



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

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

Version 0.9.11, 6/2025: This document is in development. Assume everything can change. See <http://riscv.org/spec-state> for details.

Table of Contents

Preamble	1
Copyright and license information	2
Contributors	3
1. Introduction	4
2. S-mode Physical Memory Protection (SPMP)	5
2.1. Requirements	5
2.2. S-mode Physical Memory Protection CSRs	6
2.3. Encoding of Permissions	8
2.4. Address Matching	9
2.5. Matching Logic	9
2.6. SPMP and Paging	10
2.7. Exceptions	10
2.8. Context Switching Optimization	11
2.9. Access Method of SPMP CSRs	12
3. Machine-level Modification	13
3.1. Resource Sharing between PMP and SPMP	13
3.2. Access Method of PMP_Resource	14
4. Summary of Hardware Changes	15
5. Recommended Programming Guidelines	16
5.1. Static Configuration	16
5.2. Dynamic Reconfiguration	16
5.3. Entry Configuration Recommendations	17
5.4. Re-configuration Non-preemption and Synchronization	18
6. Interaction with other proposals	19

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

This specification is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). The full license text is available at creativecommons.org/licenses/by/4.0/.

Copyright 2023 by RISC-V International.

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 Offenberger
- Allen Baum
- Bill Huffman
- Xu Lu
- Wenhao Li
- Yubin Xia
- Joe Xie
- Paul Ku
- Jonathan Behrens
- Robin Zheng
- Zeyu Mi
- Fabrice Marinet
- Nouredine Ait Said

Chapter 1. Introduction

This document describes RISC-V S-mode Physical Memory Protection (SPMP) proposal to provide isolation when MMU is unavailable or disabled. RISC-V based processors recently stimulated great interest in the emerging internet of things (IoT) and automotive devices. However, page-based virtual memory (MMU) is usually undesired in order to meet resource and latency constraints. It is hard to isolate the S-mode OSes (e.g., RTOS) and user-mode applications for such devices. To support secure processing and isolate faults of U-mode software, the SPMP is desirable to enable S-mode OS to limit the physical addresses accessible by U-mode software on a hart.

Chapter 2. S-mode Physical Memory Protection (SPMP)

An optional RISC-V S-mode Physical Memory Protection (SPMP) provides per-hart supervisor-mode control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region.

If PMP/ePMP is implemented, accesses succeed only if both PMP/ePMP and SPMP permission checks pass. The implementation can perform SPMP checks in parallel with PMA and PMP. The SPMP exception reports have higher priority than PMP or PMA exceptions (i.e., if the access violates both SPMP and PMP/PMA, the SPMP exception will be reported).

SPMP checks will be applied to all accesses whose effective privilege mode is S or U, including instruction fetches and data accesses in S and U mode, and data accesses in M-mode when the MPRV bit in mstatus is set and the MPP field in mstatus contains S or U.

SPMP registers can always be modified by M-mode and S-mode software.

SPMP can grant permissions to U-mode, which has none by default. SPMP can also revoke permissions from S-mode.

2.1. Requirements

1. S mode should be implemented
2. The Sscsrind extension should be implemented to support indirect CSR access.
3. The sstatus.SUM (permit Supervisor User Memory access) bit must be **writeable**.

The Privileged Architecture specification states the following



SUM has no effect when page-based virtual memory is not in effect, nor when executing in U-mode. SUM is read-only 0 if satp.MODE is read-only 0.

— Supervisor-Level ISA, Version 1.13 >> "Memory Privilege in `sstatus` Register"

In SPMP, this bit modifies the privilege with which S-mode loads and stores access to physical memory, hence the need to make it writeable.

4. The sstatus.MXR (Make eXecutable Readable) bit must be **writeable**.

The Privileged Architecture specification states that



MXR has no effect when page-based virtual memory is not in effect.

— Machine-Level ISA, Version 1.13 >> "Memory Privilege in `mstatus` Register"

In SPMP, the MXR bit modifies the privilege with which loads access physical memory. Its semantics are consistent with those of the Machine-Level ISA.

In SPMP, this bit is made writeable to support M-mode emulation handlers where instructions are read with MXR=1 and MPRV=1.

2.2. S-mode Physical Memory Protection CSRs

Like PMP, SPMP entries are described by an 8-bit configuration register and one XLEN-bit address register. Some SPMP settings additionally use the address register associated with the preceding SPMP entry. Up to 64 SPMP entries are supported.

The SPMP configuration registers are packed into CSRs the same way as PMP. For RV32, 16 CSRs, `spmpcfg0`–`spmpcfg15`, hold the configurations `spmp0cfg`–`spmp63cfg` for the 64 SPMP entries. For RV64, even numbered CSRs (i.e., `spmpcfg0`, `spmpcfg2`, ..., `spmpcfg14`) hold the configurations for the 64 SPMP entries; odd numbered CSRs (e.g., `spmpcfg1`) are illegal. Figure 1 and Figure 2 demonstrate the first 16 entries of SPMP. The layout of the rest of the entries is identical.



An SPMP entry denotes a pair of `spmpcfg[i]` / `spmpaddr[i]` registers.

An SPMP rule denotes the contents of a `spmpcfg` register and its associated `spmpaddr` register(s), that encode a valid protected physical memory region, where `spmpcfg[i].A != OFF`, and if `spmpcfg[i].A == TOR`, `spmpaddr[i-1] < spmpaddr[i]`.

*The implementation must decode all SPMP CSRs, and it can modify the number of **writable SPMP entries** while the remaining SPMP CSRs are read-only zero.*

The lowest-numbered SPMP entries must be implemented first.

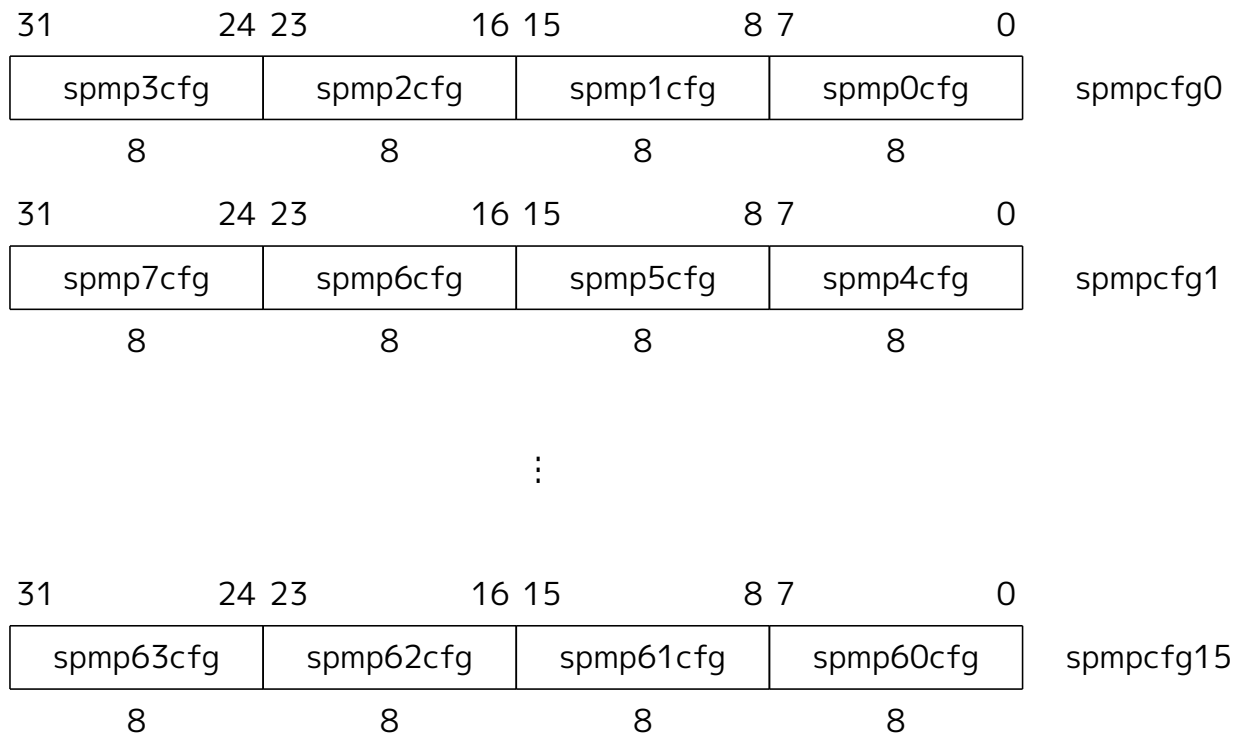


Figure 1. RV32 SPMP configuration CSR layout.

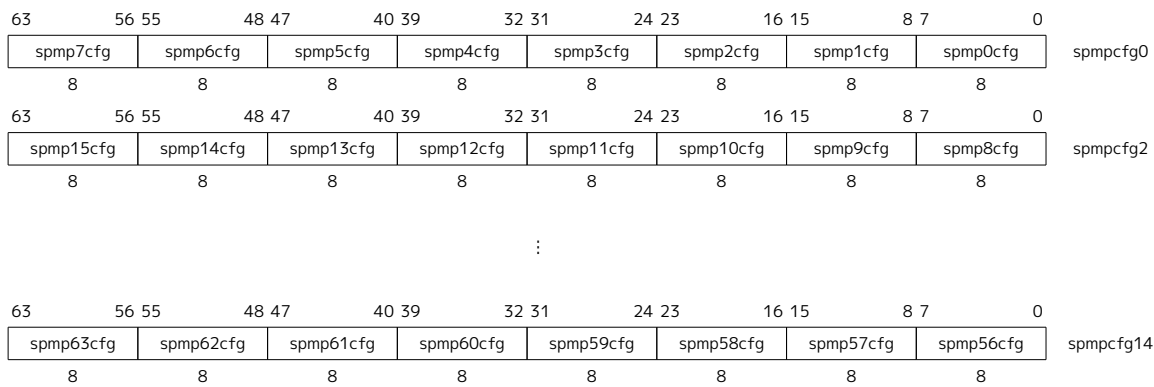


Figure 2. RV64 SPMP configuration CSR layout.

The SPMP address registers are CSRs named `smpaddr0`–`smpaddr63`. Each SPMP address register encodes bits 33–2 of 34-bit physical address for RV32, as shown in Figure 3. For RV64, each SPMP address encodes bits 55–2 of a 56-bit physical address, as shown in Figure 4. Fewer address bits may be implemented for specific reasons, e.g., systems with smaller physical address space. The number of address bits should be the same for all **writable SPMP entries**. Implemented address bits must extend to the LSB format, except as otherwise permitted by granularity rules. See the Privileged Architecture specification, Section 3.7: Physical Memory Protection, Address Matching.

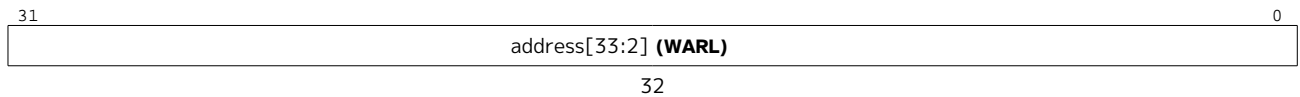


Figure 3. SPMP address register format, RV32.

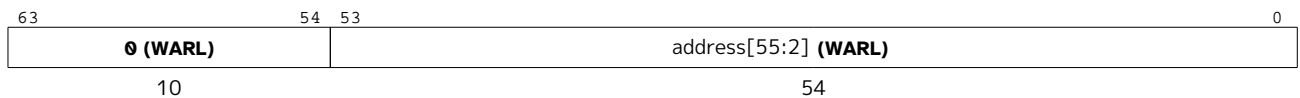


Figure 4. SPMP address register format, RV64.

The layout of SPMP configuration registers is shown in Figure 5. The register is WARL. The rules and encodings for permission are explained in Section 2.3.



1. The L bit marks an entry as locked, i.e., writes to the configuration register and associated address registers are ignored.

The L bit is only accessible to M-mode (Detailed in Section 2.9).

The L bit is not a sticky bit, a locked SPMP entry can only be reset by M-mode.

Setting the L bit locks the SPMP entry even when the A field is set to OFF.



For a locked SPMP entry i , writes to `smp[i]cfg` and `smpaddr[i]` will succeed if the effective privilege mode is M.

For a locked SPMP entry i , writes to `smp[i]cfg` and `smpaddr[i]` are ignored if the effective privilege mode is S. Additionally, if `smp[i]cfg.A` of the locked entry is set to TOR, S-mode writes to `smpaddr[i-1]` are ignored.



The L bit can be used by M-mode to contain software running in S-mode by setting and locking highest-priority SPMP entries with `smp[i]cfg.S` set. This can be useful to prevent privilege escalation attacks that would reprogram SPMP entries used to limit S-mode accesses.

Although this could arguably be achieved by using PMP/ePMP entries, the resulting configuration would not be equivalent as they do not differentiate between S and U modes. Furthermore, in cases resource sharing is statically defined (i.e., `mpmpdeleg.pmpnum` is hardwired - see [Section 3.1](#)) there may be insufficient PMP/ePMP entries available to implement the desired isolation.

1. The S bit marks a rule as **S-mode-only** when set and **U-mode-only** when unset.
2. Bit 5 is reserved for future use.
3. The A field will be described in the following sections.
4. The R/W/X bits control read, write, and instruction execution permissions.

7	6	5	4	3	2	1	0
L (WARL)	S (WARL)	Reserved	A (WARL)	X (WARL)	W (WARL)	R (WARL)	
1	1	1	2	1	1	1	1

Figure 5. SPMP configuration register format.

2.3. Encoding of Permissions

SPMP has three kinds of rules: **S-mode-only**, **U-mode-only** and **Shared-Region** rules.

1. An **S-mode-only** rule is **enforced** on Supervisor mode and **denied** on User mode.
2. A **U-mode-only** rule is **enforced** on User modes and is either **denied** or **enforced** on Supervisor mode depending on the value of `sstatus.SUM` bit:
 - If `sstatus.SUM` is set, a U-mode-only rule is enforced on Supervisor mode, yet not be executable. This ensures the Supervisor Memory Execution Prevention (SMEP).
 - If `sstatus.SUM` is unset, a U-mode-only rule is denied on Supervisor mode. This ensures the Supervisor Memory Access Prevention (SMAP).
3. A **Shared-Region** rule is enforced on both Supervisor and User modes, with restrictions depending on the `spmpcfg.S` and `spmpcfg.X` bits:
 - If `spmpcfg.S` is not set, the region can be used for sharing data between S-mode and U-mode, yet not be executable. S-mode has RW permission to that region. U-mode has RW permission if `spmpcfg.X` is set, and it is restricted to read-only if `spmpcfg.X` is cleared.
 - If `spmpcfg.S` is set, the region can be used for sharing code between S-mode and U-mode, yet not be writeable. S-mode has RX permission to that region. U-mode has RX permission if `spmpcfg.X` is set, and it is restricted to execute-only if `spmpcfg.X` is cleared.
4. The encoding `spmpcfg.SRWX=1000` is reserved for future standard use.

The encoding and results are shown in [Figure 6](#):

	smpcfg.S=0			smpcfg.S=1		
smpcfg	S mode Access	S mode Access	U mode Access	S mode Access	S mode Access	U mode Access
RWX	sstatus.SUM =0	sstatus.SUM =1	sstatus.SUM =x	sstatus.SUM =0	sstatus.SUM =1	sstatus.SUM =x
R --	Deny	EnforceNoX	Enforce	Enforce	Enforce	Deny
R - X	Deny	EnforceNoX	Enforce	Enforce	Enforce	Deny
-- X	Deny	EnforceNoX	Enforce	Enforce	Enforce	Deny
- - -	Deny	EnforceNoX	Enforce	RSVD		
RW -	Deny	EnforceNoX	Enforce	Enforce	Enforce	Deny
RWX	Deny	EnforceNoX	Enforce	Enforce	Enforce	Deny
- WX	SHR RW			SHR RX		
- W -	SHR RW		SHR RO	SHR RX		SHR X

Figure 6. SPMP Encoding Table

Deny: Access fails.

Enforce: The R/W/X permissions are enforced on accesses.

EnforceNoX: The R/W permissions are enforced on accesses, while the X bit is forced to be zero.

SHR: It is shared between S/U modes with X, RX, RW, or ReadOnly privileges.

RSVD: It is reserved for future use.

SUM bit: The SPMP uses the sstatus.SUM (permit Supervisor User Memory access) bit to modify the privilege with which S-mode loads and stores access to physical memory. The semantics of sstatus.SUM in SPMP are consistent with those of the Machine-Level ISA (Please refer to the "Memory Privilege in mstatus Register" subsection in the riscv-privileged spec for detailed information).

2.4. Address Matching


The A field in an SPMP entry's configuration register encodes the address-matching mode of the associated SPMP address register. It is the same as PMP/ePMP.

Please refer to the "Address Matching" subsection of PMP in the riscv-privileged spec for detailed information.



Software may determine the SPMP granularity by writing zero to smp0cfg, then writing all ones to smpaddr0, then reading back smpaddr0. If G is the index of the least-significant bit set, the SPMP granularity is 2^{G+2}

2.5. Matching Logic

- SPMP entries are statically prioritized. 
- The lowest-numbered SPMP entry that matches any byte of access (indicated by an address and the accessed length) determines whether that access is allowed or denied
- The SPMP entry must match **all** bytes of access, or the access fails
- This matching is done irrespective of the S, R, W, and X bits

On some implementations, misaligned loads, stores, and instruction fetches may also be decomposed into multiple accesses, some of which may succeed before an exception occurs. In particular, a portion of a misaligned store that passes the SPMP check may become visible, even if another portion fails the SPMP check. The same behavior may manifest for stores wider than XLEN bits (e.g., the FSD instruction in RV32D), even when the store address is naturally aligned.

1. If the effective privilege mode of the access is M, the access is allowed;
2. If the effective privilege mode of the access is S/U and no SPMP entry matches, but at least one SPMP entry is delegated, the access is denied; (Description of the delegated SPMP entry is in [Section 3.1](#))
3. 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 SRWX encoding corresponding to the access type.



The SPMP rules are checked for all implicit and explicit accesses in all S-mode and lesser-privileged modes. Implicit accesses in S-mode are treated as S-mode accesses.

The execution environment should configure SPMP entry(s) to grant the most permissive access to S-mode. Then S-mode code can set up SPMP as desired.

2.6. SPMP and Paging

The table below shows which mechanism to use. (Assume both paged virtual memory and SPMP are implemented.)

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

SPMP and paged virtual memory cannot be active simultaneously for two reasons:

1. An additional permission check layer would be introduced for each memory access.
2. Sufficient protection is provided by paged virtual memory.

That means SPMP is enabled when `satp.mode==Bare` and SPMP is implemented.



Please refer to Table "Encoding of satp MODE field" in the riscv-privileged spec for detailed information on the satp.MODE field.

2.7. Exceptions

When an access fails, SPMP generates an exception based on the access type (i.e., load accesses, store/AMO accesses, and instruction fetches). Each exception has a different code.

The SPMP reuses page fault exception codes for SPMP faults since page faults are typically delegated to S-mode. S-mode software (i.e., OS) can distinguish between SPMP and page faults by checking `satp.mode`, since SPMP and paged virtual memory cannot be active simultaneously (as described in [Section 2.6](#)).

Note that a single instruction may generate multiple accesses, which may not be mutually atomic.

Table of exception codes:



Interrupt	Exception Code	Description
0	12	Instruction page fault
0	13	Load page fault
0	15	Store/AMO page fault



Please refer to Table "Supervisor cause register (scause) values after trap" in the riscv-privileged spec for detailed information on exception codes.

Delegation: Unlike PMP, which uses access faults for violations, SPMP uses page faults for violations. The benefit of using page faults is that the violations caused by SPMP can be delegated to S-mode, while the access violations caused by PMP can still be handled by machine mode.

2.8. Context Switching Optimization

Context switching with SPMP requires storing 64 address and 8 configuration registers (RV64), creating significant overhead. To optimize this:

- In RV32: two XLEN-bit read/write CSRs called `spmpswitch` and `spmpswitchh` are added, as depicted in [Figure 7](#).
- In RV64: one XLEN-bit read/write CSR called `spmpswitch` is added, as depicted in [Figure 8](#).

Each bit controls the activation of its corresponding SPMP entry. An entry is active only when both its `spmpswitch[i]` bit and `spmp[i]cfg.A` field are set, i.e., `spmpswitch[i] & spmp[i]cfg.A != 0`.

The `spmpswitch` registers must be cleared on reset.

Please refer to [Chapter 5](#) for how software can use the optimization to reduce context switch overhead.



Figure 7. SPMP domain switch registers (`spmpswitch` and `spmpswitchh`), RV32.



Figure 8. SPMP domain switch register (`spmpswitch`), RV64.



The context switching optimization is **optional**.

When `spmpswitch` is implemented and `spmpcfg[i].A == TOR`, an entry matches any address y where:

1. $\text{spmpaddr}[i-1] \leq y < \text{spmpaddr}[i]$
2. This matching occurs regardless of `spmpcfg[i-1]` and `spmpswitch[i-1]` values

In case where `spmpswitch[i] == 0` and `spmp[i]cfg.L == 1`, entry i remains enabled. An implementation can hardwire the `L` bit to 0 if the lock functionality is not required.



Utilizing `spmpswitch` for optimizing context switches can be beneficial in several scenarios, including (but not limited to):

1. When the number of available SPMP entries is sufficient to accommodate all tasks executing on a given hart, each task's memory regions can be permanently mapped to a fixed subset of SPMP entries. In this model, switching SPMP contexts reduces to a single write to `spmpswitch` (or two writes in RV32 systems: `spmpswitch` and `spmpswitchh`) to deactivate the outgoing task and enable the entries associated with the incoming task.
2. A subset of SPMP entries may be reserved for timing-critical or latency-sensitive tasks, such as interrupt handlers. This ensures minimal overhead when switching into these contexts, avoiding the need for dynamic reconfiguration of SPMP entries.

2.9. Access Method of SPMP CSRs



Indirect CSR access: The SPMP CSRs are accessed indirectly. Each combination of `siselect` and `sireg` represents an access to the corresponding SPMP CSR.



The indirect CSR access avoids the potential cost in pipeline flushes. However, there is no ordering guarantee between writes to different SPMP CSRs, except when explicitly executing an `SFENCE.VMA` instruction with `rs1=x0` and `rs2=x0`.

S-mode can only access bit 0-6 of each field of the configuration register. This means that `spmp[i]cfg.L` can only be accessed by M-mode.



siselect number	indirect CSR access of sireg
siselect#1	sireg → <code>spmpaddr[0]</code> , <code>sireg2</code> → <code>spmp[0]cfg{0..6}</code>
siselect#2	sireg → <code>spmpaddr[1]</code> , <code>sireg2</code> → <code>spmp[1]cfg{0..6}</code>
...	...
siselect#64	sireg → <code>spmpaddr[63]</code> , <code>sireg2</code> → <code>spmp[63]cfg{0..6}</code>



The rationale for SPMP only assign one rule per `siselect` value is due to performance consideration. If multiple SPMP rules are assigned to each `siselect`, a jump table or additional calculations would be needed to determine which `sireg` to assess.

Please refer to the `Sscsrind` extension specification for details on indirect CSR accesses: github.com/riscv/riscv-indirect-csr-access

Chapter 3. Machine-level Modification

Given that PMP and SPMP have similar layout of address/config registers and the same address matching logic. Reusing registers and comparators between PMP and SPMP may be beneficial (in some cases) to save hardware resources. This chapter introduces the resource sharing mechanism that can support dynamic reallocation of hardware resource between PMP and SPMP.

3.1. Resource Sharing between PMP and SPMP

Implementations should consider PMP/SPMP entries as a resource pool (called PMP_Resource). Specifically, each PMP_Resource consists of an address CSR, a configuration CSR, and associated micro-architecture state. A new M-mode CSR called `mpmpdeleg` is introduced to control the sharing of PMP_Resource between PMP and SPMP.

In the following description, we will refer to the PMP/SPMP from the hardware perspective as PMP_Resource, and the PMP/SPMP from the software perspective as entry.

The 16-bit CSR shown in [Figure 9](#) has one `pmpnum` field:

1. `pmpnum` is 7-bit, allowing a value of 0–64 to specify the number of PMP entries.
2. Any PMP_Resource greater than or equal to the `pmpnum` is delegated to S-mode (SPMP). The lower numbered PMP_Resource are left for M-mode (PMP).
3. M-mode could set `pmpnum` ≥ 64 (the number of implemented PMP_Resource), to reserve all resources for PMP.
4. M-mode could set `pmpnum` = 0 to delegate all PMP_Resource to SPMP.
5. The reset value of `pmpnum` is 0b100_0000.
6. The `pmpnum` is locked when `mseccfg.MML` is set (Please refer to the `Smpmp` extension specification for details on `mseccfg.MML`).

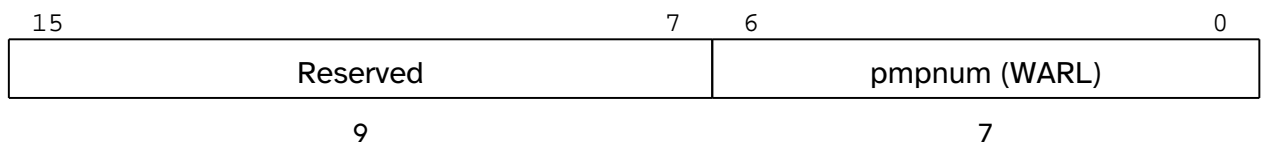


Figure 9. `mpmpdeleg` CSR format.

Constraints:

1. With RV32, the values of `pmpnum`, can only be a multiple of 4; with RV64, it can only be a multiple of 8. This design avoids sharing the same configuration CSR between S-mode and M-mode.
2. The `pmpnum` is a WARL field. Illegal writes (e.g., values that are not multiples of 4 (RV32) or 8 (RV64)) should be ignored.
3. If the SPMP entry with lowest CSR number is configured with TOR address-matching mode, zero is used for the lower bound.



The resource sharing is **mandatory**.

The `mpmpdeleg` CSR is WARL, and allows an implementation to hardwire the PMP/SPMP split if desired.

To reduce context-switch overhead, S-mode software should evaluate the number of SPMP rules each S-mode context needs and set `mpmpdeleg.pmpnum` accordingly.

Addressing:

Both PMP and SPMP entries will be supported contiguously. The PMP entries begin with the lowest CSR number, while the SPMP entries begin with `mpmpdeleg.pmpnum`. For instance, given an implementation with a total of 64 PMP Resource entries, if `mpmpdeleg.pmpnum` is set to 16 during runtime, `PMPResource[0]` to `PMPResource[15]` would map to `PMP[0]` to `PMP[15]`. The remaining entries, `PMPResource[16]` to `PMPResource[63]`, would be mapped as `SPMP[0]` to `SPMP[47]`.

From a software perspective, the SPMP entries start from `SPMP[0]`. The available number of SPMP entries can be discovered by writing to and reading from the SPMP CSRs. Illegal writes to SPMP CSRs will be ignored.

Re-configuration:

M-mode software can re-configure the entries for PMP and SPMP by modifying the `mpmpdeleg` CSR. A re-configuration involving locked PMP entry will leave `mpmpdeleg` unchanged.

3.2. Access Method of PMP_Resource

Indirect CSR access: PMP_Resource CSRs can be accessed indirectly from M-mode. Each combination of `miselect` and `mireg` represents an access to the corresponding PMP_Resource CSR.

There is no ordering guarantee between writes to different PMP_Resource CSRs via indirect access, except when explicitly executing an `SFENCE.VMA` instruction with `rs1=x0` and `rs2=x0`.

M-mode can access every bits of configuration registers.

miselect number	indirect CSR access of mireg
miselect#1	mireg → pmp/spmpaddr[0], mireg2 → pmp/spmp[0]cfg{0..7}
miselect#2	mireg → pmp/spmpaddr[1], mireg2 → pmp/spmp[1]cfg{0..7}
...	...
miselect#64	mireg → pmp/spmpaddr[63], mireg2 → pmp/spmp[63]cfg{0..7}

Chapter 4. Summary of Hardware Changes

Supervisor-level Changes:

Item	Changes
CSR for domain switch (optional)	2 new CSRs for RV32 and 1 for RV64
Indirect access to delegated PMP_Resource	64 new siselect values

Machine-level Changes:

Item	Changes
CSR for resource sharing	1 new CSR
Indirect access to PMP_Resource	64 new siselect values

Chapter 5. Recommended Programming Guidelines

When configuring SPMP to isolate user-mode tasks from each other and from the operating system (OS) executing in supervisor mode, two primary usage models arise, depending on whether the available SPMP entries can simultaneously accommodate all required memory ranges for both the user tasks and the OS:

- **Static Configuration:** all SPMP entries are programmed during system initialization. This model assumes that the number of available entries is sufficient to cover the complete set of memory regions assigned to user tasks and the OS without further modification.
- **Dynamic Configuration:** SPMP entries are reprogrammed on each context switch. This model is employed when the number of available SPMP entries is insufficient to simultaneously represent all relevant memory regions, requiring dynamic updates to enforce memory isolation between tasks.

5.1. Static Configuration

In the static configuration model, the number of available SPMP entries is sufficient to accommodate all required memory ranges for user-mode tasks and the OS. In this case, SPMP entries are programmed once during system initialization and remain unchanged at runtime. Only the `spmpswitch` register(s) need to be updated during context switches between user-mode tasks.

The M-mode software is responsible for allocating SPMP entries and configuring them with the appropriate address ranges and permissions for S-mode software during boot.

The OS begins by allocating SPMP entries and populating the `spmpaddr[i]` and `spmpcfg[i]` CSRs with the appropriate address ranges and permissions for each user-mode task. If the OS itself is to be protected using SPMP, additional entries must be allocated and configured with the OS memory ranges, with the S bit (Supervisor bit) set in `spmpcfg[i]`.

After initializing the entries, the OS must also set the corresponding bits in the `spmpswitch` register to activate these entries.

Prior to launching the first user task, the OS sets the bits in `spmpswitch` corresponding to the SPMP entries assigned to that task. During a context switch, the OS clears the current task's entry bits and sets those of the newly scheduled task using the `csrc` and `csrs` instructions, respectively. On RV32 systems, both `spmpswitch` and `spmpswitchh` must be managed as a continuous pair.

5.2. Dynamic Reconfiguration

In this configuration model, the available SPMP entries are insufficient to simultaneously represent the memory ranges required for all user-mode tasks and the supervisor. As a result, the OS must dynamically reconfigure SPMP entries for user tasks on every context switch. Notably, for any given hart, the number of SPMP entries must still be sufficient to hold both the supervisor entries and the entries for the currently executing user-mode task.

Where sufficient SPMP entries exist to cover all tasks and the OS, an implementation may simply update `spmpswitch` on context switches. Otherwise, the following sequence is recommended for dynamic reconfiguration:

1. **Disable Entries for the Outgoing Task.** Use the `csrc` instruction to clear the `spmpswitch` bits corresponding to the SPMP entries of the outgoing task. A bitmask representing active entries

(typically stored in the task's control block) should be passed as an argument.

2. **Update SPMP Address Registers.** Write the `spmpaddr[i]` CSRs with the address ranges corresponding to the memory regions of the incoming task.
3. **Update SPMP Configuration Registers.** For each corresponding `spmpcfg[i]` field:
 - Clear the existing configuration bits using the `csrc` instruction with an appropriate mask;
 - Set the desired configuration using the `csrs` instruction.
4. **Enable Entries for the Incoming Task.** Use the `csrs` instruction to write to `spmpswitch`, passing a bitmask that enables the SPMP entries allocated to the incoming task.



It is recommended that SPMP entries configured to protect the supervisor (i.e., entries with `spmpXcfg[i].S = 1`) remain resident and are not reprogrammed during the context-switch process. Maintaining these entries as persistent minimizes reconfiguration overhead and ensures consistent enforcement of memory protection for the supervisor across task switches.

5.3. Entry Configuration Recommendations

When programming SPMP entries, a trade-off exists between using exclusively Naturally Aligned Power-Of-Two (NAPOT) or Top-Of-Range (TOR) address-matching modes.

While NAPOT allows compact encoding for power-of-two-aligned regions using a single entry, it may lead to internal fragmentation if the region size exceeds the actual requirement, resulting in memory waste. On the other hand, TOR mode (particularly in its generic form) requires two SPMP entries per protected region (base and top), which may exhaust available entries more quickly. However, for regions that are naturally power-of-two aligned, TOR may still be encoded with fewer entries.

This trade-off becomes especially relevant in MCU environments, where local memories are often sparsely mapped to fixed address ranges associated with specific core functions. In such cases, exclusive use of NAPOT or naïvely pairing TOR entries may be inefficient or lead to undesirable gaps in protection. Additionally, while it is possible to define multiple contiguous regions with different access permissions using overlapping or consecutive TOR entries, this technique can introduce subtle dependencies. Sharing a top address between two address spaces (e.g., supervisor and user) can lead to unintended interactions: reducing the supervisor region may inadvertently expand the adjacent user region. Similarly, replacing a shared top address during context switches could expose previously protected memory.

Given these risks, the following configuration model is recommended:

- Use TOR mode exclusively, treating each pair of `spmpaddr[i]` registers (even/odd indexed) as a base/top pair defining a single memory region;
- Conceptually, view SPMP entries as organized into pairs: (0/1), (2/3), ..., (62/63). Only the odd-indexed entries are enabled via the corresponding bits in `spmpswitch`, as each odd entry finalizes the definition of a region;
- Allocate and populate SPMP entries in descending index order (i.e., from lower priority to higher priority), starting from the highest index downward. This allocation strategy allows the OS to define temporary subregions by configuring unused lower-index entries without needing to reconfigure existing higher-index (priority) ones.

This disciplined use of TOR-mode SPMP entries ensures clearer isolation boundaries, reduces the

likelihood of configuration errors, and improves runtime flexibility for memory protection schemes.

5.4. Re-configuration Non-preemption and Synchronization

To preserve the integrity of SPMP state, the reconfiguration process during a context switch must be executed as a non-preemptible critical section. This requirement stems from the need to update multiple control and configuration CSRs, and any interruption or concurrent modification during this process can result in transient inconsistencies or unintended access permissions.

In the **dynamic reconfiguration model**, the critical section must be enforced across updates to the following CSRs: `spmpaddr[i]`, `spmpcfg[i]` , and `spmpswitch` . In the **static configuration model**, this concern is relevant only for RV32 systems with more than 32 SPMP entries, where both `spmpswitch` and `spmpswitchh` must be updated in coordination.

To prevent asynchronous supervisor-level interrupts during reconfiguration, the supervisor must clear the SIE (Supervisor Interrupt Enable) bit in the `sstatus` CSR. Furthermore, synchronisation mechanisms (e.g., mutexes or spinlocks) must be employed to serialise access to SPMP CSRs in multi-threaded or multi-core systems, ensuring that concurrent modifications do not result in conflicting or corrupted configurations.

By enforcing non-preemption and proper synchronisation, software ensures that SPMP protections remain deterministic, secure, and verifiable across context switches.

Chapter 6. Interaction with other proposals

This section discusses how SPMP interacts with other proposals.

J-extension pointer masking proposal: When both PM and SPMP are used, SPMP checking should be performed using the actual addresses generated by PM (pointer masking).

Hypervisor extension: Virtualization support for the SPMP will be an independent proposal.

Smstateen extension: SPMP adds readable and writable states in S-mode, which can be abused as a covert channel if the OS/hypervisor is not aware of SPMP (thus the states won't be context-switched). It is desired that SPMP occupies a bit in mstateen register of Smstateen extension, which can control supervisor access to SPMP states.