RISC-V Pointer Masking

Version 0.4, 12/2022: Development

Table of Contents

Preface

The proposed mechanism is an independent ISA extension developed as part of the RISC-V J Extension. The identifier of this extension is `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].
- **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.

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.
- **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. For untranslated addresses (i.e., for Bare and M-mode) canonicity requirement is for the high bits to be all zeros.
- **NVBITS:** The upper bits within an 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. For Bare mode, this is implementation-dependent.
- **VBITS:** The bits within an address that affect which memory is addressed. If virtual memory is enabled, 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 (i.e., subject to the current address translation mode). This includes, for example, 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 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 N bits always has the same effect, independently of the underlying address translation mode or privilege mode. 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. 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.

2.3. Example

Table 1 shows an example of address translation when PM is enabled for RV64 under Sv57 with N=8.

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

Page-based profile	**Sv57 on RV64**
Effective Address	0xAAFFFFFF12345678 NVBITS[1010101] VBITS[0111111111111111111111111110001...000]
N	8
Mask	0x00FFFFFFFFFFFFFF NVBITS[0000000] VBITS[01111111111111111111111111111...111]
N+1st bit from the top	1
Transformed address	0xFFFFFFFF12345678 NVBITS[1111111] VBITS[1111111111111111111111111110001...000]

2.4. Allowed Values 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 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).

Systems may therefore choose to disallow certain values of N. The minimum set of values of N that must be supported may be defined by an ISA profile. The supported values of N may differ between privilege modes and between address translation modes but must otherwise be consistent.

When trying to enable pointer masking and setting a value of N in the corresponding CSR (Chapter 3), an implementation may choose a different value of N and reflect this in the updated register value. The behavior must be consistent: For a given pair of (address translation mode, privilege mode) there is an implementation-specific, fixed maximum number of supported mask bits (Nmax). All values of N ⇐ Nmax must be supported. If software tries to set N to a value larger than Nmax, the implementation will set N=Nmax. This follows the WARL CSR pattern (see Chapter 3 for details).

We expect that the initial profile will only mandate that systems support Nmax to be the number of NVBITS of the current privilege mode and address translation mode (which is equivalent to disabling the canonicity check). For Bare address translation, the profile will choose an Nmax such that the top Nmax+1 bits of the physical address space can be pinned to zero without substantially impacting implementations. Systems may choose to support a larger Nmax (which would encroach on the VBITS and may be more expensive to implement), but this would not initially be required by the profile.

2.5. Pointer Masking and Privilege Modes

Pointer masking is controlled separately for different privilege modes. 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.

Similarly, the pointer masking extension does not mandate which values of N are allowed beyond what is described in Section 2.4, “Allowed Values of N”. However, not all values of N make sense. For example, an operating system’s ABI may mandate that (e.g.,) user mode should never mask more bits than kernel mode.

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). To achieve this while using the full virtual address space of a given address translation mode, it may request N to be smaller than the maximum N available (Nmax). In addition, 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.

Values written to CSRs (e.g., stval) do not automatically have the ignore transformation applied before doing so. It is the responsibility of software to apply the ignore operation manually to such values before writing them into a CSR.

2.7. Instruction Fetches

Pointer masking can optionally be applied to instruction fetches. The profile mandates whether or not this feature needs to be supported, and software discovers the availability of this feature using WARL fields in the pointer masking configuration CSRs. 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.

Chapter 3. ISA Extension

The `zjpm` extension adds four new configuration CSRs: upm, spm, vspm and 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 S-mode and HS-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 on RV64 can be found in Figure 1a, 1b, 1c and 1d for M, (H)S, VS and (V)U-mode. On RV32, the layout is equivalent but the **bits** fields occupy 5 bits instead of 6.

A portion of each register is 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 once these portions get used by 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-1:9]	mpm[8:3]	mpm[2]	mpm[1]	mpm[0]
WPRI	mbits (WARL)	minst (WARL)	mself (WARL)	menable (WARL)

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

spm[XLEN-1:9]	spm[8:3]	spm[2]	spm[1]	spm[0]
WPRI	sbits (WARL)	sinst (WARL)	sself (WARL)	senable (WARL)

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

vspm[XLEN-1:9]	vspm[8:3]	vspm[2]	vspm[1]	vspm[0]
WPRI	vsbits (WARL)	vsinst (WARL)	vsself (WARL)	vsenable (WARL)

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

upm[XLEN-1:9]	upm[8:3]	upm[2]	upm[1]	upm[0]
WPRI	ubits (WARL)	uinst (WARL)	uself (WARL)	uenable (WARL)

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

3.1. mbits/sbits/vsbits/ubits

These bits indicate the number of masked bits, referred to as N throughout the specification. This is a WARL field following the description in Section 2.4, “Allowed Values of N”. Writing to this field will always result in the field being updated, but the value of N that is written may be smaller than what was requested. This does not affect any other field updates.

If pointer masking is disabled for this privilege mode (enable = 0), this field is ignored.

3.2. 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 may be unsupported by the implementation. If so, 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.

3.3. mself/sself/vsself/uself

This field controls 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.

For M-mode, this field is hard-wired to 1 (with WARL semantics).

Note that setting CSRs of lower privilege modes is always allowed and CSRs of higher privilege modes are not visible.

3.4. menable/senable/vsenable/uenable

The value of this field controls whether pointer masking is active for the corresponding privilege mode.

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. Field Invalidation

When changes to the current address translation mode and/or the privilege mode would cause the current privilege mode's bits value to become illegal, it is immediately reset to the largest legal value of N under the new address translation mode.

The intention of this definition is to provide well-defined behavior that is consistent with the rules of the privileged specification. Software should prevent this case from occurring as it is likely to result in unintended behavior.

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