RISC-V Pointer Masking

Version 0.5.4, 04/2023: Development

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 (`Zjpm`, `Ssjpm`, `Smjpm`, `Smjpmbare16`) that are collectively referred to as **pointer masking**.

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

Contributors: Jecel Assumpcao, Allen Baum, Paul Donahue, Greg Favor, Andy Glew, Deepak Gupta, John Ingalls, Christos Kotselidis, Philip Reames, Ian Rogers, Josh Scheid, Kostya Serebryany, Boris Shingarov, 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 MMU to ignore the top N 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 specifically targets **Tag checks**: When an address is accessed, the tag stored in the masked bits is 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).

Tag checks can be implemented in software or hardware. If implemented in software, pointer masking still provides performance benefits since all 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 **Tag checks** is the primary use-case for the current extension, but a number of future hardware/software features may be implemented leveraging Pointer Masking. Some of these use cases include:

- **Sandbox enforcement:** When an address is accessed, the masked bits are checked against a particular sandbox tag to enforce that addresses are limited to a particular range in memory. This is used for isolating heap and runtime memory in a managed runtime, and isolation of untrusted code in M mode.
- **Read barriers:** When an address is accessed, the masked bits are checked against (e.g.,) a generation in a garbage collected environment, redirecting to slow path code on mismatch.
- **Object type checks:** When an address is accessed, the masked bits are checked against a fixed type tag in an object-oriented runtime, redirecting to slow path code on mismatch.

Note that some of these use cases may require extending pointer masking to instruction fetches. This feature may be introduced as a separate extension in the future.

The `Zjpm` pointer masking extension depends on the `Zicsr` extension. 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, “Constituent 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 addresses 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 top N bits of an address, where N is a configurable parameter (we will use N 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 privilege mode and address translation mode that is currently in use (e.g., Sv57, Sv48, Sv39, etc.). For an 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 virtual addresses). 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. This definition only applies to virtual addresses.
- **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 (Listing 1) is expressed independently of the current address translation mode. Conceptually, it replaces the top N bits with the sign extension of the N+1st bit from the top.

Listing 1. "Ignore" Transformation expressed in Verilog code.

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

If N is smaller or equal to NVBITS for the current address translation mode, this is equivalent to ignoring a subset of NVBITS. This enables cheap implementations that disable validity checks in the MMU instead of performing the sign extension.

When pointer masking is enabled, this 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. 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 implemenation 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 N always has the same effect. 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 N is known. 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. These system calls abstract away the details of the underlying address translation mode from the application code. Software therefore needs to be aware of the value of N to ensure that its minimally required number of tag bits is supported. Section 2.4, “Determining the Value of N” covers how this value is derived.

2.3. Example

Table 1 shows an example of address translation when PM is enabled for RV64 under Sv57 with N=7.

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

Page-based profile	**Sv57 on RV64**
Effective Address	0xAAFFFFFF12345678 NVBITS[1010101] VBITS[111111111111111111111111110001...000]
N	7
Mask	0x01FFFFFFFFFFFFFF NVBITS[0000000] VBITS[111111111111111111111111111111...111]
N+1st bit from the top	1
Transformed address	0xFFFFFFFF12345678 NVBITS[1111111] VBITS[111111111111111111111111110001...000]

2.4. Determining the Value of N

From an implementation perspective, ignoring bits is deeply connected to the address translation mode (e.g., Bare, Sv48, Sv57). In particular, applying the above transformation is cheap if it covers only the NVBITS (as it is equivalent to switching off validity checks) but 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 pin them to zero. Applying the ignore transformation to all but the lowest of those bits is cheap as well (the lowest bit is still pinned to zero to ensure that sign extension will result in a valid physical address where all the top bits are zeros).

The specified extensions **do not** support N to be configurable. Instead, N is a static function of the currently active address translation mode. Specifically, N is the number of NVBITS of this address translation mode (e.g., 16 for Sv48) if it is a virtual-memory scheme. Pointer masking in Bare and M-mode address translation mode is a separate extension (`Smjpmbare16`). If this extension is present, N is hardcoded to be 16 in Bare and M-mode (otherwise it is 0).

Future versions of the pointer masking extension may introduce the ability to configure the value of N. The current extension does not define the behavior if N 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 N.

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 delegation mechanism allows higher privilege modes to optionally restrict lower privilege modes from changing their own pointer masking settings (Chapter 3).

Since pointer masking forms the foundation for security-related mechanisms, configurability per-privilege mode is critical for avoiding a window in time (e.g., when jumping into a trap handler) when the incorrect masking is applied.

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 the N+1st bit 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 the N+1st bit.

2.6. Memory Accesses Subject to Pointer Masking

Pointer masking applies to all explicit memory accesses. 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, LFQ, 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.
- **Memory Management**: FENCE, FENCE.I (if the currently unused address fields become enabled in the future), SFENCE.*, HFENCE.*, SINVAL.*, HINVAL.*.

MPRV affects pointer masking as well, causing the pointer masking settings of the effective privilege mode to be applied. Just like in the absence of pointer masking, MPRV does not affect instruction fetch and the current rather than the effective privilege mode's pointer masking settings are applied to instructions. Pointer masking also applies to HLV, HLVX and HSV instructions.

For other extensions, pointer masking applies to all explicit memory accesses by default. This includes, e.g., vector loads and stores. Future extensions may add specific language to indicate whether particular accesses are or are not included in pointer masking.

> It is important to note that Cache Management Operation (CMO) must respect and take into account Pointer masking. Otherwise, a few serious security problem can appear, including:
>
> - CBO.ZERO may work as a STORE operation and if Pointer Masking is not respected, it will be possible to write to the memory bypassing the masking enforcement
> - If CMO won't respect PM it would be possible to weaponize it in side-channel attack e.g., U-mode would be able flush PA 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). For example, it does not apply to accesses generated by the IOMMU or devices.

Misaligned accesses are supported, subject to the same limitations that would exist in the absence of pointer masking. If a misaligned access crosses the boundary to the masked bits, it will behave as if the ignore transformation is applied to each constituent access and thus "wrap around". This

ensures that both hardware implementations and emulation of misaligned accesses in M-mode behave the same way.

No pointer masking operations are applied when software reads/writes to CSRs meant to hold addresses. If software needs to put 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. 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), no pointer masking will be applied.

There is no guarantee that reading a CSR containing an address will retain any masked bits, even if a tagged address was previously written into this CSR.

2.7. Constituent Extensions

As indicated in Chapter 1, *Introduction*, pointer masking refers to a number of separate extensions. This approach is used to capture optionality of pointer masking features. Profiles and implementations may choose to support an arbitrary subset of these extensions.

Unprivileged Extension:

- **Zjpm**: U/VU-mode pointer masking is available if and only if this extension is present.

Privileged Extensions:

- **Ssjpm**: S/HS/VS-mode pointer masking is available if and only if this extension is present.
- **Smjpm**: M-mode pointer masking is available if and only if this extension is present.
- **Smjpmbare16**: Pointer masking applies to the Bare address translation mode, across all supported privilege modes. If this extension is present, N is hardcoded to 16 in Bare address translation mode. If it is not present, N is hardcoded to 0 in Bare address translation mode.

Pointer masking only applies to RV64. On RV32, trying to enable pointer masking will cause an exception (see Chapter 3, *Unprivileged ISA Extensions* and Chapter 4, *Privileged 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 even on RV32, the CSRs introduced by pointer masking are still present, for compatibility between RV32 and larger systems with UXL/SXL/MXL set to 1.

Chapter 3. Unprivileged ISA Extensions

This section describes the `Zjpm` user mode extension. The extension adds a new configuration CSR (upm). This CSR controls pointer masking in (V)U-mode and is read/write.

Most bits in the register are currently unused. The remaining bits are reserved for future use by extensions that leverage pointer masking functionality. This will reduce the number of registers that need to be context-switched in the future.

*Figure 1: Pointer Masking register for U-mode (**upm**)*

upm[XLEN-1:1]	**upm[0]**
WPRI	uxpmen (WARL)

We now describe the meaning of the uxpmen field. This field controls whether pointer masking is active for the corresponding privilege mode. It defaults and resets to 0. A value of 1 indicates that pointer masking is enabled. Non user-mode components of the system may determine that user mode is not allowed to modify uxpmen. If this is the case, a write to upm that modifies the uxpmen bit results in an exception.

Enabling and disabling is expected to be a frequent operation. The uxpmen field has therefore been placed at the low end of the CSR, to allow pointer masking to be enabled/disabled with a single CSRRSI or CSRRCI instruction.

Note that the number of bits that are being masked (N) is outside the control of user mode and needs to be communicated to user mode using an approach such as a syscall or other discovery mechanism. This communication mechanism is part of the ABI and not the ISA, and beyond the scope of this document.

Chapter 4. Privileged ISA Extensions

This section describes the privileged pointer masking extensions `Ssjpm`, `Smjpm` and `Smjpmbare16`. Pointer masking adds several new CSRs, depending on which extensions are present.

4.1. Added CSRs

The following CSRs are added for each of the extensions:

- **Ssjpm**: spm, vspm
- **Smjpm**: mpm

All CSRs are read/write. Pointer masking in M-mode is controlled by mpm. The spm and vspm registers control pointer masking in HS/S-mode and VS-mode, and follow the conventions established by the hypervisor extension. Specifically, spm controls pointer masking when running in unvirtualized (H)S-mode (which includes unvirtualized OS kernels as well as Type 1 and Type 2 hypervisors). When running in virtualized mode (VS), the content of the vspm register will be treated as the spm register.

The layout of the four CSRs can be found in Figure 2a, 2b, and 2c for M, (H)S, and VS-mode.

Most bits in the registers are currently unused. The remaining bits are reserved for future use by extensions that leverage pointer masking functionality. This will reduce the number of registers that need to be context-switched in the future. While the layout of the registers is currently identical, this may change with future extensions.

The upm register (Chapter 3, *Unprivileged ISA Extensions*) is visible to all privilege modes. The spm register is visible to S-mode and M-mode (including HS/VS-mode if the hypervisor extension is present). The vspm register is visible to HS-mode and M-mode if the hypervisor extension is present. The mpm register is only visible to M-mode.

*Figure 2a: Pointer Masking register for M-mode (**mpm**)*

mpm[XLEN-1:1]	**mpm[0]**
WPRI	mxpmen (WARL)

*Figure 2b: Pointer Masking register for S-mode (**spm**)*

spm[XLEN-1:1]	**spm[0]**
WPRI	sxpmen (WARL)

*Figure 2c: Pointer Masking register for virtualized S-mode (**vspm**)*

vspm[XLEN-1:1]	**vspm[0]**
WPRI	vsxpmen (WARL)

We now describe the meaning of the mxpmen/sxpmen/vsxpmen field. The field controls whether

pointer masking is active for the corresponding privilege mode. It defaults and resets to 0. A value of 1 indicates that pointer masking is enabled. Higher privilege modes may determine that a particular privilege mode is not allowed to modify its *xpmen field. If this is the case, a write to the CSR that modifies the *xpmen bit results in an exception.

4.2. Additions to menvcfg, senvcfg and henvcfg

`Ssjpm` reserves two new 1-bit WARL fields (`spmself` and `upmself`) in `menvcfg`. It also reserves a 1-bit WARL field in `henvcfg` and `senvcfg` that aliases `upmself` (as S-mode cannot access `menvcfg`).

The fields control whether a privilege mode can modify its own pointer masking CSR. For example, if `upmself` is set to 0 at the start of a write to the `upm` CSR and the current privilege mode is U-mode, the write will be ignored. If the field is set to 1, the CSR write will proceed as usual. These rules apply whether or not pointer masking is enabled.

If `Zjpm` is present but `Ssjpm` is not, the behavior is the same as if the `upmself` field was set to 1.

4.3. Number of Masked Bits

As described in Section 2.4, “Determining the Value of N”, the number of masked bits depends on the currently active address translation mode. The table below describes the number of masked bits (N) as a function of the current address translation mode.

Application-level software does not have access to the current address translation mode. It is the responsibility of the OS to communicate the number of masked bits to the U-mode process in order to enable U-mode pointer masking.

Figure 3: Number of masked bits

Address Translation Mode	Masked Bits (N)
Sv39	16
Sv48	16
Sv57	7
Sv64	0 (to be addressed in future standards)
Bare	16 if Smjpmbare16 is present (0 otherwise)
M-mode	16 if Smjpmbare16 is present (0 otherwise)

We set N to 16 for Sv39 rather than 25 to facilitate 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. If Sv57 is supported, tag matching needs to be conditional on the current address translation mode. Note that the number of Masked Bits supported for each address translation mode may change in the future (e.g., future extensions may require N=25 for Sv39).

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).


RISC-V