RISC-V Pointer Masking

Version 0.6.1, 08/2023: 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 (`Smmjpm`, `Smnjpm`, `Ssnjpm`) 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 upper 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).

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

Listing 1. "Ignore" Transformation for virtual addresses, 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 applied to a physical address, the ignore transformation replaces the upper N bits with 0. This includes both the case of running in M-mode and running in other privilege modes with Bare address translation mode.

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

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

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 machanisms 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 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 (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[1111111111111111111111110001...000]
N	7
Mask	0x01FFFFFFFFFFFFFF NVBITS[0000000] VBITS[1111111111111111111111111111...111]
N+1st bit from the top	1
Transformed address	0xFFFFFFFF12345678 NVBITS[1111111] VBITS[1111111111111111111111110001...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 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 as well.

The specified extensions **do not** support N to be configurable and the extension does not mandate specific values of N. Instead, N is implementation-dependent (ISA profiles are expected to constrain possible values of N). As a result, compilers and runtime systems will need to rely on a discovery mechanisum (outside the scope of this document) to retrieve the value of N. Different values of N may be defined for each privilege mode (U/VU, S/HS, and M) and for different active address translation modes.

> 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 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 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. 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, 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.*.
- **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.

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

MPRV affects pointer masking as well, causing the pointer masking settings of the effective privilege mode to be applied. Pointer masking also applies to HLV, HLVX and HSV instructions.

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.

> Cache Management Operations (CMOs) 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. 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 the IOMMU or devices.

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 operation to the address and then issuing the constituent accesses based on the transformed address. 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.

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 N=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 writing the `stval` CSR in a trap handler).

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** to any address written into `stval`.

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 N.

Extensions:

- **Ssnjpm**: U/VU-mode pointer masking is available if and only if this extension is present. It is controlled at the supervisor level.
- **Smnjpm**: S/HS-mode pointer masking is available if and only if this extension is present. It is controlled at the machine level. In the presence of virtualization, this extension also adds VS-mode pointer masking, controlled at the hypervisor level.
- **Smjpm**: M-mode pointer masking is available if and only if this extension is present. It is controlled at the machine level.

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

Pointer masking only applies to RV64. On RV32, trying to enable pointer masking will result in an illegal WARL write and not update the pointer masking enable bit (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 even on 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.

Chapter 3. ISA Extensions

This section describes the pointer masking extensions `Smmjpm`, `Smnjpm` and `Ssnjpm`. All of these extensions are privileged ISA extensions and do not add any new CSRs.

> 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.

3.1. Ssnjpm

`Ssnjpm` adds a new 1-bit WARL field (`upmen`) to `senvcfg`. Setting `upmen` enables (1) or disables (0) pointer masking for U/VU mode.

3.2. Smnjpm

`Smnjpm` adds a new 1-bit WARL field (`spmen`) to `menvcfg`. Setting `spmen` enables (1) or disables (0) pointer masking for S/HS mode.

In systems that support virtualization, `Smnjpm` also adds a new 1-bit WARL field (`hpmen`) to `henvcfg` that have the equivalent effect for VS-mode pointer masking.

3.3. Smmjpm

`Smmjpm` adds a new 1-bit WARL field (`mpmen`) to `mseccfg`. Setting `mpmen` enables (1) or disables (0) pointer masking for M mode.

> There is an open question whether an SFENCE.VMA would be required after enabling/disabling pointer masking. The decision is TBD.

3.4. 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 privilege mode and address translation mode. It is implementation-dependent, but the ISA profiles are expected to constrain the value of N (e.g., by setting a required minimum).

Implementations may choose to not make pointer masking available under certain address translation modes. For example, an implementation that supports Sv57 may choose to not support pointer masking within this address mode. Trying to enable pointer masking in an unsupported scenario represents an illegal write to the corresponding pointer masking enable bit and follows WARL semantics.

> Implementations may feel tempted to set N depending on the current addressing modes, and profiles may set a valid range of N that allows such implementations. For example, N could be set to 16 for Sv48 and to 25 for Sv39. However, having a

single value of N (e.g., setting N 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. Note that the number of Masked Bits supported for each address translation mode may change in the future, determined by the profile (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