `miselect` CSR space, rather than merely differentiating by privilege mode.

The design choice to map only one SPMP entry per `siselect` value is motivated by performance. Mapping multiple entries would necessitate a jump table or extra logic to select the correct target `sireg` register, adding overhead.

When `spmpcfg1` is written to via `siselect` and `sireg2`, the full SXLEN-bit value is written right-justified, even though the hardware may store the upper bits and lower 8 bits in separate register arrays.

For further details on indirect CSR access, refer to the `Sscsrind` extension specification in the "RISC-V Privileged Architecture".

3.8. Exceptions

A failed SPMP check triggers an exception whose type corresponds to the memory operation (load, store/AMO, or instruction fetch). Each fault type is assigned a distinct exception code.

The `Sspmp` extension co-opts the existing page fault exception codes for SPMP violations, as page fault handling is typically already delegated to S-mode. S-mode software (e.g., an OS) can differentiate between an SPMP fault and a standard page fault by querying the `satp.mode` field, given that SPMP and paging are mutually exclusive (see Section 3.1).

It is important to note that a single instruction can result in multiple memory accesses that are not guaranteed to be atomic relative to each other.

If the Hypervisor Extension is implemented, when a VS/VU access is denied by SPMP, an exception is raised. The type of exception depends on the nature of the access attempt, i.e., whether it is a load, store/AMO, or instruction fetch. SPMP reuses the guest page fault exception codes (20, 21, and 23, for instruction, load, and store/AMO accesses, respectively) defined by the Hypervisor extension.

Table of exception codes:

Interrupt	**Exception Code**	**Description**
0	12	Instruction page fault
0	13	Load page fault
0	15	Store/AMO page fault
0	20	Instruction guest-page fault
0	21	Load guest-page fault

Interrupt	Exception Code	Description
0	23	Store/AMO guest-page fault

Since, when V = 1, SPMP is only active when hgatp.MODE = Bare, SPMP protection and G-stage translation are mutually exclusive. Consequently, when SPMP is active, paged virtual memory translations are disabled, and guest page fault exception codes can be repurposed to indicate guest SPMP access violations.

Chapter 4. "Sspmpsw" Extension for Optimizing Context Switching of SPMP Entries

In RV64, a context switch for the SPMP mechanism involves updating as many as 64 address registers and configuration registers. The `Sspmpsw` extension, introduced in this chapter, is an optional feature designed to enhance the performance of SPMP context switches.

- For RV64 architectures, it introduces a 64-bit WARL CSR, `sspmpswitch`.
- For RV32 architectures, it introduces an additional 32-bit WARL CSR, `sspmpswitchh`, which serves as an alias for the upper 32 bits of the `sspmpswitch` register.

The activation of each SPMP entry is governed by its corresponding bit in the `sspmpswitch` register. An SPMP entry `i` is considered active only when both the `sspmpswitch[i]` bit is set and the `spmpcfg[i].A` field is enabled. The formal condition for this activation is `sspmpswitch[i] & spmpcfg[i].A != 0`.

When an entry `i` is locked, as indicated by `spmpcfg[i].L == 1`, its corresponding `sspmpswitch[i]` bit becomes read-only.

All accesses to the `sspmpswitch` CSR must adhere to the protocols defined in Section 3.7. In an implementation featuring 64 PMP entries with 48 delegated to the supervisor level, the bits `sspmpswitch[0..47]` control their respective SPMP entries `[0..47]`. Any attempt to write to the upper bits of the register, `sspmpswitch[48..63]` are ignored.

If the `Sspmpsw` extension is present and an entry's addressing mode is set to TOR via `spmpcfg[i].A`, that entry matches any address y that satisfies the following conditions:

1. `spmpaddr[i-1]` $\leq y <$ `spmpaddr[i]`.
2. This address matching is independent of the configuration or activation state of the preceding entry, i.e., `spmpcfg[i-1]` and `sspmpswitch[i-1]`.

The `sspmpswitch` register offers significant benefits for context switch optimization in various scenarios, including the following:

1. *When a hart possesses enough SPMP entries for all its concurrent tasks, memory regions for each task can be statically assigned to a dedicated subset of these entries. Under this configuration, an SPMP context switch is streamlined into a single write operation to `sspmpswitch` (or two writes on RV32 systems: `sspmpswitch` and `sspmpswitchh`). This operation simultaneously deactivates the SPMP entries of the outgoing task while activating those of the incoming task.*
2. *A subset of SPMP entries may be reserved for tasks with strict timing or latency requirements, such as interrupt service routines. This approach guarantees minimal overhead when activating these critical contexts, thereby eliminating the need to dynamically reconfigure SPMP entries during the switch.*

Chapter 5. Recommended Programming Guidelines

Two primary models guide the configuration of SPMP for isolating user-mode tasks from each other and from the S-mode operating system (OS). The selection of a model hinges on whether the number of available SPMP entries is sufficient to simultaneously contain all memory regions required by both user tasks and the OS.

- **Static Configuration**: This model involves programming all SPMP entries once at system initialization. It is predicated on the availability of enough entries to statically map all memory regions for both user tasks and the OS. This approach is only viable if the `Sspmpsw` extension is implemented.
- **Dynamic Configuration**: This model requires reprogramming SPMP entries during each context switch. It is employed when the SPMP entry count is insufficient to concurrently map all task memory regions, necessitating dynamic updates to maintain memory isolation.

5.1. Static Configuration

The static configuration model is applicable when the number of SPMP entries is sufficient to map all memory regions for both user-mode tasks and the OS. Under this model, SPMP entries are configured a single time at system initialization and are not modified at runtime. Consequently, context switches between user-mode tasks only require updating the `sspmpswitch` register(s).

During the boot process, machine-mode software is responsible for allocating SPMP entries and configuring them with the address ranges and permissions required by supervisor-mode software.

The OS then proceeds to populate the `spmpaddr[i]` and `spmpcfg[i]` CSRs with the specific address ranges and permissions for each user-mode task.

To protect its own memory, the OS also configures a set of SPMP entries for its address space, ensuring the `spmpcfg[i].U` (User) bit is cleared for these entries. Following initialization, the OS must activate its own entries by setting the corresponding bits in the `sspmpswitch` register.

Before dispatching the first user task, the OS activates the SPMP entries assigned to that task by setting the appropriate bits in `sspmpswitch`. During a context switch, the OS atomically deactivates the outgoing task's entries and activates the incoming task's entries using the `CSRRC` and `CSRRS` instructions, respectively. On RV32 systems, `sspmpswitch` and `sspmpswitchh` must be updated as an uninterruptible sequence to guarantee complete protection.

5.2. Dynamic Reconfiguration

The dynamic model is necessary when the number of available SPMP entries is insufficient to concurrently map all memory regions for the supervisor and all user tasks. Consequently, the OS must dynamically reconfigure the SPMP entries assigned to user tasks at every context switch. It is important to note that the number of SPMP entries on a hart must still be adequate to hold all supervisor entries plus the entries for one user-mode task at any given time.

The following sequence details the recommended procedure for dynamic reconfiguration:

1. **Disable Outgoing Task Entries**. Disable the SPMP entries for the outgoing task using a bitmask of its active entries, which is typically maintained in the task's control block. If the `Sspmpsw` extension is available, the OS clears the relevant bits in `sspmpswitch` via a `CSRRC` instruction.

Otherwise, the OS clears the `spmpcfg[i].A` field for each of the task's entries.

2. **Update SPMP Address Registers**. Program the `spmpaddr[i]` CSRs with the memory region boundaries of the incoming task.
3. **Update SPMP Configuration Registers**. For each relevant `spmpcfg[i]` field:
   - First, clear the existing configuration bits with a `CSRRC` instruction.
   - Then, set the new configuration bits with a `CSRRS` instruction.
4. **Enable Incoming Task Entries**. Activate the SPMP entries for the incoming task using its corresponding bitmask. If the `Sspmpsw` extension is implemented, the OS sets the bits in `sspmpswitch` with a `CSRRS` instruction. Otherwise, the OS sets the `spmpcfg[i].A` field for each of the new task's entries.

SPMP entries configured to protect the supervisor, which are identified by `spmpcfg[i].U == 0`*, should be treated as resident. It is highly recommended that these entries not be reprogrammed during the context switch procedure. Keeping supervisor entries persistent minimizes reconfiguration overhead and guarantees the consistent enforcement of supervisor memory protection.*

5.3. Entry Configuration Recommendations

Programming SPMP entries involves a trade-off between the Naturally Aligned Power-of-Two (NAPOT) and Top-of-Range (TOR) address-matching modes (see Section 3.5).

While NAPOT provides a compact, single-entry encoding for power-of-two-aligned regions, it can cause internal fragmentation if the allocated region is larger than required, leading to memory inefficiency. Conversely, TOR mode typically requires two entries to define an arbitrary region (base and top), which can consume the available entries more rapidly. However, TOR may prove more entry-efficient for certain power-of-two aligned regions.

This trade-off is particularly pronounced in Microcontroller Unit (MCU) systems, where memories often have a sparse, fixed mapping. In such scenarios, a rigid adherence to either NAPOT or a naive pairing of TOR entries can be inefficient or create unintended protection gaps. Furthermore, using consecutive or overlapping TOR entries to define multiple regions with distinct permissions can introduce subtle and hazardous dependencies. For example, sharing a spmpaddr register as a boundary between a supervisor and a user region means that shrinking the supervisor region could unintentionally enlarge the user region. Likewise, modifying a shared boundary during a context switch might inadvertently expose protected memory.

To mitigate these risks, the following disciplined configuration model is recommended:

- Exclusively use the TOR mode, and treat each even/odd indexed pair of `spmpaddr[i]` registers as a single base/top definition for a memory region.
- Establish a convention where SPMP entries are managed in pairs, such as (0, 1), (2, 3), …, (62, 63). Region activation should only be controlled via the odd-indexed entries (e.g., `sspmpswitch[1]`, `sspmpswitch[3]`), as each odd entry completes a region's definition.
- Allocate SPMP entry pairs in descending order of their indices, from highest to lowest. This strategy, which corresponds to an ascending order of priority, permits the OS to create temporary, higher-priority subregions using lower-indexed entries without disturbing existing configurations.

Adhering to this structured approach with TOR-mode entries fosters clearer isolation boundaries, minimizes the risk of configuration errors, and enhances the runtime flexibility of the memory protection scheme.

5.4. Reconfiguration Non-preemption and Synchronization

To maintain the integrity of the SPMP configuration, the entire reconfiguration sequence during a context switch must execute atomically as a non-preemptible critical section. This is necessary because the process involves modifications to multiple CSRs, and any interruption could leave the system in an inconsistent or insecure state.

For the **dynamic reconfiguration model**, this critical section must encompass all updates to `spmpaddr[i]`, `spmpcfg[i]`, and `sspmpswitch`. For the **static configuration model**, atomicity is a concern only on RV32 systems with over 32 SPMP entries, which require a coordinated update of both `sspmpswitch` and `sspmpswitchh`.

To block asynchronous interrupts during this process, the supervisor must temporarily clear the `SIE` (Supervisor Interrupt Enable) bit in the `sstatus` CSR.

By enforcing non-preemption and correct synchronization, the software guarantees that SPMP enforcement remains deterministic, secure, and verifiable across all context switches.