RISC-V Pointer Masking

Version 1.0-rc3, 08/2024: Frozen

Table of Contents

Preface

The proposed mechanism is an independent set of ISA extensions developed as part of the RISC-V J Extension. This document defines a set of individual extensions (`Smmpm`, `Smnpm`, `Ssnpm`, `Sspm`, `Supm`) that are collectively referred to as **pointer masking**.

Contributors

Authors: Adam Zabrocki, Martin Maas, Lee Campbell, RISC-V Runtime Integrity and J Extension Task Groups

Contributors: Krste Asanovic, Jecel Assumpcao, James Ball, Alexey Baturo, Allen Baum, Andrew Bresticker, Paul Donahue, Greg Favor, Andy Glew, Deepak Gupta, John Hauser, Samuel Holland, John Ingalls, James Kenney, Earl Killian, Christos Kotselidis, Dean Liberty, Philip Reames, Ian Rogers, Josh Scheid, Kostya Serebryany, Ved Shanbhogue, Boris Shingarov, Andrew Waterman, David Weaver, Foivos Zakkak, Members of the Runtime Integrity and J Extension Task Groups

Chapter 1. Introduction

RISC-V Pointer Masking (PM) is a feature that, when enabled, causes the CPU to ignore the upper bits of the effective address (these terms will be defined more precisely in the Background section). This allows these bits to be used in whichever way the application chooses. The version of the extension being described here specifically targets **tag checks**: When an address is accessed, the tag stored in the masked bits can be compared against a range-based tag. This is used for dynamic safety checkers such as HWASAN [1]. Such tools can be applied in all privilege modes (U, S and M).

HWASAN leverages tags in the upper bits of the address to identify memory errors such as use-after-free or buffer overflow errors. By storing a **pointer tag** in the upper bits of the address and checking it against a **memory tag** stored in a side table, it can identify whether a pointer is pointing to a valid location. Doing this without hardware support introduces significant overheads since the pointer tag needs to be manually removed for every conventional memory operation. Pointer masking support reduces these overheads.

Pointer masking only adds the ability to ignore pointer tags during regular memory accesses. The tag checks themselves can be implemented in software or hardware. If implemented in software, pointer masking still provides performance benefits since non-checked accesses do not need to transform the address before every memory access. Hardware implementations are expected to provide even larger benefits due to performing tag checks out-of-band and hardening security guarantees derived from these checks. We anticipate that future extensions may build on pointer masking to support this functionality in hardware.

It is worth mentioning that while HWASAN is the primary use-case for the current pointer masking extension, a number of other hardware/software features may be implemented leveraging Pointer Masking. Some of these use cases include sandboxing, object type checks and garbage collection bits in runtime systems. Note that the current version of the spec does not explicitly address these use cases, but future extensions may build on it to do so.

While we describe the high-level concepts of pointer masking as if it was a single extension, it is, in reality, a family of extensions that implementations or profiles may choose to individually include or exclude (see Section 2.7, “Pointer Masking Extensions”).

Chapter 2. Background

2.1. Definitions

We now define basic terms. Note that these rely on the definition of an "ignore" transformation, which is defined in Chapter 2.2.

- **Effective address (as defined in the RISC-V Base ISA):** A load/store effective address sent to the memory subsystem (e.g., as generated during the execution of load/store instructions). This does not include addresses corresponding to implicit accesses, such as page table walks.
- **Masked bits:** The upper PMLEN bits of an address, where PMLEN is a configurable parameter. We will use PMLEN consistently throughout this document to refer to this parameter.
- **Transformed address:** An effective address after the ignore transformation has been applied.
- **Address translation mode:** The MODE of the currently active address translation scheme as defined in the RISC-V privileged specification. This could, for example, refer to Bare, Sv39, Sv48, and Sv57. In accordance with the privileged specification, non-Bare translation modes are referred to as virtual-memory schemes. For the purpose of this specification, M-mode translation is treated as equivalent to Bare.
- **Address validity:** The RISC-V privileged spec defines validity of addresses based on the address translation mode that is currently in use (e.g., Sv57, Sv48, Sv39, etc.). For a virtual address to be valid, all bits in the unused portion of the address must be the same as the Most Significant Bit (MSB) of the used portion. For example, when page-based 48-bit virtual memory (Sv48) is used, load/store effective addresses, which are 64 bits, must have bits 63–48 all set to bit 47, or else a page-fault exception will occur. For physical addresses, validity means that bits XLEN-1 to PABITS are zero, where PABITS is the number of physical address bits supported by the processor.
- **NVBITS:** The upper bits within a virtual address that have no effect on addressing memory and are only used for validity checks. These bits depend on the currently active address translation mode. For example, in Sv48, these are bits 63-48.
- **VBITS:** The bits within a virtual address that affect which memory is addressed. These are the bits of an address which are used to index into page tables.

2.2. The "Ignore" Transformation

The ignore transformation differs depending on whether it applies to a virtual or physical address. For virtual addresses, it replaces the upper PMLEN bits with the sign extension of the PMLEN+1st bit.

```
transformed_effective_address =
  {{PMLEN{effective_address[XLEN-PMLEN-1]}}, effective_address[XLEN-
PMLEN-1:0]}
```

Listing 1. "Ignore" Transformation for virtual addresses, expressed in Verilog code.

If PMLEN is less than or equal to NVBITS for the largest supported address translation mode on a given architecture, this is equivalent to ignoring a subset of NVBITS. This enables cheap implementations that modify validity checks in the CPU instead of performing the sign extension.

When applied to a physical address, including guest-physical addresses (i.e., all cases except when the

active satp register's MODE field != Bare), the ignore transformation replaces the upper PMLEN bits with 0. This includes both the case of running in M-mode and running in other privilege modes with Bare address translation mode.

```
transformed_effective_address =
  {{PMLEN{0}}, effective_address[XLEN-PMLEN-1:0]}
```

Listing 2. "Ignore" Transformation for physical addresses, expressed in Verilog code.

This definition is consistent with the way that RISC-V already handles physical and virtual addresses differently. While the unused upper bits of virtual addresses are the sign-extension of the used bits (see the definition of "address validity" in Section 2.1, "Definitions"), the equivalent bits in physical addresses are zero-extended. This is necessary due to their interactions with other mechanisms such as Physical Memory Protection (PMP).

When pointer masking is enabled, the ignore transformation will be applied to every explicit memory access (e.g., loads/stores, atomics operations, and floating point loads/stores). The transformation **does not** apply to implicit accesses such as page table walks or instruction fetches. The set of accesses that pointer masking applies to is described in Section 2.6, "Memory Accesses Subject to Pointer Masking".

Pointer masking does not change the underlying address generation logic or permission checks. Under a fixed address translation mode, it is semantically equivalent to replacing a subset of instructions (e.g., loads and stores) with an instruction sequence that applies the ignore operation to the target address of this instruction and then applies the instruction to the transformed address. References to address translation and other implementation details in the text are primarily to explain design decisions and common implementation patterns.

Note that pointer masking is purely an arithmetic operation on the address that makes no assumption about the meaning of the addresses it is applied to. Pointer masking with the same value of PMLEN always has the same effect for the same type of address (virtual or physical). This ensures that code that relies on pointer masking does not need to be aware of the environment it runs in once pointer masking has been enabled, as long as the value of PMLEN is known, and whether or not addresses are virtual or physical. For example, the same application or library code can run in user mode, supervisor mode or M-mode (with different address translation modes) without modification.

A common scenario for such code is that addresses are generated by mmap system calls. This abstracts away the details of the underlying address translation mode from the application code. Software therefore needs to be aware of the value of PMLEN to ensure that its minimally required number of tag bits is supported. Section 2.4, "Determining the Value of PMLEN" covers how this value is derived.

2.3. Example

Table 1 shows an example of the pointer masking transformation on a virtual address when PM is enabled for RV64 under Sv57 (PMLEN=7).

Table 1. Example of PM address translation for RV64 under Sv57

Page-based profile	Sv57 on RV64
Effective Address	0xABFFFFFF12345678 NVBITS[1010101] VBITS[1111111111111111111111110001...000]
PMLEN	7
Mask	0x01FFFFFFFFFFFFFF NVBITS[0000000] VBITS[1111111111111111111111111111...111]
PMLEN+1st bit from the top (i.e., bit XLEN-PMLEN-1)	1
Transformed effective address	0xFFFFFFFF12345678 NVBITS[1111111] VBITS[1111111111111111111111110001...000]

If the address was a physical address rather than a virtual address with Sv57, the transformed address with PMLEN=7 would be 0x1FFFFFF12345678.

2.4. Determining the Value of PMLEN

From an implementation perspective, ignoring bits is deeply connected to the maximum virtual and physical address space supported by the processor (e.g., Bare, Sv48, Sv57). In particular, applying the above transformation is cheap if it covers only bits that are not used by **any** supported address translation mode (as it is equivalent to switching off validity checks). Masking NVBITS beyond those bits is more expensive as it requires ignoring them in the TLB tag, and even more expensive if the masked bits extend into the VBITS portion of the address (as it requires performing the actual sign extension). Similarly, when running in Bare or M mode, it is common for implementations to not use a particular number of bits at the top of the physical address range and fix them to zero. Applying the ignore transformation to those bits is cheap as well, since it will result in a valid physical address with all the upper bits fixed to 0.

The current standard only supports PMLEN=XLEN-48 (i.e., PMLEN=16 in RV64) and PMLEN=XLEN-57 (i.e., PMLEN=7 in RV64). A setting has been reserved to potentially support other values of PMLEN in future standards. In such future standards, different supported values of PMLEN may be defined for each privilege mode (U/VU, S/HS, and M).

Future versions of the pointer masking extension may introduce the ability to freely configure the value of PMLEN. The current extension does not define the behavior if PMLEN was different from the values defined above. In particular, there is no guarantee that a future pointer masking extension would define the ignore operation in the same way for those values of PMLEN.

2.5. Pointer Masking and Privilege Modes

Pointer masking is controlled separately for different privilege modes. The subset of supported privilege modes is determined by the set of supported pointer masking extensions. Different privilege modes may have different pointer masking settings active simultaneously and the hardware will automatically apply the pointer masking settings of the currently active privilege mode. A privilege mode's pointer masking setting is configured by bits in configuration registers of the next-higher privilege mode.

Note that the pointer masking setting that is applied only depends on the active privilege mode, not on the address that is being masked. Some operating systems (e.g., Linux) may use certain bits in the

address to disambiguate between different types of addresses (e.g., kernel and user-mode addresses). Pointer masking *does not* take these semantics into account and is purely an arithmetic operation on the address it is given.

Linux places kernel addresses in the upper half of the address space and user addresses in the lower half of the address space. As such, the MSB is often used to identify the type of a particular address. With pointer masking enabled, this role is now played by bit XLEN-PMLEN-1 and code that checks whether a pointer is a kernel or a user address needs to inspect this bit instead. For backward compatibility, it may be desirable that the MSB still indicates whether an address is a user or a kernel address. An operating system's ABI may mandate this, but it does not affect the pointer masking mechanism itself. For example, the Linux ABI may choose to mandate that the MSB is not used for tagging and replicates bit XLEN-PMLEN-1 bit (note that for such a mechanism to be secure, the kernel needs to check the MSB of any user mode-supplied address and ensure that this invariant holds before using it; alternatively, it can apply the transformation from Listing 1 or 2 to ensure that the MSB is set to the correct value).

2.6. Memory Accesses Subject to Pointer Masking

Pointer masking applies to all explicit memory accesses. Currently, in the Base and Privileged ISAs, these are:

- **Base Instruction Set**: LB, LH, LW, LBU, LHU, LWU, LD, SB, SH, SW, SD.
- **Atomics**: All instructions in RV32A and RV64A.
- **Floating Point**: FLW, FLD, FLQ, FSW, FSD, FSQ.
- **Compressed**: All instructions mapping to any of the above, and C.LWSP, C.LDSP, C.LQSP, C.FLWSP, C.FLDSP, C.SWSP, C.SDSP, C.SQSP, C.FSWSP, C.FSDSP.
- **Hypervisor Extension**: HLV.*, HSV.* (in some cases; see Section 3.1, "Ssnpm").
- **Cache Management Operations**: All instructions in Zicbom, Zicbop and Zicboz.
- **Vector Extension**: All vector load and store instructions in the ratified RVV 1.0 spec.
- **Assorted**: FENCE, FENCE.I (if the currently unused address fields become enabled in the future).

This list will grow over time as new extensions introduce new instructions that perform explicit memory accesses.

For other extensions, pointer masking applies to all explicit memory accesses by default. Future extensions may add specific language to indicate whether particular accesses are or are not included in pointer masking.

It is worth noting that pointer masking is not applied to SFENCE., HFENCE.*, SINVAL.*, or HINVAL.*. When such an operation is invoked, it is the responsibility of the software to provide the correct address.*

MPRV and SPVP affect pointer masking as well, causing the pointer masking settings of the effective privilege mode to be applied. When MXR is in effect at the effective privilege mode where explicit memory access is performed, pointer masking does not apply.

Cache Management Operations (CMOs) must respect and take into account pointer masking. Otherwise, a few serious security problems can appear, including:

- *CBO.ZERO may work as a STORE operation. If pointer masking is not respected, it would be possible to write to memory bypassing the mask enforcement.*
- *If CMOs did not respect pointer masking, it would be possible to weaponize this in a side-channel attack. For example, U-mode would be able to flush a physical address (without masking) that it should not be permitted to.*

Pointer masking only applies to accesses generated by instructions on the CPU (including CPU extensions such as an FPU). E.g., it does not apply to accesses generated by page table walks, the IOMMU, or devices.

Pointer Masking does not apply to DMA controllers and other devices. It is therefore the responsibility of the software to manually untag these addresses.

Misaligned accesses are supported, subject to the same limitations as in the absence of pointer masking. The behavior is identical to applying the pointer masking transformation to every constituent aligned memory access. In other words, the accessed bytes should be identical to the bytes that would be accessed if the pointer masking transformation was individually applied to every byte of the access without pointer masking. This ensures that both hardware implementations and emulation of misaligned accesses in M-mode behave the same way, and that the M-mode implementation is identical whether or not pointer masking is enabled (e.g., such an implementation may leverage MPRV to apply the correct privilege mode's pointer masking setting).

No pointer masking operations are applied when software reads/writes to CSRs, including those meant to hold addresses. If software stores tagged addresses into such CSRs, data load or data store operations based on those addresses are subject to pointer masking only if they are explicit (Section 2.6, "Memory Accesses Subject to Pointer Masking") and pointer masking is enabled for the privilege mode that performs the access. The implemented WARL width of CSRs is unaffected by pointer masking (e.g., if a CSR supports 52 bits of valid addresses and pointer masking is supported with PMLEN=16, the necessary number of WARL bits remains 52 independently of whether pointer masking is enabled or disabled).

In contrast to software writes, pointer masking **is applied** for hardware writes to a CSR (e.g., when the hardware writes the transformed address to `stval` when taking an exception). Pointer masking is also applied to the memory access address when matching address triggers in debug.

For example, software is free to write a tagged or untagged address to `stvec`, but on trap delivery (e.g., due to an exception or interrupt), pointer masking **will not be applied** to the address of the trap handler. However, pointer masking **will be applied** by the hardware to any address written into `stval` when delivering an exception.

The rationale for this choice is that delivering the additional bits may add overheads in some hardware implementations. Further, pointer masking is configured per privilege mode, so all trap handlers in supervisor mode would need to be careful to configure pointer masking the same way as user mode or manually unmask (which is expensive).

2.7. Pointer Masking Extensions

Pointer masking refers to a number of separate extensions, all of which are privileged. This approach is used to capture optionality of pointer masking features. Profiles and implementations may choose to support an arbitrary subset of these extensions and must define valid ranges for their corresponding values of PMLEN.

Extensions:

- **Ssnpm**: A supervisor-level extension that provides pointer masking for the next lower privilege mode (U-mode), and for VS- and VU-modes if the H extension is present.
- **Smnpm**: A machine-level extension that provides pointer masking for the next lower privilege mode (S/HS if S-mode is implemented, or U-mode otherwise).
- **Smmpm**: A machine-level extension that provides pointer masking for M-mode.

See Chapter 3, *ISA Extensions* for details on how each of these extensions is configured.

In addition, the pointer masking standard defines two extensions that describe an execution environment but have no bearing on hardware implementations. These extensions are intended to be used in profile specifications where a User profile or a Supervisor profile can only reference User level or Supervisor level pointer masking functionality, and not the associated CSR controls that exist at a higher privilege level (i.e., in the execution environment).

- **Sspm**: An extension that indicates that there is pointer-masking support available in supervisor mode, with some facility provided in the supervisor execution environment to control pointer masking.
- **Supm**: An extension that indicates that there is pointer-masking support available in user mode, with some facility provided in the application execution environment to control pointer masking.

The precise nature of these facilities is left to the respective execution environment.

Pointer masking only applies to RV64. In RV32, trying to enable pointer masking will result in an illegal WARL write and not update the pointer masking configuration bits (see Chapter 3, *ISA Extensions* for details). The same is the case on RV64 or larger systems when UXL/SXL/MXL is set to 1 for the corresponding privilege mode. Note that in RV32, the CSR bits introduced by pointer masking are still present, for compatibility between RV32 and larger systems with UXL/SXL/MXL set to 1. Setting UXL/SXL/MXL to 1 will clear the corresponding pointer masking configuration bits.

Note that setting UXL/SXL/MXL to 1 and back to 0 does not preserve the previous values of the PMM bits. This includes the case of entering an RV32 virtual machine from an RV64 hypervisor and returning.

Chapter 3. ISA Extensions

This section describes the pointer masking extensions `Smmpm`, `Smnpm` and `Ssnpm`. All of these extensions are privileged ISA extensions and do not add any new CSRs. For the definitions of `Sspm` and `Supm`, see Section 2.7, "Pointer Masking Extensions".

Future extensions may introduce additional CSRs to allow different privilege modes to modify their own pointer masking settings. This may be required for future use cases in managed runtime systems that are not currently addressed as part of this extension.

Each extension introduces a 2-bit WARL field (`PMM`) that may take on the following values to set the pointer masking settings for a particular privilege mode.

Table 2. Possible values of `PMM` *WARL field.*

Value	Description
00	Pointer masking is disabled (PMLEN=0)
01	Reserved
10	Pointer masking is enabled with PMLEN=XLEN-57 (PMLEN=7 on RV64)
11	Pointer masking is enabled with PMLEN=XLEN-48 (PMLEN=16 on RV64)

All of these fields are read-only 0 on RV32 systems.

3.1. Ssnpm

`Ssnpm` adds a new 2-bit WARL field (`PMM`) to bits 33:32 of `senvcfg`. Setting `PMM` enables or disables pointer masking for the next lower privilege mode (U/VU mode), according to the values in Table 2.

In systems where the H Extension is present, `Ssnpm` also adds a new 2-bit WARL field (`PMM`) to bits 33:32 of `henvcfg`. Setting `PMM` enables or disables pointer masking for VS-mode, according to the values in Table 2. Further, a 2-bit WARL field (`HUPMM`) is added to bits 49:48 of `hstatus`. Setting `hstatus.HUPMM` enables or disables pointer masking for `HLV.*` and `HSV.*` instructions in U-mode, according to the values in Table 2, when their explicit memory access is performed as though in VU-mode. In HS- and M-modes, pointer masking for these instructions is enabled or disabled by `senvcfg.PMM`, when their explicit memory access is performed as though in VU-mode. Setting `henvcfg.PMM` enables or disables pointer masking for `HLV.*` and `HSV.*` when their explicit memory access is performed as though in VS-mode.

The hypervisor should copy the value written to `senvcfg.PMM` *by the guest to the* `hstatus.HUPMM` *field prior to invoking* `HLV.*` *or* `HSV.*` *instructions in U-mode.*

The memory accesses performed by the `HLVX.*` instructions are not subject to pointer masking.

`HLVX.*` *instructions, designed for emulating implicit access to fetch instructions from guest memory, perform memory accesses that are exempt from pointer masking to facilitate this emulation. For the same reason, pointer masking does not apply when MXR is set.*

3.2. Smnpm

`Smnpm` adds a new 2-bit WARL field (`PMM`) to bits 33:32 of `menvcfg`. Setting `PMM` enables or disables pointer masking for the next lower privilege mode (S-/HS-mode if S-mode is implemented, or U-mode otherwise), according to the values in Table 2.

The type of address determines which type of pointer masking is applied. For example, when running with virtualization in VS/VU mode with `vsatp.MODE` *= Bare, physical address pointer masking (zero extension) applies.*

3.3. Smmpm

`Smmpm` adds a new 2-bit WARL field (`PMM`) to bits 33:32 of `mseccfg`. The presence of `Smmpm` implies the presence of the `mseccfg` register, even if it would not otherwise be present. Setting `PMM` enables or disables pointer masking for M mode, according to the values in Table 2.

3.4. Interaction with SFENCE.VMA

Since pointer masking applies to the effective address only and does not affect any memory-management data structures, no SFENCE.VMA is required after enabling/disabling pointer masking.

3.5. Interaction with Two-Stage Address Translation

Guest physical addresses (GPAs) are 2 bits wider than the corresponding virtual address translation modes, resulting in additional address translation schemes Sv32x4, Sv39x4, Sv48x4 and Sv57x4 for translating guest physical addresses to supervisor physical addresses. When running with virtualization in VS/VU mode with `vsatp.MODE` = Bare, this means that those two bits may be subject to pointer masking, depending on `hgatp.MODE` and `senvcfg.PMM`/`henvcfg.PMM` (for VU/VS mode). If `vsatp.MODE` != BARE, this issue does **not** apply.

An implementation could mask those two bits on the TLB access path, but this can have a significant timing impact. Alternatively, an implementation may choose to "waste" TLB capacity by having up to 4 duplicate entries for each page. In this case, the pointer masking operation can be applied on the TLB refill path, where it is unlikely to affect timing. To support this approach, some TLB entries need to be flushed when PMLEN changes in a way that may affect these duplicate entries.

To support implementations where (XLEN-PMLEN) can be less than the GPA width supported by `hgatp.MODE`, hypervisors should execute an `HFENCE.GVMA` with rs1=x0 if the `henvcfg.PMM` is changed from or to a value where (XLEN-PMLEN) is less than GPA width supported by the `hgatp` translation mode of that guest. Specifically, these cases are:

- `PMLEN=7` and `hgatp.MODE=sv57x4`
- `PMLEN=16` and `hgatp.MODE=sv57x4`
- `PMLEN=16` and `hgatp.MODE=sv48x4`

`Smmpm` *implementations need to satisfy max(largest supported virtual address size, largest supported supervisor physical address size) ⇐ (XLEN - PMLEN) bits to avoid any masking logic on the TLB access path.*

Implementation of an address-specific `HFENCE.GVMA` should either ignore the address argument, or should ignore the top masked GPA bits of entries when comparing for an address match.

3.6. Number of Masked Bits

As described in Section 2.4, "Determining the Value of PMLEN", the supported values of PMLEN may depend on the effective privilege mode. The current standard only defines PMLEN=XLEN-48 and PMLEN=XLEN-57, but this assumption may be relaxed in future extensions and profiles. Trying to enable pointer masking in an unsupported scenario represents an illegal write to the corresponding pointer masking enable bit and follows WARL semantics. Future profiles may choose to define certain combinations of privilege modes and supported values of PMLEN as mandatory.

An option that was considered but discarded was to allow implementations to set PMLEN depending on the active addressing mode. For example, PMLEN could be set to 16 for Sv48 and to 25 for Sv39. However, having a single value of PMLEN (e.g., setting PMLEN to 16 for both Sv39 and Sv48 rather than 25) facilitates TLB implementations in designs that support Sv39 and Sv48 but not Sv57. 16 bits are sufficient for current pointer masking use cases but allow for a TLB implementation that matches against the same number of virtual tag bits independently of whether it is running with Sv39 or Sv48. However, if Sv57 is supported, tag matching may need to be conditional on the current address translation mode.

Bibliography

[1] Serebryany, Kostya, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. "Memory tagging and how it improves C/C++ memory safety." arXiv preprint arXiv:1802.09517 (2018).