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

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

Version 1.0.0-rc3, 8/2025: 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. "Smpmpdeleg" Extension for Sharing Hardware Resources between PMP and SPMP **5**
2.1. Resource Sharing between PMP and SPMP 5
3. "Sspmp" Extension for S-level Physical Memory Protection (SPMP) **6**
3.1. Extension Dependencies 6
3.2. S-level Physical Memory Protection CSRs 6
3.3. Encoding of Permissions 7
3.4. Address Matching 9
3.5. Matching Logic 9
3.6. The Access Methods for SPMP CSRs in M-mode 10
3.7. The Access Method for SPMP CSRs in S-mode 10
3.8. SPMP and Paging 11
3.9. Exceptions 11
4. "Sspmpsw" Extension for Optimizing Context Switching of SPMP Entries **13**
5. Recommended Programming Guidelines **14**
5.1. Static Configuration 14
5.2. Dynamic Reconfiguration 14
5.3. Entry Configuration Recomendations 15
5.4. Re-configuration Non-preemption and Synchronization 16

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 describes the RISC-V S-level Physical Memory Protection (SPMP) proposal to provide isolation when virtual memory (via an MMU) is unavailable or disabled. RISC-V based processors recently stimulated great interest in the emerging markets of internet of things (IoT) and automotive devices. However, use of virtual memory is usually undesirable for these markets in order to meet resource and latency constraints. Without the use of an MMU, it is difficult on such devices to isolate the S-mode operating systems (e.g., RTOS) and user-mode applications from each other. The SPMP extension features an architecture and programming model similar to the existing PMP approach, providing mechanisms for an S-mode OS to support secure processing and fault isolation of U-mode software by limiting the physical addresses accessible to U-mode software running on a hart.

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

Given that the PMP and SPMP registers have a similar layout of address/config registers and the same address matching logic, reusing registers and comparators between PMP and SPMP is beneficial to save hardware resources. This chapter presents the `Smpmpdeleg` extension, a resource sharing mechanism enabling dynamic reallocation of hardware resources between PMP and SPMP. This extension is mandatory.

2.1. Resource Sharing between PMP and SPMP

The `Smpmpdeleg` extension enables delegation of PMP entries for S-level use. These delegated entries are referred to as S-level PMP (SPMP) entries.

A 32-bit M-mode CSR called `mpmpdeleg` is introduced to control the sharing of PMP entries between PMP and SPMP. The layout of `mpmpdeleg` is shown in Figure 1:

1. The `pmpnum` field is a WARL field specifying the index of the lowest PMP entry delegated to S-level. All PMP entries with an index greater than or equal to `pmpnum` are delegated as SPMP entries.
2. If `pmpnum` is written with a value greater than the number of implemented PMP entries, the field will read back as the number of implemented PMP entries.
3. When `pmpnum` is set to zero, all PMP entries are delegated; when it is set to the number of implemented PMP entries, no entries are delegated.
4. Unless `pmpnum` is hardwired, its reset value equals the number of implemented PMP entries.

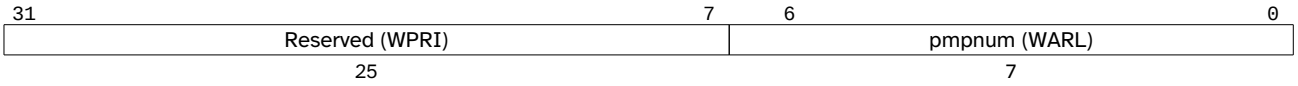

Figure 1. mpmpdeleg CSR format.

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

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 `pmpnum`. For instance, given an implementation with a total of 64 PMP entries, if `pmpnum` is set to 16 during runtime, `PMP entry[0]` to `PMP entry[15]` would map to `PMP[0]` to `PMP[15]`. The remaining entries, `PMP entry[16]` to `PMP entry[63]`, would be mapped as `SPMP[16]` to `SPMP[63]`. A read of an out-of-index PMP entry (e.g., `PMP[16]` or `SPMP[15]`) will return `0`, and a write to such a PMP entry will be ignored.

Software that uses SPMP should start with `SPMP[pmpnum]`.

Re-configuration:

1. M-mode software can re-configure the allocation of entries for PMP vs. SPMP by modifying the `mpmpdeleg` CSR.
2. The `pmpnum` must be set to a value larger than the index of any **locked** PMP entry. For example, if `PMP[7]` is locked, `pmpnum` must be no less than 8.

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

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

Memory accesses succeed only if both PMP/ePMP and SPMP permission checks pass. The implementation can perform SPMP checks in parallel with the PMA and PMP checks. The SPMP exception has 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 are applicable to all memory accesses where the effective privilege mode is less than M.

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

3.1. Extension Dependencies

1. The SPMP is dependent on `Sm1p13` and `Ss1p13`.
2. The `Smpmpdeleg` extension must be implemented to support resource sharing between PMP and SPMP.
3. The `Sscsrind` extension must be implemented to support indirect CSR access.
4. The `sstatus.SUM` (permit Supervisor User Memory access) bit must be **writable**, which is a change from the Privileged Architecture. In SPMP, this bit modifies how S-mode load and store operations access user memory, so it must be writable.
5. The `sstatus.MXR` (Make eXecutable Readable) bit must be **writable**, which is also a change from the Privileged Architecture. In SPMP, this bit is made writable to support M-mode emulation handlers where instructions are read with `MXR=1` and `MPRV=1`.

3.2. S-level Physical Memory Protection CSRs

SPMP entries are described by a 16-bit field in an XLEN-bit configuration register and an XLEN-bit address register. When the TOR mode is used, the address register associated with the preceding SPMP entry is also used (detailed in Section 3.4). Up to 64 SPMP entries are supported.

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

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

The SPMP address registers are CSRs named `spmpaddr0-spmpaddr63`, and has the same layout as the PMP architecture. For RV32, each SPMP address register encodes bits 33-2 of 34-bit physical address, as shown in Figure 2. For RV64, each SPMP address encodes bits 55-2 of a 56-bit physical address, as shown in Figure 3. 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**. Not all physical address bits may be implemented, and so the SPMP address registers are

WARL, except as otherwise permitted by granularity rules. See the "RISC-V Privileged Architecture", Section 3.7: Physical Memory Protection, Address Matching.

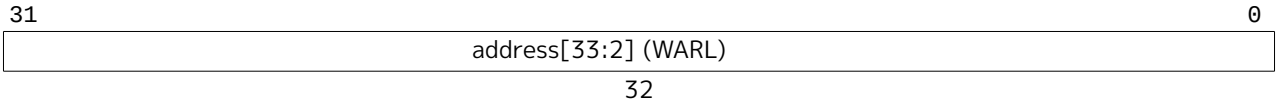

Figure 2. SPMP address register format, RV32.

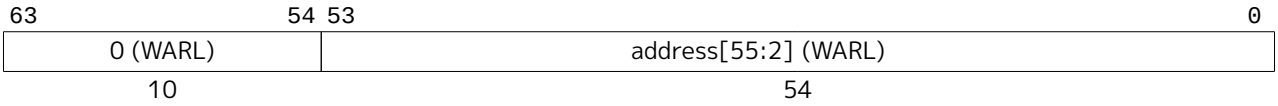

Figure 3. SPMP address register format, RV64.

Each SPMP entry features a 16-bit configuration field, denoted as `spmpcfg[i]`, whose lower 8 bits alias the 8 bits of the corresponding PMP configuration register field. The layout within each `spmpcfg[i]` is shown in Figure 4. The rules and encodings for permission are explained in Section 3.3.

1. The R/W/X bits control read, write, and instruction execution permissions.
2. The A field will be described in Section 3.4.
3. Bits 5 and 6 are used for memory types if `Smpmpmt` is implemented, else reserved (please refer to the specification of `Smpmpmt` for more details).
4. The L bit marks an entry as locked. Setting the L bit locks the SPMP entry even when the A field is set to OFF. Writes to locked `spmpcfg[i]` and `spmpaddr[i]` will succeed only if the privilege mode is M via the `miselect` CSR. Writes to locked `spmpcfg[i]` and `spmpaddr[i]` are ignored from both privilege modes (M and S) via the `siselect` CSR (see Section 3.6 and Section 3.7). Additionally, if `spmpcfg[i].A` of the locked entry is set to TOR, writes to `spmpaddr[i-1]` via `siselect` are ignored.
5. An implementation can hardwire the L bit to `0` if the lock functionality is not required.
6. For a rule that is not `Shared-Region`, the U bit marks it as **U-mode-only** when set and **S-mode-only** when cleared (details in Section 3.3).
7. The SHARED bit is used to mark **Shared-Region** rules.

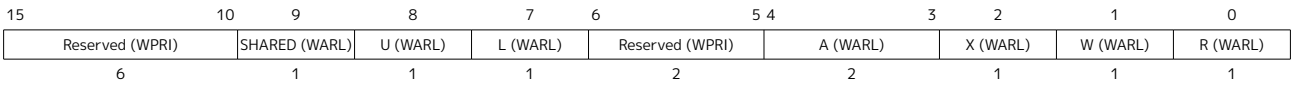

Figure 4. SPMP configuration register format.

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 `spmpcfg[i].U == 1`. 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 2.1) there may be insufficient PMP/ePMP entries available to implement the desired isolation.

3.3. 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, but denies executable permission by S-mode (denoted as **EnforceNoX** in Figure 5). This ensures the Supervisor Memory Execution Prevention (SMEP) guarantee.
   - If `sstatus.SUM` is cleared, a U-mode-only rule is denied on Supervisor mode. This ensures the Supervisor Memory Access Prevention (SMAP) guarantee.
3. `spmpcfg.SHARED == 1 and spmpcfg.U == 1`, denotes a **Shared-Region** rule. The SUM bit is ignored in this case.
4. A **Shared-Region** rule is **enforced** on both Supervisor and User modes, with the restriction that read and write permissions are mutually exclusive in User mode.
5. The R, W, and X fields form a collective WARL field.
6. The encoding `spmpcfg.XWR=010`, `spmpcfg.XWR=110`, and the combination `spmpcfg.SHARED == 1 and spmpcfg.U == 0` is reserved for future standard use.

The encoding and results are shown 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	
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 =0	sstatus.SUM =1	sstatus.SUM is ignored	
RWX=000	Enforce	Deny	EnforceNoX	Deny	Enforce	Enforce	Enforce	Enforce
RWX =100	Enforce	Deny	EnforceNoX	Deny	Enforce	Enforce	Enforce	Enforce
RWX =110	Enforce	Deny	EnforceNoX	Deny	Enforce	Enforce	Read-only	Enforce
RWX =001	Enforce	Deny	EnforceNoX	Deny	Enforce	Enforce	Enforce	Enforce
RWX =101	Enforce	Deny	EnforceNoX	Deny	Enforce	Enforce	Enforce	Enforce
RWX =111	Enforce	Deny	EnforceNoX	Deny	Enforce	Enforce	Exec-only	Enforce
RWX =010	A shadow stack region if Zicfiss is enabled, else reserved						Reserved	
RWX =011	Reserved						Reserved	

Figure 5. SPMP Encoding Table

Deny: Access fails.

Enforce: The R/W/X permissions specified in `spmpcfg` are enforced on accesses.

EnforceNoX: The R/W permissions specified in `spmpcfg` are enforced on accesses, while execute permission is revoked.

Reserved: 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 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 "RISC-V Privileged Architecture" for detailed information).

3.4. Address Matching

The A field in an SPMP entry's configuration register encodes the address-matching mode of the associated SPMP address register. The encoding of this field is shown in the following table:

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

It aligns with PMP/ePMP. Please refer to the "Address Matching" subsection of PMP in the "RISC-V Privileged Architecture" for detailed information.

Software may determine the SPMP granularity by writing zero to `spmpcfg[i]`*, then writing all ones to* `spmpaddr[i]`*, then reading back* `spmpaddr[i]`*. If* G *is the index of the least-significant bit set, the SPMP granularity is* 2^{G+2}

Software may also determine the size of physical address by setting `spmpcfg[i].A == 0b11`*, then writing all ones to* `spmpaddr[i]` *and reading back. (Please refer to the "NAPOT range encoding in PMP address and configuration registers" table in the "RISC-V Privileged Architecture" for detailed information.)*

The `spmpcfg[i].A` *is WARL, so an implementation can hardwire the address matching method to a specific one it desires.*

3.5. Matching Logic

- SPMP entries are statically prioritized.
- 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.
- 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 3.9).
- This matching is done irrespective of the SHARED, U, 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`.
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 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.

The execution environment should configure SPMP entry(s) to grant the most permissive access to S-mode. S-mode software can then further refine SPMP entries as desired.

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

SPMP CSRs are only accessible indirectly. This indirect CSR access avoids the potential cost in pipeline flushes due to the `switch` statement or series of `if` statements that would otherwise be required.

Each `miselect` represents an access to the corresponding SPMP CSRs. The `mireg` accesses `spmpaddr`, and the `mireg2` accesses `spmpcfg`. `mireg3`, `mireg4`, `mireg5`, and `mireg6` are read-only `0`.

There is no ordering guarantee between writes to different SPMP CSRs via indirect access. Executing an `SFENCE.VMA` instruction with `rs1=x0` and `rs2=x0` orders all following implicit and explicit accesses with respect to all preceding writes to SPMP CSRs.

The `spmpcfg[i].L` bit can only be reset when accessing in M-mode via `miselect`.

The `miselect` has the same view of `siselect` (see Section 3.7). For example, given an implementation with 64 PMP entries, where 48 entries are delegated to S-level. S-mode can access `SPMP[16..63]` via `siselect#16..63`. M-mode can access `SPMP[16..63]` via `siselect#16..63` or `miselect#16..63`. In such a case, both privileged mode attempts to access `SPMP[i]`, where `i < mpmpdeleg.pmpnum`, will read zero. Writes to such SPMP entries will be ignored.

`miselect` **number**	**indirect CSR access of** `mireg`
`miselect#0`	`mireg` → `spmpaddr[0]`, `mireg2` → `spmpcfg[0]`
`miselect#1`	`mireg` → `spmpaddr[1]`, `mireg2` → `spmpcfg[1]`
...	...
`miselect#63`	`mireg` → `spmpaddr[63]`, `mireg2` → `spmpcfg[63]`

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

Each `siselect` represents an access to the corresponding SPMP CSRs. The `sireg` accesses `spmpaddr`, and the `sireg2` accesses `spmpcfg`. `sireg3`, `sireg4`, `sireg5`, and `sireg6` are read-only `0`.

S-mode can set `spmpcfg[i].L` to lock an SPMP entry. When `spmpcfg[i].L` is set, SPMP writes via `siselect` are ignored, regardless of the privilege mode. Only M-mode access via `miselect` can reset `spmpcfg[i].L` (see Section 3.6).

Given an implementation with 64 PMP entries, where 48 entries are delegated to S-level. S-mode can only access `SPMP[16..63]` via `siselect#16..63`. Otherwise, the reads of out-of-index SPMP entries will return zero, and writes will be ignored.

`siselect` number	indirect CSR access of `sireg`
`siselect#0`	`sireg` → `spmpaddr[0]`, `sireg2` → `spmpcfg[0]`
`siselect#1`	`sireg` → `spmpaddr[1]`, `sireg2` → `spmpcfg[1]`
...	...
`siselect#63`	`sireg` → `spmpaddr[63]`, `sireg2` → `spmpcfg[63]`

The rationale for the fact that M-mode cannot reset `spmpcfg[i].L` *via* `siselect` *is to separate this permission by the CSR address space rather than only by privileged mode.*

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

When accessing `spmpcfg1` *via* `siselect` *and* `sireg2`*, all 16 bits will be written right-justified, despite the higher 8 bits and lower 8 bits not being stored in the same register array.*

Please refer to the `Sscsrind` *extension specification in the "RISC-V Privileged Architecture" for details on indirect CSR accesses.*

3.8. 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.

Please refer to Table "Encoding of satp MODE field" in the "RISC-V Privileged Architecture" for detailed information on the satp.MODE field.

3.9. 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 extension reuses page fault exception codes for SPMP faults since page faults are typically delegated to S-mode. S-mode software (i.e., an OS) can distinguish between SPMP-generated exceptions and page faults by checking `satp.mode`, since SPMP and paged virtual memory cannot be active simultaneously (as described in Section 3.8).

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 "RISC-V Privileged Architecture" 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 may remain undelegated and thus still be handled by machine mode.

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

Context switching with SPMP requires updating up to 64 address and 8 configuration registers (RV64). This chapter introduces the **`Sspmpsw`**, an optional extension for optimizing the context switching performance of the SPMP.

- In RV64: one 64-bit WARL CSR called `sspmpswitch` is added.
- In RV32: in addition to `sspmpswitch`, a 32-bit WARL CSR called `sspmpswitchh` is added, which is an alias of the upper half of `sspmpswitch`.

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

If an entry `i` is locked (i.e., `spmpcfg[i].L == 1`), then `sspmpswitch[i]` is **read-only** using **S-mode CSRs**.

Please refer to Chapter 5 for how software can use the optimization to reduce context switch overhead.

Accessing `sspmpswitch` CSR should follow the rule specified in Section 3.7. Specifically, given an implementation with 64 PMP_Entries, where 48 entries are delegated to S-level. The `sspmpswitch[16..63]` are used to control the activation of SPMP entries `[16..63]`. Writes to `sspmpswitch[0..15]` are ignored.

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

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

Utilizing `sspmpswitch` *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* `sspmpswitch` *(or two writes in RV32 systems:* `sspmpswitch` *and* `sspmpswitchh`*) 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.*

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. The static configuration can only be employed when the `Sspmpsw` extension is also implemented.
- **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 `sspmpswitch` 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.

The OS should allocate and configure SPMP entries for its own memory ranges, with the `spmpcfg[i].U` (the User bit) cleared to protect itself. After initializing the entries, the OS must also set the corresponding bits in the `sspmpswitch` register to activate these entries for itself.

Prior to launching the first user task, the OS sets the bits in `sspmpswitch` 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 `CSRRC` and `CSRRS` instructions, respectively. On RV32 systems, `sspmpswitch` and `sspmpswitchh` must be written continuously to prevent incomplete protection.

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.

The following sequence is recommended for dynamic reconfiguration:

1. **Disable Entries for the Outgoing Task**. A bitmask representing active entries (typically stored in the task's control block) should be passed as an argument. If the `Sspmpsw` extension is implemented, the OS should use the `CSRRC` instruction to clear the `sspmpswitch` bits associated with

the outgoing task's SPMP entries. Otherwise, the OS should clear the `spmpcfg[i].A` field associated with the outgoing task's SPMP entries.

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 `CSRRC` instruction;
   - Set the desired configuration using the `CSRRS` instruction.
4. **Enable Entries for the Incoming Task**. Pass a bitmask to enable the SPMP entries allocated to the incoming task. If the `Sspmpsw` extension is implemented, the OS should use the `CSRRS` instruction to write to `sspmpswitch`. Otherwise, the OS should set the `spmpcfg[i].A` field corresponding to the SPMP entries of the incoming task.

It is recommended that SPMP entries configured to protect the supervisor (i.e., entries with `spmpcfg[i].U == 0`) 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 Recomendations

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 (see Section 3.4).

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 Microcontroller Unit (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 naively 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 `sspmpswitch` (if the `Sspmpsw` extension is implemented), 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 `sspmpswitch` . In the **static configuration model**, this concern is relevant only for RV32 systems with more than 32 SPMP entries, where both `sspmpswitch` and `sspmpswitchh` must be updated in coordination.

To prevent asynchronous S-mode 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.