RISC-V Pointer Masking

Version 0.5.1, 02/2023: Development

Table of Contents

Preface

This document is released under the Creative Commons Attribution 4.0 International License.

The proposed mechanism is an independent ISA extension developed as part of the RISC-V J Extension. This document defines a set of individual subextensions (`Zjpmu`, `Zjpms`, `Zjpmm`, `Zjpmbare16`, `Zjpminst`) that are collectively referred to by the identifier `Zjpm`.

Authors: Adam Zabrocki, Martin Maas, Lee Campbell, RISC-V TEE 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 TEE 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. Most commonly, these bits are used to store various types of tags, which can be leveraged by a number of hardware/software features, including sandboxing mechanisms and dynamic safety checkers such as HWASAN [1].

This extension only adds the pointer masking functionality. Extensions that make use of the masked bits will be ratified independently and layered on top of the basic pointer masking functionality. Examples of such extensions include:

- **Tag checks:** When a masked 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).
- **Sandbox enforcement:** When a masked 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 a masked 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 a masked 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.

All of these use cases 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 unmask 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.

The Zjpm extension depends on the Zicsr extension. While we describe Zjpm 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 3.1, “Subextensions”).

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):** An address generated by the instruction fetch and load/store effective addresses sent to the memory subsystem. There is no special distinction of physical vs. virtual memory. 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).
- **Masked address:** An effective address after the ignore transformation has been applied to it.
- **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 canonicity:** The RISC-V privileged spec defines canonicity of an address based on the privilege mode and address translation mode that is currently in use (e.g., Sv57, Sv48, Sv39, etc.). Canonicity enforces that for the translated addresses 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, instruction fetch addresses and 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 canonicity 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 canonicity 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, with the exception of instruction fetches if the corresponding pointer masking feature is enabled. The full set of accesses that pointer masking applies to is shown in Section 2.6, "Memory Accesses Subject to Pointer Masking".

Pointer masking does not change the underlying address generation logic or permission checks. For data, pointer masking 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 masked 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[0111111111111111111111110001...000]
N	7
Mask	0x01FFFFFFFFFFFFFF NVBITS[0000000] VBITS[11111111111111111111111111111...111]
N+1st bit from the top	1
Transformed address	0xFFFFFFFF12345678 NVBITS[1111111] VBITS[11111111111111111111111110001...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 canonicity 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 extension **does 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 subextension (`Zjpmbare16`). If this subextension 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 subextensions of `Zjpm`. 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. This will be particularly important for the future sandboxing use case, where such a window would allow code to escape its sandbox.

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 can be 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. Since pointer masking is defined based on sign extension, no other parts of the operating system need to be changed since the masked address will still point to the correct part of the address space for both kernel and user addresses. The operating system needs to ensure to keep the N+1st bit available for determining the type of address (e.g., mmap calls on Linux would set this bit to zero). Further, the Linux ABI may mandate that the MSB of the address is not used for tagging and replicates the N+1st bit; this is necessary since the Linux kernel contains many places where kernel and user addresses are disambiguated by comparing them to a threshold.

2.6. Memory Accesses Subject to Pointer Masking

Pointer masking applies to all explicit memory accesses in the Base and Privileged ISAs:

- **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.*.

Pointer masking *does not* apply to HLV, HLVX and HSV instructions.

In the absence of additional specification changes, pointer masking only applies to the above instructions. ISA extensions decide individually which of their instructions are subject to pointer masking. By default, it will not apply and thus not change any existing behavior.

If pointer masking is used for sandboxing, any extension that does not apply pointer masking cannot be used within sandboxed code as this would circumvent the sandbox. Further, not applying pointer masking would significantly reduce the benefit of other extensions such as CMOs, as the masking operation would need to be applied manually.

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 `Zjpm`. If a misaligned access crosses the boundary to the masked bits, a page-fault exception will occur. This is identical to the behavior in the absence of pointer masking, since such an operation would result in multiple accesses, at least one of which would be to a non-canonical address.

No pointer masking operations are applied when software reads/writes to CSRs meant to hold addresses. If software needs to put tagged pointers into such CSRs, it can do so. However, instruction fetch, data load or data store operations based on those addresses are subject to pointer masking only if they are explicitly included (Section 2.6, “Memory Accesses Subject to Pointer Masking”) and pointer masking is enabled for this privilege mode. For example, software is free to write tagged or untagged pointer to `stvec`, but when a trap delivery happens (due to an exception or interrupt), pointer masking will only be applied to the address if S-mode has instruction pointer masking enabled.

2.7. Instruction Fetches

Pointer masking can optionally be applied to instruction fetches. The availability of this feature is determined by the presence of the `Zjpminst` subextension. On implementations where the feature is available, it can be enabled or disabled by software.

When enabled, the ignore operation applies to every instruction fetch, including those resulting from monotonic PC increases due to straight line execution, control transfers (e.g., branches and

direct/indirect jumps and uret/sret/mret).

URET, SRET and MRET apply the pointer masking setting of the privilege mode they are returning to. Similarly, other privilege mode transfers (e.g., traps or exceptions) apply the pointer masking setting of the mode they are entering.

During trap delivery, `stval` will contain the unmasked pointer, even when instruction pointer masking is enabled.

Chapter 3. ISA Extension

3.1. Subextensions

As indicated in Chapter 1, *Introduction*, `Zjpm` is a conceptual way of referring to a number of separate subextensions. This approach is used to capture optionality of features in the extension. Profiles and implementations may choose to support an arbitrary subset of these extensions:

- **Zjpmu**: U/VU-mode pointer masking is available if and only if this extension is present.
- **Zjpms**: S/HS/VS-mode pointer masking is available if and only if this extension is present.
- **Zjpmm**: M-mode pointer masking is available if and only if this extension is present.
- **Zjpmbare16**: 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.
- **Zjpminst**: Pointer masking for instructions is supported, across all supported privilege modes. In the absence of this extension, the corresponding CSR bits (`*inst`) are hard-wired to 0.

3.2. Added CSRs

Zjpm adds up to four new configuration CSRs, depending on which subextensions are present. The following CSRs are added for each of the subextensions:

- **Zjpmu**: upm
- **Zjpms**: spm, vspm
- **Zjpmm**: mpm

All four CSRs are read/write. Pointer masking in (V)U-mode is controlled by upm, 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 1a, 1b, 1c and 1d for M, (H)S, VS and (V)U-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 four registers is currently identical, this may change with future extensions.

The upm register 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 1a: Pointer Masking register for M-mode ***(mpm)***

mpm[XLEN-1:2]	mpm[1]	mpm[0]
WPRI	minst (WARL)	menable (WARL)

*Figure 1b: Pointer Masking register for S-mode (**spm**)*

spm[XLEN-1:2]	spm[1]	spm[0]
WPRI	sinst (WARL)	senable (WARL)

*Figure 1c: Pointer Masking register for virtualized S-mode (**vspm**)*

vspm[XLEN-1:2]	vspm[1]	vspm[0]
WPRI	vsinst (WARL)	vsenable (WARL)

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

upm[XLEN-1:2]	upm[1]	upm[0]
WPRI	uinst (WARL)	uenable (WARL)

We now describe the meaning of each of these fields. All fields default and reset to 0.

3.3. menable/senable/vsenable/uenable

This field controls whether pointer masking is active for the corresponding privilege mode. A value of 1 indicates that pointer masking is enabled.

3.4. minst/sinst/vsinst/uinst

If pointer masking is enabled for this privilege mode, inst=1 indicates that it applies to both instruction and data accesses. The semantics of this are described in Section 2.7, “Instruction Fetches”. If inst=0, pointer masking only applies to data accesses. If pointer masking is disabled (enable = 0), this field is ignored.

This is a WARL field. Pointer masking for instructions is only supported if the `Zjpminst` extension is present. If it is not present, an attempt to write 1 to this field may result in the field being set to 0. This does not affect any other field updates.

Enabling and disabling is expected to be a more frequent operation than other field updates. The enable 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.

3.5. Additions to menvcfg, senvcfg and henvcfg

`Zjpm` 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. If this field is set to 0 at the start of a write to the CSR and the current privilege mode is the one that this CSR corresponds to (e.g., U-mode for upm), 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.

3.6. 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.

Figure 2: Number of masked bits

Address Translation Mode	Masked Bits (N)
Sv39	25
Sv48	16
Sv57	7
Sv64	0 (to be addressed in future standards)
Bare	16 if Zjpmbare16 is present (0 otherwise)
M-mode	16 if Zjpmbare16 is present (0 otherwise)

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.

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