Normative Rules

3.1. Machine-Level ISA, Version 1.13

This chapter describes the machine-level operations available in machine-mode (M-mode), which is the highest privilege mode in a RISC-V hart. M-mode is used for low-level access to a hardware platform and is the first mode entered at reset. M-mode can also be used to implement features that are too difficult or expensive to implement in hardware directly. The RISC-V machine-level ISA contains a common core that is extended depending on which other privilege levels are supported and other details of the hardware implementation.

3.1.1. Machine-Level CSRs

In addition to the machine-level CSRs described in this section, M-mode code can access all CSRs at lower privilege levels.

3.1.1.1. Machine ISA (`misa`) Register

The `misa` CSR is a **WARL** read-write register reporting the ISA supported by the hart. This register must be readable in any implementation, but a value of zero can be returned to indicate the `misa` register has not been implemented, requiring that CPU capabilities be determined through a separate non-standard mechanism.

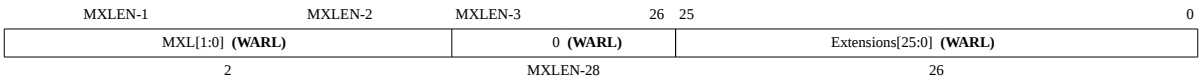


Figure 1. Machine ISA register (misa)

The MXL (Machine XLEN) field encodes the native base integer ISA width as shown in Encoding of MXL field in `misa`. The MXL field is read-only. If `misa` is nonzero, the MXL field indicates the effective XLEN in M-mode, a constant termed *MXLEN*. XLEN is never greater than MXLEN, but XLEN might be smaller than MXLEN in less-privileged modes.

Table 1. Encoding of MXL field in `misa`

MXL	XLEN

MXL	XLEN
1	32
2	64
3	*Reserved*

The `misa` CSR is MXLEN bits wide.

NOTE

The base width can be quickly ascertained using branches on the sign of the returned `misa` value, and possibly a shift left by one and a second branch on the sign. These checks can be written in assembly code without knowing the register width (MXLEN) of the hart. The base width is given by $MXLEN=2^{MXL+4}$.

The base width can also be found if `misa` is zero, by placing the immediate 2 in a register, then shifting the register left by 31 bits. If zero, the hart is RV32, else it is RV64.

The Extensions field encodes the presence of the standard extensions, with a single bit per letter of the alphabet (bit 0 encodes presence of extension "A" , bit 1 encodes presence of extension "B", through to bit 25 which encodes "Z"). The "I" bit will be set for the RV32I and RV64I base ISAs, and the "E" bit will be set for RV32E and RV64E. The Extensions field is a **WARL** field that can contain writable bits where the implementation allows the supported ISA to be modified. At reset, the Extensions field shall contain the maximal set of supported extensions, and "I" shall be selected over "E" if both are available.

When a standard extension is disabled by clearing its bit in `misa`, the instructions and CSRs defined or modified by the extension revert to their defined or reserved behaviors as if the extension is not implemented.

NOTE

For a given RISC-V execution environment, an instruction, extension, or other feature of the RISC-V ISA is ordinarily judged to be *implemented* or not by the observable execution behavior in that environment. For example, the F extension is said to be implemented for an execution environment if and only if the instructions that the RISC-V Unprivileged ISA defines for F execute as specified.

With this definition of *implemented*, disabling an extension by clearing its bit in `misa` results in the extension being considered *not implemented* in M-mode. For example, setting `misa`.F=0 results in the F extension being not implemented for M-mode, because the F extension's instructions will not act as the Unprivileged ISA requires but may instead raise an illegal-instruction exception.

Defining the term *implemented* based strictly on the observable behavior might conflict with other common understandings of the same word. In particular, although common usage may allow for the combination "implemented but disabled," in this document it is considered a contradiction of terms, because *disabled* implies execution will not behave as required for the feature to be considered *implemented*. In the same vein, "implemented and enabled" is redundant here; "implemented" suffices.

All bits that are reserved for future use must return zero when read.

Table 2. Encoding of Extensions field in `misa`.

Bit	Character	Description
0	A	Atomic extension
1	B	B extension
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E/64E base ISA
5	F	Single-precision floating-point extension
6	G	*Reserved*
7	H	Hypervisor extension
8	I	RV32I/64I base ISA
9	J	*Reserved*
10	K	*Reserved*
11	L	*Reserved*
12	M	Integer Multiply/Divide extension
13	N	*Tentatively reserved for User-Level Interrupts extension*
14	O	*Reserved*
15	P	*Tentatively reserved for Packed-SIMD extension*
16	Q	Quad-precision floating-point extension
17	R	*Reserved*
18	S	Supervisor mode implemented
19	T	*Reserved*
20	U	User mode implemented
21	V	Vector extension
22	W	*Reserved*
23	X	Non-standard extensions present
24	Y	*Reserved*
25	Z	*Reserved*

The "X" bit will be set if there are any non-standard extensions.

When the "B" bit is 1, the implementation supports the instructions provided by the Zba, Zbb, and Zbs extensions. When the "B" bit is 0, it indicates that the implementation might not support one or more of the Zba, Zbb, or Zbs extensions.

When the "M" bit is 1, the implementation supports all multiply and division instructions defined by the M extension. When the "M" bit is 0, it indicates that the implementation might not support those instructions. However if the Zmmul extension is supported then the multiply instructions it specifies are supported irrespective of the value of the "M" bit.

When the "S" bit is 1, the implementation supports supervisor mode. When the "S" bit is 0, the implementation might not support supervisor mode.

When the "U" bit is 1, the implementation supports user mode. When the "U" bit is 0, the implementation might not support user mode.

NOTE

The `misa` CSR exposes a rudimentary catalog of CPU features to machine-mode code. More extensive information can be obtained in machine mode by probing other machine registers, and examining other ROM storage in the system as part of the boot process.

We require that lower privilege levels execute environment calls instead of reading CPU registers to determine features available at each privilege level. This enables virtualization layers to alter the ISA observed at any level, and supports a much richer command interface without burdening hardware designs.

The "E" bit is read-only. Unless `misa` is all read-only zero, the "E" bit always reads as the complement of the "I" bit. If an execution environment supports both RV32E and RV32I, software can select RV32E by clearing the "I" bit.

If an ISA feature *x* depends on an ISA feature *y*, then attempting to enable feature *x* but disable feature *y* results in both features being disabled. For example, setting "F"=0 and "D"=1 results in both "F" and "D" being cleared. Similarly, setting "U"=0 and "S"=1" results in both "U" and "S" being cleared.

An implementation may impose additional constraints on the collective setting of two or more `misa` fields, in which case they function collectively as a single **WARL** field. An attempt to write an unsupported combination causes those bits to be set to some supported combination.

Writing `misa` may increase IALIGN, e.g., by disabling the "C" extension. If an instruction that would write `misa` increases IALIGN, and the subsequent instruction's address is not IALIGN-bit aligned, the write to `misa` is suppressed, leaving `misa` unchanged.

When software enables an extension that was previously disabled, then all state uniquely associated with that extension is UNSPECIFIED, unless otherwise specified by that extension.

NOTE

Although one of the bits 25—0 in `misa` being set to 1 implies that the corresponding feature is implemented, the inverse is not necessarily true: one of these bits being clear does not necessarily imply that the corresponding feature is not implemented. This follows from the fact that, when a feature is not implemented, the corresponding opcodes and CSRs become reserved, not necessarily illegal.

3.1.1.2. Machine Vendor ID (`mvendorid`) Register

The `mvendorid` CSR is a 32-bit read-only register providing the JEDEC manufacturer ID of the provider of the core. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented or that this is a non-commercial implementation.

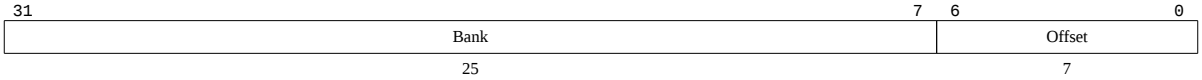


Figure 2. Vendor ID register (`mvendorid`)

JEDEC manufacturer IDs are ordinarily encoded as a sequence of one-byte continuation codes `0x7f`, terminated by a one-byte ID not equal to `0x7f`, with an odd parity bit in the most-significant bit of each byte. `mvendorid` encodes the number of one-byte continuation codes in the Bank field, and encodes the final byte in the Offset field, discarding the parity bit. For example, the JEDEC manufacturer ID `0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x8a` (twelve continuation codes followed by `0x8a`) would be encoded in the `mvendorid` CSR as `0x60a`.

NOTE

In JEDEC's parlance, the bank number is one greater than the number of continuation codes; hence, the `mvendorid` Bank field encodes a value that is one less than the JEDEC bank number.

Previously the vendor ID was to be a number allocated by RISC-V International, but this duplicates the work of JEDEC in maintaining a manufacturer ID standard. At time of writing, registering a manufacturer ID with JEDEC has a one-time cost of $500.

3.1.1.3. Machine Architecture ID (`marchid`) Register

The `marchid` CSR is an MXLEN-bit read-only register encoding the base microarchitecture of the hart. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented. The combination of `mvendorid` and `marchid` should uniquely identify the type of hart microarchitecture that is implemented.

Figure 3. Machine Architecture ID (`marchid`) register

Open-source project architecture IDs are allocated globally by RISC-V International, and have non-zero architecture IDs with a zero most-significant-bit (MSB). Commercial architecture IDs are allocated by each commercial vendor independently, but must have the MSB set and cannot contain zero in the remaining MXLEN-1 bits.

NOTE

NOTE

The intent is for the architecture ID to represent the microarchitecture associated with the project around which development occurs rather than a particular organization. Commercial fabrications of open-source designs should (and might be required by the license to) retain the original architecture ID. This will aid in reducing fragmentation and tool support costs, as well as provide attribution. Open-source architecture IDs are administered by RISC-V International and should only be allocated to released, functioning open-source projects. Commercial architecture IDs can be managed independently by any registered vendor but are required to have IDs disjoint from the open-source architecture IDs (MSB set) to prevent collisions if a vendor wishes to use both closed-source and open-source microarchitectures.

The convention adopted within the following Implementation field can be used to segregate branches of the same architecture design, including by organization. The `misa` register also helps distinguish different variants of a design.

3.1.1.4. Machine Implementation ID (`mimpid`) Register

The `mimpid` CSR provides a unique encoding of the version of the processor implementation. This register must be readable in any implementation, but a value of 0 can be returned to indicate that the field is not implemented. The Implementation value should reflect the design of the RISC-V processor itself and not any surrounding system.

Figure 4. Machine Implementation ID (`mimpid`) register

NOTE

The format of this field is left to the provider of the architecture source code, but will often be printed by standard tools as a hexadecimal string without any leading or trailing zeros, so the Implementation value can be left-justified (i.e., filled in from most-significant nibble down) with subfields aligned on nibble boundaries to ease human readability.

3.1.1.5. Hart ID (`mhartid`) Register

The `mhartid` CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but one hart must have a hart ID of zero. Hart IDs must be unique within the execution environment.

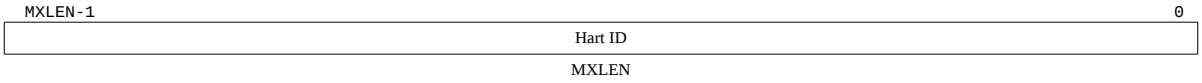


Figure 5. Hart ID (`mhartid`) register

NOTE

NOTE

In certain cases, we must ensure exactly one hart runs some code (e.g., at reset), and so require one hart to have a known hart ID of zero.

For efficiency, system implementers should aim to reduce the magnitude of the largest hart ID used in a system.

3.1.1.6. Machine Status (`mstatus` and `mstatush`) Registers

The `mstatus` register is an MXLEN-bit read/write register formatted as shown in Machine-mode status (`mstatus`) register for RV32 for RV32 and Machine-mode status (`mstatus`) register for RV64 for RV64. The `mstatus` register keeps track of and controls the hart's current operating state. A restricted view of `mstatus` appears as the `sstatus` register in the S-level ISA.

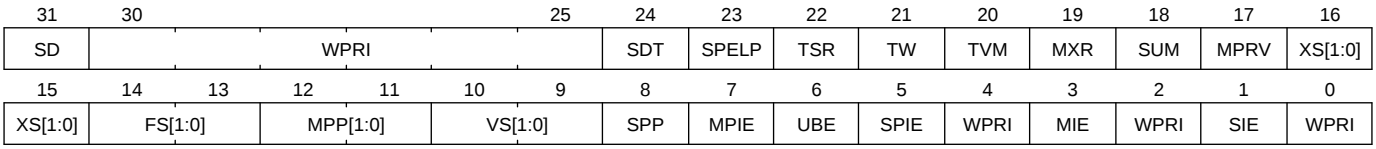


Figure 6. Machine-mode status (`mstatus`) register for RV32

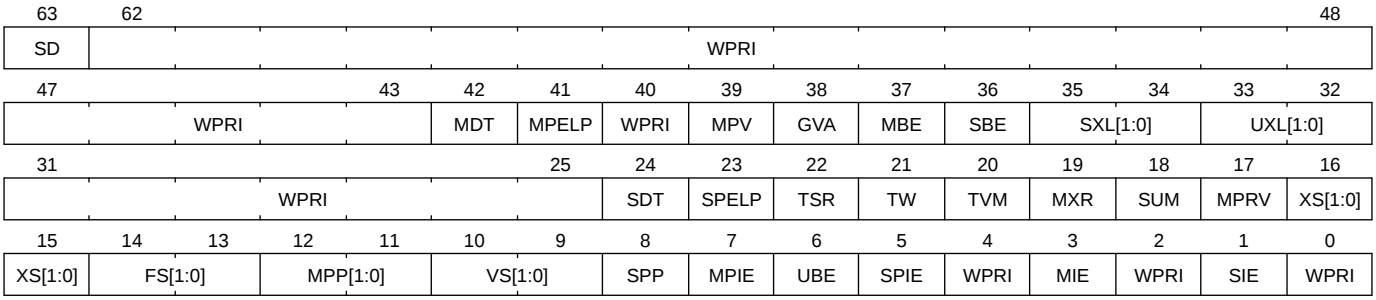


Figure 7. Machine-mode status (`mstatus`) register for RV64

For RV32 only, `mstatush` is a 32-bit read/write register formatted as shown in Additional machine-mode status (`mstatush`) register for RV32.. Bits 30:4 of `mstatush` generally contain the same fields found in bits 62:36 of `mstatus` for RV64. Fields SD, SXL, and UXL do not exist in `mstatush`.

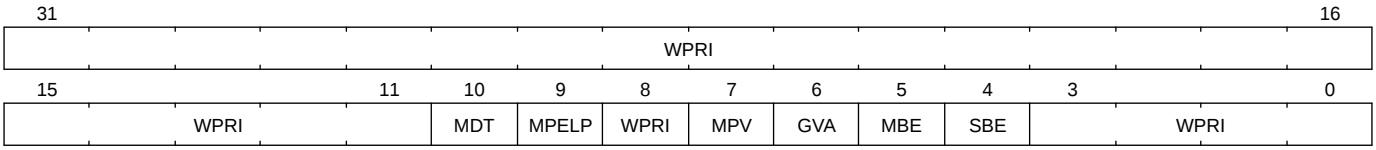


Figure 8. Additional machine-mode status (`mstatush`) register for RV32.

3.1.1.6.1. Privilege and Global Interrupt-Enable Stack in `mstatus` register

Global interrupt-enable bits, MIE and SIE, are provided for M-mode and S-mode respectively. These bits are primarily used to guarantee atomicity with respect to interrupt handlers in the current privilege mode.

NOTE

NOTE

The global *x*IE bits are located in the low-order bits of `mstatus`, allowing them to be atomically set or cleared with a single CSR instruction.

When a hart is executing in privilege mode *x*, interrupts are globally enabled when *x*IE=1 and globally disabled when *x*IE=0. Interrupts for lower-privilege modes, *w*<*x*, are always globally disabled regardless of the setting of any global *w*IE bit for the lower-privilege mode. Interrupts for higher-privilege modes, *y*>*x*, are always globally enabled regardless of the setting of the global *y*IE bit for the higher-privilege mode. Higher-privilege-level code can use separate per-interrupt enable bits to disable selected higher-privilege-mode interrupts before ceding control to a lower-privilege mode. If supervisor mode is not implemented, then SIE and SPIE are read-only 0.

NOTE

A higher-privilege mode *y* could disable all of its interrupts before ceding control to a lower-privilege mode but this would be unusual as it would leave only a synchronous trap, non-maskable interrupt, or reset as means to regain control of the hart.

To support nested traps, each privilege mode *x* that can respond to interrupts has a two-level stack of interrupt-enable bits and privilege modes. *x*PIE holds the value of the interrupt-enable bit active prior to the trap, and *x*PP holds the previous privilege mode. The *x*PP fields can only hold privilege modes up to *x*, so MPP is two bits wide and SPP is one bit wide. When a trap is taken from privilege mode *y* into privilege mode *x*, *x*PIE is set to the value of *x*IE; *x*IE is set to 0; and *x*PP is set to *y*.

NOTE

For lower privilege modes, any trap (synchronous or asynchronous) is usually taken at a higher privilege mode with interrupts disabled upon entry. The higher-level trap handler will either service the trap and return using the stacked information, or, if not returning immediately to the interrupted context, will save the privilege stack before re-enabling interrupts, so only one entry per stack is required.

An MRET or SRET instruction is used to return from a trap in M-mode or S-mode respectively. When executing an *x*RET instruction, supposing *x*PP holds the value *y*, *x*IE is set to *x*PIE; the privilege mode is changed to *y*; *x*PIE is set to 1; and *x*PP is set to the least-privileged supported mode (U if U-mode is implemented, else M). If *y*≠M, *x*RET also sets MPRV=0.

NOTE

Setting *x*PP to the least-privileged supported mode on an *x*RET helps identify software bugs in the management of the two-level privilege-mode stack.

NOTE

NOTE

Trap handlers must be designed to neither enable interrupts nor cause exceptions during the phase of handling where the trap handler preserves the critical state information required to handle and resume from the trap. An exception or interrupt in this critical phase of trap handling may lead to a trap that can overwrite such critical state. This could result in the loss of data needed to recover from the initial trap. Further, if an exception occurs in the code path needed to handle traps, then such a situation may lead to an infinite loop of traps. To prevent this, trap handlers must be meticulously designed to identify and safely manage exceptions within their operational flow.

*x*PP fields are **WARL** fields that can hold only privilege mode *x* and any implemented privilege mode lower than *x*. If privilege mode *x* is not implemented, then *x*PP must be read-only 0.

NOTE

M-mode software can determine whether a privilege mode is implemented by writing that mode to MPP then reading it back.

If the machine provides only U and M modes, then only a single hardware storage bit is required to represent either 00 or 11 in MPP.

3.1.1.6.2. Double Trap Control in `mstatus` Register

A double trap typically arises during a sensitive phase in trap handling operations — when an exception or interrupt occurs while the trap handler (the component responsible for managing these events) is in a non-reentrant state. This non-reentrancy usually occurs in the early phase of trap handling, wherein the trap handler has not yet preserved the necessary state to handle and resume from the trap. The occurrence of a trap during this phase can lead to an overwrite of critical state information, resulting in the loss of data needed to recover from the initial trap. The trap that caused this critical error condition is henceforth called the *unexpected trap*. Trap handlers are designed to neither enable interrupts nor cause exceptions during this phase of handling. However, managing Hardware-Error exceptions, which may occur unpredictably, presents significant challenges in trap handler implementation due to the potential risk of a double trap.

The M-mode-disable-trap (`MDT`) bit is a WARL field introduced by the Smdbltrp extension. Upon reset, the `MDT` field is set to 1. When the `MDT` bit is set to 1 by an explicit CSR write, the `MIE` (Machine Interrupt Enable) bit is cleared to 0. For RV64, this clearing occurs regardless of the value written, if any, to the `MIE` bit by the same write. The `MIE` bit can only be set to 1 by an explicit CSR write if the `MDT` bit is already 0 or, for RV64, is being set to 0 by the same write (For RV32, the `MDT` bit is in `mstatush` and the `MIE` bit in `mstatus` register).

When a trap is to be taken into M-mode, if the `MDT` bit is currently 0, it is then set to 1, and the trap is delivered as expected. However, if `MDT` is already set to 1, then this is an *unexpected trap*. When the Smrnmi extension is implemented, a trap caused by an RNMI is not considered an *unexpected trap*

irrespective of the state of the `MDT` bit. A trap caused by an RNMI does not set the `MDT` bit. However, a trap that occurs when executing in M-mode with `mnstatus.NMIE` set to 0 is an *unexpected trap*.

In the event of a *unexpected trap*, the handling is as follows:

- When the Smrnmi extension is implemented and `mnstatus.NMIE` is 1, the hart traps to the RNMI handler. To deliver this trap, the `mnepc` and `mncause` registers are written with the values that the *unexpected trap* would have written to the `mepc` and `mcause` registers respectively. The privilege mode information fields in the `mnstatus` register are written to indicate M-mode and its `NMIE` field is set to 0.

NOTE

The consequence of this specification is that on occurrence of double trap the RNMI handler is not provided with information that a trap reports in the `mtval` and the `mtval2` registers. This information, if needed, can be obtained by the RNMI handler by decoding the instruction at the address in `mnepc` and examining its source register contents.

- When the Smrnmi extension is not implemented, or if the Smrnmi extension is implemented and `mnstatus.NMIE` is 0, the hart enters a critical-error state without updating any architectural state, including the `pc`. This state involves ceasing execution, disabling all interrupts (including NMIs), and asserting a `critical-error` signal to the platform. Whether performance counters and timers are updated in the critical-error state is UNSPECIFIED.

NOTE

The actions performed by the platform when a hart asserts a `critical-error` signal are platform-specific. The range of possible actions include restarting the affected hart or restarting the entire platform, among others.

The `MRET` and `SRET` instructions, when executed in M-mode, set the `MDT` bit to 0. If the new privilege mode is U, VS, or VU, then `sstatus.SDT` is also set to 0. Additionally, if it is VU, then `vsstatus.SDT` is also set to 0.

The `MNRET` instruction, provided by the Smrnmi extension, sets the `MDT` bit to 0 if the new privilege mode is not M. If it is U, VS, or VU, then `sstatus.SDT` is also set to 0. Additionally, if it is VU, then `vsstatus.SDT` is also set to 0.

3.1.1.6.3. Base ISA Control in `mstatus` Register

For RV64 harts, the SXL and UXL fields are **WARL** fields that control the value of XLEN for S-mode and U-mode, respectively. The encoding of these fields is the same as the MXL field of `misa`, shown in Encoding of MXL field in `misa`. The effective XLEN in S-mode and U-mode are termed *SXLEN* and *UXLEN*, respectively.

When MXLEN=32, the SXL and UXL fields do not exist, and SXLEN=32 and UXLEN=32.

When MXLEN=64, if S-mode is not supported, then SXL is read-only zero. Otherwise, it is a **WARL** field that encodes the current value of SXLEN. In particular, an implementation may make SXL be a read-only field whose value always ensures that SXLEN=MXLEN.

When MXLEN=64, if U-mode is not supported, then UXL is read-only zero. Otherwise, it is a **WARL** field that encodes the current value of UXLEN. In particular, an implementation may make UXL be a read-only field whose value always ensures that UXLEN=MXLEN or UXLEN=SXLEN.

If S-mode is implemented, the set of legal values that the UXL field may assume excludes those that would cause UXLEN to be greater than SXLEN.

Whenever XLEN in any mode is set to a value less than the widest supported XLEN, all operations must ignore source operand register bits above the configured XLEN, and must sign-extend results to fill the entire widest supported XLEN in the destination register. Similarly, `pc` bits above XLEN are ignored, and when the `pc` is written, it is sign-extended to fill the widest supported XLEN.

NOTE

We require that operations always fill the entire underlying hardware registers with defined values to avoid implementation-defined behavior.

To reduce hardware complexity, the architecture imposes no checks that lower-privilege modes have XLEN settings less than or equal to the next-higher privilege mode. In practice, such settings would almost always be a software bug, but machine operation is well-defined even in this case.

Some HINT instructions are encoded as integer computational instructions that overwrite their destination register with its current value, e.g., `c.addi x8, 0`. When such a HINT is executed with XLEN < MXLEN and bits MXLEN..XLEN of the destination register not all equal to bit XLEN-1, it is implementation-defined whether bits MXLEN..XLEN of the destination register are unchanged or are overwritten with copies of bit XLEN-1.

NOTE

This definition allows implementations to elide register write-back for some HINTs, while allowing them to execute other HINTs in the same manner as other integer computational instructions. The implementation choice is observable only by privilege modes with an XLEN setting greater than the current XLEN; it is invisible to the current privilege mode.

3.1.1.6.4. Memory Privilege in `mstatus` Register

The MPRV (Modify PRiVilege) bit modifies the *effective privilege mode*, i.e., the privilege level at which explicit memory accesses execute. When MPRV=0, explicit memory accesses behave as normal, using the translation and protection mechanisms of the current privilege mode. When MPRV=1, load and

store memory addresses are translated and protected, and endianness is applied, as though the current privilege mode were set to MPP. Instruction address-translation and protection are unaffected by the setting of MPRV. MPRV is read-only 0 if U-mode is not supported.

An MRET or SRET instruction that changes the privilege mode to a mode less privileged than M also sets MPRV=0.

The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1 in Sv32 page table entry) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect. MXR is read-only 0 if S-mode is not supported.

NOTE

The MPRV and MXR mechanisms were conceived to improve the efficiency of M-mode routines that emulate missing hardware features, e.g., misaligned loads and stores. MPRV obviates the need to perform address translation in software. MXR allows instruction words to be loaded from pages marked execute-only.

The current privilege mode and the privilege mode specified by MPP might have different XLEN settings. When MPRV=1, load and store memory addresses are treated as though the current XLEN were set to MPP's XLEN, following the rules in 3.1.1.6.3. Base ISA Control in `mstatus` Register.

The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode (U=1 in Sv32 page table entry) will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect. Note that, while SUM is ordinarily ignored when not executing in S-mode, it *is* in effect when MPRV=1 and MPP=S. SUM is read-only 0 if S-mode is not supported or if `satp`.MODE is read-only 0.

The MXR and SUM mechanisms only affect the interpretation of permissions encoded in page-table entries. In particular, they have no impact on whether access-fault exceptions are raised due to PMAs or PMP.

3.1.1.6.5. Endianness Control in `mstatus` and `mstatush` Registers

The MBE, SBE, and UBE bits in `mstatus` and `mstatush` are **WARL** fields that control the endianness of memory accesses other than instruction fetches. Instruction fetches are always little-endian.

MBE controls whether non-instruction-fetch memory accesses made from M-mode (assuming `mstatus`.MPRV=0) are little-endian (MBE=0) or big-endian (MBE=1).

If S-mode is not supported, SBE is read-only 0. Otherwise, SBE controls whether explicit load and store memory accesses made from S-mode are little-endian (SBE=0) or big-endian (SBE=1).

If U-mode is not supported, UBE is read-only 0. Otherwise, UBE controls whether explicit load and store memory accesses made from U-mode are little-endian (UBE=0) or big-endian (UBE=1).

For *implicit* accesses to supervisor-level memory management data structures, such as page tables, endianness is always controlled by SBE. Since changing SBE alters the implementation's interpretation of these data structures, if any such data structures remain in use across a change to SBE, M-mode software must follow such a change to SBE by executing an SFENCE.VMA instruction with *rs1*=`x0` and *rs2*=`x0`.

NOTE

Only in contrived scenarios will a given memory-management data structure be interpreted as both little-endian and big-endian. In practice, SBE will only be changed at runtime on world switches, in which case neither the old nor new memory-management data structure will be reinterpreted in a different endianness. In this case, no additional SFENCE.VMA is necessary, beyond what would ordinarily be required for a world switch.

If S-mode is supported, an implementation may make SBE be a read-only copy of MBE. If U-mode is supported, an implementation may make UBE be a read-only copy of either MBE or SBE.

NOTE

NOTE

An implementation supports only little-endian memory accesses if fields MBE, SBE, and UBE are all read-only 0. An implementation supports only big-endian memory accesses (aside from instruction fetches) if MBE is read-only 1 and SBE and UBE are each read-only 1 when S-mode and U-mode are supported.

Volume I defines a hart's address space as a circular sequence of 2^{XLEN} bytes at consecutive addresses. The correspondence between addresses and byte locations is fixed and not affected by any endianness mode. Rather, the applicable endianness mode determines the order of mapping between memory bytes and a multibyte quantity (halfword, word, etc.).

Standard RISC-V ABIs are expected to be purely little-endian-only or big-endian-only, with no accommodation for mixing endianness. Nevertheless, endianness control has been defined so as to permit, for instance, an OS of one endianness to execute user-mode programs of the opposite endianness. Consideration has been given also to the possibility of non-standard usages whereby software flips the endianness of memory accesses as needed.

RISC-V instructions are uniformly little-endian to decouple instruction encoding from the current endianness settings, for the benefit of both hardware and software. Otherwise, for instance, a RISC-V assembler or disassembler would always need to know the intended active endianness, despite that the endianness mode might change dynamically during execution. In contrast, by giving instructions a fixed endianness, it is sometimes possible for carefully written software to be endianness-agnostic even in binary form, much like position-independent code.

The choice to have instructions be only little-endian does have consequences, however, for RISC-V software that encodes or decodes machine instructions. In big-endian mode, such software must account for the fact that explicit loads and stores have endianness opposite that of instructions, for example by swapping byte order after loads and before stores.

3.1.1.6.6. Virtualization Support in `mstatus` Register

The TVM (Trap Virtual Memory) bit is a **WARL** field that supports intercepting supervisor virtual-memory management operations. When TVM=1, attempts to read or write the `satp` CSR or execute an SFENCE.VMA or SINVAL.VMA instruction while executing in S-mode will raise an illegal-instruction exception. When TVM=0, these operations are permitted in S-mode. TVM is read-only 0 when S-mode is not supported.

NOTE

The TVM mechanism improves virtualization efficiency by permitting guest operating systems to execute in S-mode, rather than classically virtualizing them in U-mode. This approach obviates the need to trap accesses to most S-mode CSRs.

Trapping `satp` accesses and the SFENCE.VMA and SINVAL.VMA instructions provides the hooks necessary to lazily populate shadow page tables.

The TW (Timeout Wait) bit is a **WARL** field that supports intercepting the WFI instruction (see 3.1.3.3. Wait for Interrupt). When TW=0, the WFI instruction may execute in modes less privileged than M when not prevented for some other reason. When TW=1, then if WFI is executed in any less-privileged mode, and it does not complete within an implementation-specific, bounded time limit, the WFI instruction causes an illegal-instruction exception. An implementation may have WFI always raise an illegal-instruction exception in modes less privileged than M when TW=1, even if there are pending globally-disabled interrupts when the instruction is executed. TW is read-only 0 when there are no modes less privileged than M.

NOTE

Trapping the WFI instruction can trigger a world switch to another guest OS, rather than wastefully idling in the current guest.

When S-mode is implemented, then executing WFI in U-mode causes an illegal-instruction exception, regardless of the value of the TW bit, unless the instruction completes within an implementation-specific, bounded time limit.

The TSR (Trap SRET) bit is a **WARL** field that supports intercepting the supervisor exception return instruction, SRET. When TSR=1, attempts to execute SRET while executing in S-mode will raise an illegal-instruction exception. When TSR=0, this operation is permitted in S-mode. TSR is read-only 0 when S-mode is not supported.

NOTE

Trapping SRET is necessary to emulate the hypervisor extension (see "H" Extension for Hypervisor Support) on implementations that do not provide it.

3.1.1.6.7. Extension Context Status in `mstatus` Register

Supporting substantial extensions is one of the primary goals of RISC-V, and hence we define a standard interface to allow unchanged privileged-mode code, particularly a supervisor-level OS, to support arbitrary user-mode state extensions.

NOTE

To date, the V extension is the only standard extension that defines additional state beyond the floating-point CSR and data registers.

The FS[1:0] and VS[1:0] **WARL** fields and the XS[1:0] read-only field are used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit and any other user-mode extensions respectively. The FS field encodes the status of the floating-point unit state, including the floating-point registers `f0`–`f31` and the CSRs `fcsr`, `frm`, and `fflags`. The VS field encodes the status of the vector extension state, including the vector registers `v0`–`v31` and the CSRs `vcsr`, `vxrm`, `vxsat`, `vstart`, `vl`, `vtype`, and `vlenb`. The XS field encodes the status of

additional user-mode extensions and associated state. These fields can be checked by a context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.

NOTE

The design anticipates that most context switches will not need to save/restore state in either or both of the floating-point unit or other extensions, so provides a fast check via the SD bit.

The FS, VS, and XS fields use the same status encoding as shown in Encoding of FS[1:0], VS[1:0], and XS[1:0] status fields, with the four possible status values being Off, Initial, Clean, and Dirty.

Table 3. Encoding of FS[1:0], VS[1:0], and XS[1:0] status fields

Status	FS and VS Meaning	XS Meaning
0	Off	All off
1	Initial	None dirty or clean, some on
2	Clean	None dirty, some clean
3	Dirty	Some dirty

If the F extension is implemented, the FS field shall not be read-only zero.

If neither the F extension nor S-mode is implemented, then FS is read-only zero. If S-mode is implemented but the F extension is not, FS may optionally be read-only zero.

NOTE

Implementations with S-mode but without the F extension are permitted, but not required, to make the FS field be read-only zero. Some such implementations will choose *not* to have the FS field be read-only zero, so as to enable emulation of the F extension for both S-mode and U-mode via invisible traps into M-mode.

If the `v` registers are implemented, the VS field shall not be read-only zero.

If neither the `v` registers nor S-mode is implemented, then VS is read-only zero. If S-mode is implemented but the `v` registers are not, VS may optionally be read-only zero.

In harts without additional user extensions requiring new state, the XS field is read-only zero. Every additional extension with state provides a CSR field that encodes the equivalent of the XS states. The XS field represents a summary of all extensions' status as shown in Encoding of FS[1:0], VS[1:0], and XS[1:0] status fields.

NOTE

The XS field effectively reports the maximum status value across all user-extension status fields, though individual extensions can use a different encoding than XS.

The SD bit is a read-only bit that summarizes whether either the FS, VS, or XS fields signal the presence of some dirty state that will require saving extended user context to memory. If FS, XS, and VS are all read-only zero, then SD is also always zero.

When an extension's status is set to Off, any instruction that attempts to read or write the corresponding state will cause an illegal-instruction exception. When the status is Initial, the corresponding state should have an initial constant value. When the status is Clean, the corresponding state is potentially different from the initial value, but matches the last value stored on a context swap. When the status is Dirty, the corresponding state has potentially been modified since the last context save.

During a context save, the responsible privileged code need only write out the corresponding state if its status is Dirty, and can then reset the extension's status to Clean. During a context restore, the context need only be loaded from memory if the status is Clean (it should never be Dirty at restore). If the status is Initial, the context must be set to an initial constant value on context restore to avoid a security hole, but this can be done without accessing memory. For example, the floating-point registers can all be initialized to the immediate value 0.

The FS and XS fields are read by the privileged code before saving the context. The FS field is set directly by privileged code when resuming a user context, while the XS field is set indirectly by writing to the status register of the individual extensions. The status fields will also be updated during execution of instructions, regardless of privilege mode.

Extensions to the user-mode ISA often include additional user-mode state, and this state can be considerably larger than the base integer registers. The extensions might only be used for some applications, or might only be needed for short phases within a single application. To improve performance, the user-mode extension can define additional instructions to allow user-mode software to return the unit to an initial state or even to turn off the unit.

For example, a coprocessor might require to be configured before use and can be "unconfigured" after use. The unconfigured state would be represented as the Initial state for context save. If the same application remains running between the unconfigure and the next configure (which would set status to Dirty), there is no need to actually reinitialize the state at the unconfigure instruction, as all state is local to the user process, i.e., the Initial state may only cause the coprocessor state to be initialized to a constant value at context restore, not at every unconfigure.

Executing a user-mode instruction to disable a unit and place it into the Off state will cause an illegal-instruction exception to be raised if any subsequent instruction tries to use the unit before it is turned back on. A user-mode instruction to turn a unit on must also ensure the unit's state is properly initialized, as the unit might have been used by another context meantime.

Changing the setting of FS has no effect on the contents of the floating-point register state. In particular, setting FS=Off does not destroy the state, nor does setting FS=Initial clear the contents. Similarly, the

setting of VS has no effect on the contents of the vector register state. Other extensions, however, might not preserve state when set to Off.

Implementations may choose to track the dirtiness of the floating-point register state imprecisely by reporting the state to be dirty even when it has not been modified. On some implementations, some instructions that do not mutate the floating-point state may cause the state to transition from Initial or Clean to Dirty. On other implementations, dirtiness might not be tracked at all, in which case the valid FS states are Off and Dirty, and an attempt to set FS to Initial or Clean causes it to be set to Dirty.

NOTE

This definition of FS does not disallow setting FS to Dirty as a result of errant speculation. Some platforms may choose to disallow speculatively writing FS to close a potential side channel.

If an instruction explicitly or implicitly writes a floating-point register or the `fcsr` but does not alter its contents, and FS=Initial or FS=Clean, it is implementation-defined whether FS transitions to Dirty.

Implementations may choose to track the dirtiness of the vector register state in an analogous imprecise fashion, including possibly setting VS to Dirty when software attempts to set VS=Initial or VS=Clean. When VS=Initial or VS=Clean, it is implementation-defined whether an instruction that writes a vector register or vector CSR but does not alter its contents causes VS to transition to Dirty.

FS, VS, and XS state transitions. shows all the possible state transitions for the FS, VS, or XS status bits. Note that the standard floating-point and vector extensions do not support user-mode unconfigure or disable/enable instructions.

Table 4. FS, VS, and XS state transitions.

Current State Action	Off	Initial	Clean
		At context save in privileged code	
Save state? Next state	No Off	No Initial	No Clean
		At context restore in privileged code	
Restore state? Next state	No Off	Yes, to initial Initial	Yes, from n Clean
		Execute instruction to read state	
Action? Next state	Exception Off	Execute Initial	Execute Clean
		Execute instruction that possibly modifies state, including c	
Action? Next state	Exception Off	Execute Dirty	Execute Dirty
		Execute instruction to unconfigure unit	
Action? Next state	Exception Off	Execute Initial	Execute Initial
		Execute instruction to disable unit	
Action? Next state	Execute Off	Execute Off	Execute Off
		Execute instruction to enable unit	
Action? Next state	Execute Initial	Execute Initial	Execute Initial

Standard privileged instructions to initialize, save, and restore extension state are provided to insulate privileged code from details of the added extension state by treating the state as an opaque object.

NOTE

Many coprocessor extensions are only used in limited contexts that allows software to safely unconfigure or even disable units when done. This reduces the context-switch overhead of large stateful coprocessors.

We separate out floating-point state from other extension state, as when a floating-point unit is present the floating-point registers are part of the standard calling convention, and so user-mode software cannot know when it is safe to disable the floating-point unit.

The XS field provides a summary of all added extension state, but additional microarchitectural bits might be maintained in the extension to further reduce context save and restore overhead.

The SD bit is read-only and is set when either the FS, VS, or XS bits encode a Dirty state (i.e., `SD=(FS==0b11 OR XS==0b11 OR VS==0b11)`). This allows privileged code to quickly determine when no additional context save is required beyond the integer register set and `pc`.

The floating-point unit state is always initialized, saved, and restored using standard instructions (F, D, and/or Q), and privileged code must be aware of FLEN to determine the appropriate space to reserve for each `f` register.

Machine and Supervisor modes share a single copy of the FS, VS, and XS bits. Supervisor-level software normally uses the FS, VS, and XS bits directly to record the status with respect to the supervisor-level saved context. Machine-level software must be more conservative in saving and restoring the extension state in their corresponding version of the context.

NOTE

In any reasonable use case, the number of context switches between user and supervisor level should far outweigh the number of context switches to other privilege levels. Note that coprocessors should not require their context to be saved and restored to service asynchronous interrupts, unless the interrupt results in a user-level context swap.

3.1.1.6.8. Previous Expected Landing Pad (ELP) State in `mstatus` Register

The Zicfilp extension adds the `SPELP` and `MPELP` fields that hold the previous `ELP`, and are updated as specified in Preserving Expected Landing Pad State on Traps. The ***x***`PELP` fields are encoded as follows:

- 0 - `NO_LP_EXPECTED` - no landing pad instruction expected.
- 1 - `LP_EXPECTED` - a landing pad instruction is expected.

3.1.1.7. Machine Trap-Vector Base-Address (`mtvec`) Register

The `mtvec` register is an MXLEN-bit **WARL** read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

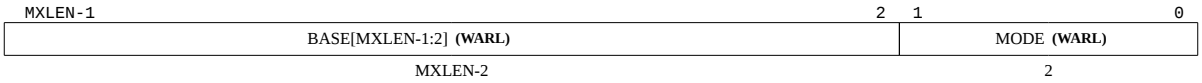


Figure 9. Encoding of mtvec MODE field.

The `mtvec` register must always be implemented, but can contain a read-only value. If `mtvec` is writable, the set of values the register may hold can vary by implementation. The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field. Note that the CSR contains only bits XLEN-1 through 2 of the address BASE. When used as an address, the lower two bits are filled with zeroes to obtain an XLEN-bit address that is always aligned on a 4-byte boundary.

NOTE

We allow for considerable flexibility in implementation of the trap vector base address. On the one hand, we do not wish to burden low-end implementations with a large number of state bits, but on the other hand, we wish to allow flexibility for larger systems.

Table 5. Encoding of mtvec MODE field.

Value	Name	Description
0	Direct	All traps set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	---	*Reserved*

The encoding of the MODE field is shown in Encoding of mtvec MODE field.. When MODE=Direct, all traps into machine mode cause the `pc` to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into machine mode cause the `pc` to be set to the address in the BASE field, whereas interrupts cause the `pc` to be set to the address in the BASE field plus four times the interrupt cause number. For example, a machine-mode timer interrupt (see Machine cause (`mcause`) register values after trap.) causes the `pc` to be set to BASE+`0x1c`.

An implementation may have different alignment constraints for different modes. In particular, MODE=Vectored may have stricter alignment constraints than MODE=Direct.

NOTE

Allowing coarser alignments in Vectored mode enables vectoring to be implemented without a hardware adder circuit.

Reset and NMI vector locations are given in a platform specification.

3.1.1.8. Machine Trap Delegation (`medeleg` and `mideleg`) Registers

By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the MRET instruction (3.1.3.2. Trap-Return Instructions). To increase performance, implementations can provide individual read/write bits within `medeleg` and `mideleg` to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level. The machine exception delegation register (`medeleg`) is a 64-bit read/write register. The machine interrupt delegation (`mideleg`) register is an MXLEN-bit read/write register.

In harts with S-mode, the `medeleg` and `mideleg` registers must exist, and setting a bit in `medeleg` or `mideleg` will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S-mode trap handler. In harts without S-mode, the `medeleg` and `mideleg` registers should not exist.

NOTE

In versions 1.9.1 and earlier , these registers existed but were hardwired to zero in M-mode only, or M/U without N harts. There is no reason to require they return zero in those cases, as the `misa` register indicates whether they exist.

When a trap is delegated to S-mode, the `scause` register is written with the trap cause; the `sepc` register is written with the virtual address of the instruction that took the trap; the `stval` register is written with an exception-specific datum; the SPP field of `mstatus` is written with the active privilege mode at the time of the trap; the SPIE field of `mstatus` is written with the value of the SIE field at the time of the trap; and the SIE field of `mstatus` is cleared. The `mcause`, `mepc`, and `mtval` registers and the MPP and MPIE fields of `mstatus` are not written.

An implementation can choose to subset the delegatable traps, with the supported delegatable bits found by writing one to every bit location, then reading back the value in `medeleg` or `mideleg` to see which bit positions hold a one.

An implementation shall not have any bits of `medeleg` be read-only one, i.e., any synchronous trap that can be delegated must support not being delegated. Similarly, an implementation shall not fix as read-only one any bits of `mideleg` corresponding to machine-level interrupts (but may do so for lower-level interrupts).

NOTE

Version 1.11 and earlier prohibited having any bits of `mideleg` be read-only one. Platform standards may always add such restrictions.

Traps never transition from a more-privileged mode to a less-privileged mode. For example, if M-mode has delegated illegal-instruction exceptions to S-mode, and M-mode software later executes an illegal instruction, the trap is taken in M-mode, rather than being delegated to S-mode. By contrast, traps may

be taken horizontally. Using the same example, if M-mode has delegated illegal-instruction exceptions to S-mode, and S-mode software later executes an illegal instruction, the trap is taken in S-mode.

Delegated interrupts result in the interrupt being masked at the delegator privilege level. For example, if the supervisor timer interrupt (STI) is delegated to S-mode by setting `mideleg`[5], STIs will not be taken when executing in M-mode. By contrast, if `mideleg`[5] is clear, STIs can be taken in any mode and regardless of current mode will transfer control to M-mode.

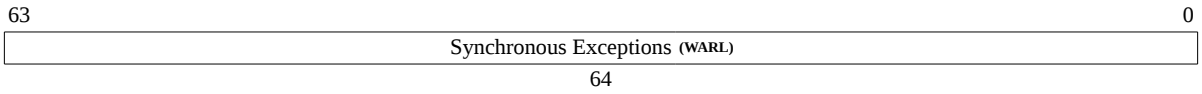


Figure 10. Machine Exception Delegation (`medeleg`) register.

`medeleg` has a bit position allocated for every synchronous exception shown in Machine cause (`mcause`) register values after trap., with the index of the bit position equal to the value returned in the `mcause` register (i.e., setting bit 8 allows user-mode environment calls to be delegated to a lower-privilege trap handler).

When XLEN=32, `medelegh` is a 32-bit read/write register that aliases bits 63:32 of `medeleg`. The `medelegh` register does not exist when XLEN=64.

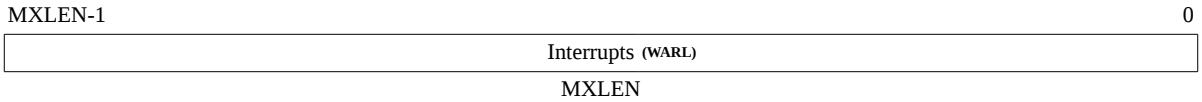


Figure 11. Machine Interrupt Delegation (`mideleg`) Register.

`mideleg` holds trap delegation bits for individual interrupts, with the layout of bits matching those in the `mip` register (i.e., STIP interrupt delegation control is located in bit 5).

For exceptions that cannot occur in less privileged modes, the corresponding `medeleg` bits should be read-only zero. In particular, `medeleg`[11] is read-only zero.

The `medeleg`[16] is read-only zero as double trap is not delegatable.

3.1.1.9. Machine Interrupt (`mip` and `mie`) Registers

The `mip` register is an MXLEN-bit read/write register containing information on pending interrupts, while `mie` is the corresponding MXLEN-bit read/write register containing interrupt enable bits. Interrupt cause number *i* (as reported in CSR `mcause`, 3.1.1.15. Machine Cause (`mcause`) Register) corresponds with bit *i* in both `mip` and `mie`. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform use.

NOTE

NOTE

Interrupts designated for platform use may be designated for custom use at the platform's discretion.

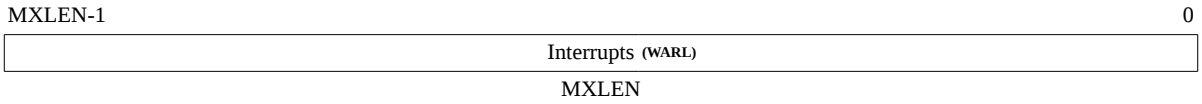


Figure 12. Machine Interrupt-Pending (`mip`) register.

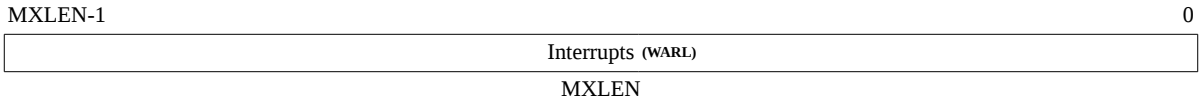


Figure 13. Machine Interrupt-Enable (`mie`) register

An interrupt *i* will trap to M-mode (causing the privilege mode to change to M-mode) if all of the following are true: (a) either the current privilege mode is M and the MIE bit in the `mstatus` register is set, or the current privilege mode has less privilege than M-mode; (b) bit *i* is set in both `mip` and `mie`; and (c) if register `mideleg` exists, bit *i* is not set in `mideleg`.

These conditions for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in `mip`, and must also be evaluated immediately following the execution of an *x*RET instruction or an explicit write to a CSR on which these interrupt trap conditions expressly depend (including `mip`, `mie`, `mstatus`, and `mideleg`).

Interrupts to M-mode take priority over any interrupts to lower privilege modes.

Each individual bit in register `mip` may be writable or may be read-only. When bit *i* in `mip` is writable, a pending interrupt *i* can be cleared by writing 0 to this bit. If interrupt *i* can become pending but bit *i* in `mip` is read-only, the implementation must provide some other mechanism for clearing the pending interrupt.

A bit in `mie` must be writable if the corresponding interrupt can ever become pending. Bits of `mie` that are not writable must be read-only zero.

The standard portions (bits 15:0) of the `mip` and `mie` registers are formatted as shown in Standard portion (bits 15:0) of `mip`. and Standard portion (bits 15:0) of `mie`. respectively.

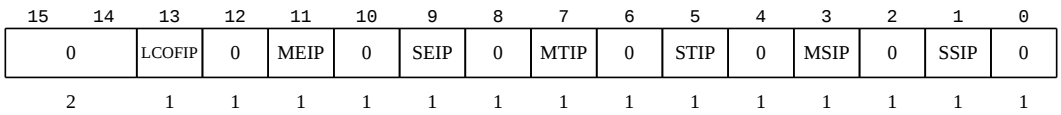


Figure 14. Standard portion (bits 15:0) of `mip`.

Figure 15. Standard portion (bits 15:0) of `mie`.

NOTE

The machine-level interrupt registers handle a few root interrupt sources which are assigned a fixed service priority for simplicity, while separate external interrupt controllers can implement a more complex prioritization scheme over a much larger set of interrupts that are then multiplexed into the machine-level interrupt sources.

The non-maskable interrupt is not made visible via the `mip` register as its presence is implicitly known when executing the NMI trap handler.

Bits `mip`.MEIP and `mie`.MEIE are the interrupt-pending and interrupt-enable bits for machine-level external interrupts. MEIP is read-only in `mip`, and is set and cleared by a platform-specific interrupt controller.

Bits `mip`.MTIP and `mie`.MTIE are the interrupt-pending and interrupt-enable bits for machine timer interrupts. MTIP is read-only in the `mip` register, and is cleared by writing to the memory-mapped machine-mode timer compare register.

Bits `mip`.MSIP and `mie`.MSIE are the interrupt-pending and interrupt-enable bits for machine-level software interrupts. MSIP is read-only in `mip`, and is written by accesses to memory-mapped control registers, which are used to provide machine-level interprocessor interrupts.

A hart's memory-mapped `msip` register is a 32-bit read/write register, where bits 31—1 read as zero and bit 0 contains the MSIP bit. When the memory-mapped `msip` register changes, it is guaranteed to be reflected in `mip`.MSIP eventually, but not necessarily immediately. If a system has only one hart, or if a platform standard supports the delivery of machine-level interprocessor interrupts through external interrupts (MEI) instead, then `mip`.MSIP and `mie`.MSIE may both be read-only zeros.

If supervisor mode is not implemented, bits SEIP, STIP, and SSIP of `mip` and SEIE, STIE, and SSIE of `mie` are read-only zeros.

If supervisor mode is implemented, bits `mip`.SEIP and `mie`.SEIE are the interrupt-pending and interrupt-enable bits for supervisor-level external interrupts. SEIP is writable in `mip`, and may be written by M-mode software to indicate to S-mode that an external interrupt is pending. Additionally, the platform-level interrupt controller may generate supervisor-level external interrupts. Supervisor-level external interrupts are made pending based on the logical-OR of the software-writable SEIP bit and the signal from the external interrupt controller. When `mip` is read with a CSR instruction, the value of the SEIP bit returned in the `rd` destination register is the logical-OR of the software-writable bit and the interrupt signal from the interrupt controller, but the signal from the interrupt controller is not used to calculate the value written to SEIP. Only the software-writable SEIP bit participates in the read-modify-write sequence of a CSRRS or CSRRC instruction.

NOTE

NOTE

For example, if we name the software-writable SEIP bit `B` and the signal from the external interrupt controller `E`, then if `csrrs t0, mip, t1` is executed, `t0[9]` is written with `B || E`, then `B` is written with `B || t1[9]`. If `csrrw t0, mip, t1` is executed, then `t0[9]` is written with `B || E`, and `B` is simply written with `t1[9]`. In neither case does `B` depend upon `E`.

The SEIP field behavior is designed to allow a higher privilege layer to mimic external interrupts cleanly, without losing any real external interrupts. The behavior of the CSR instructions is slightly modified from regular CSR accesses as a result.

If supervisor mode is implemented, its `mip`.STIP and `mie`.STIE are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. If the stimecmp register is not implemented, STIP is writable in mip, and may be written by M-mode software to deliver timer interrupts to S-mode. If the `stimecmp` (supervisor-mode timer compare) register is implemented, STIP is read-only in mip and reflects the supervisor-level timer interrupt signal resulting from stimecmp. This timer interrupt signal is cleared by writing `stimecmp` with a value greater than the current time value.

If supervisor mode is implemented, bits `mip`.SSIP and `mie`.SSIE are the interrupt-pending and interrupt-enable bits for supervisor-level software interrupts. SSIP is writable in `mip` and may also be set to 1 by a platform-specific interrupt controller.

If the Sscofpmf extension is implemented, bits `mip`.LCOFIP and `mie`.LCOFIE are the interrupt-pending and interrupt-enable bits for local-counter-overflow interrupts. LCOFIP is read-write in `mip` and reflects the occurrence of a local counter-overflow overflow interrupt request resulting from any of the `mhpmevent`*n*.OF bits being set. If the Sscofpmf extension is not implemented, `mip`.LCOFIP and `mie`.LCOFIE are read-only zeros.

Multiple simultaneous interrupts destined for M-mode are handled in the following decreasing priority order: MEI, MSI, MTI, SEI, SSI, STI, LCOFI.

NOTE

NOTE

The machine-level interrupt fixed-priority ordering rules were developed with the following rationale.

Interrupts for higher privilege modes must be serviced before interrupts for lower privilege modes to support preemption.

The platform-specific machine-level interrupt sources in bits 16 and above have platform-specific priority, but are typically chosen to have the highest service priority to support very fast local vectored interrupts.

External interrupts are handled before internal (timer/software) interrupts as external interrupts are usually generated by devices that might require low interrupt service times.

Software interrupts are handled before internal timer interrupts, because internal timer interrupts are usually intended for time slicing, where time precision is less important, whereas software interrupts are used for inter-processor messaging. Software interrupts can be avoided when high-precision timing is required, or high-precision timer interrupts can be routed via a different interrupt path. Software interrupts are located in the lowest four bits of `mip` as these are often written by software, and this position allows the use of a single CSR instruction with a five-bit immediate.

Restricted views of the `mip` and `mie` registers appear as the `sip` and `sie` registers for supervisor level. If an interrupt is delegated to S-mode by setting a bit in the `mideleg` register, it becomes visible in the `sip` register and is maskable using the `sie` register. Otherwise, the corresponding bits in `sip` and `sie` are read-only zero.

3.1.1.10. Hardware Performance Monitor

M-mode includes a basic hardware performance-monitoring facility. The `mcycle` CSR counts the number of clock cycles executed by the processor core on which the hart is running. The `minstret` CSR counts the number of instructions the hart has retired. The `mcycle` and `minstret` registers have 64-bit precision on all RV32 and RV64 harts.

The counter registers have an arbitrary value after the hart is reset, and can be written with a given value. Any CSR write takes effect after the writing instruction has otherwise completed. The `mcycle` CSR may be shared between harts on the same core, in which case writes to `mcycle` will be visible to those harts. The platform should provide a mechanism to indicate which harts share an `mcycle` CSR.

The hardware performance monitor includes 29 additional 64-bit event counters, `mhpmcounter3`-`mhpmcounter31`. The event selector CSRs, `mhpmevent3`-`mhpmevent31`, are 64-bit **WARL** registers that control which event causes the corresponding counter to increment. The meaning of these events is defined by the platform, but event 0 is defined to mean "no event." All counters should be implemented, but a legal implementation is to make both the counter and its corresponding event selector be read-only 0.

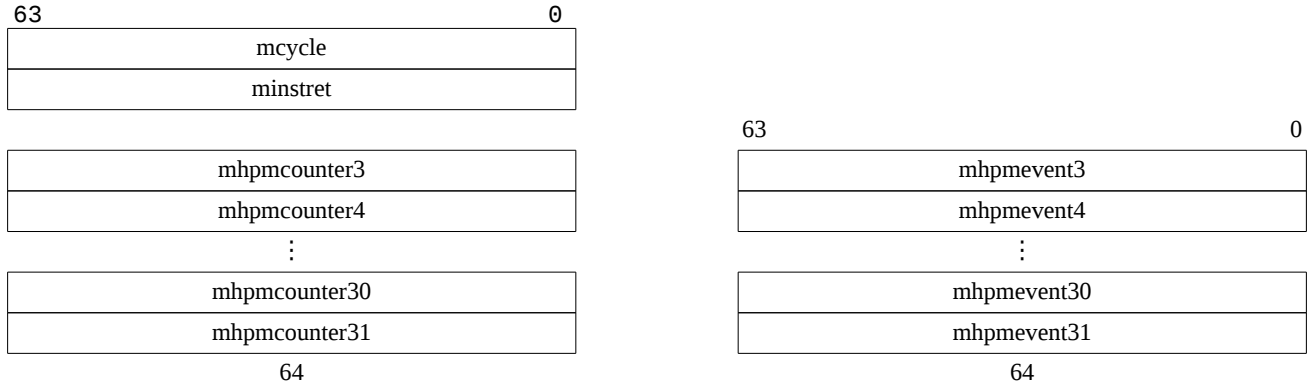


Figure 16. Hardware performance monitor counters.

The `mhpmcounters` are **WARL** registers that support up to 64 bits of precision on RV32 and RV64.

When XLEN=32, reads of the `mcycle`, `minstret`, `mhpmcounter`*n*, and `mhpmevent`*n* CSRs return bitj 31-0 of the corresponding register, and writes change only bits 31-0; reads of the `mcycleh`, `minstreth`, `mhpmcounter`*n*`h`, and `mhpmevent`*n*`h` CSRs return bits 63-32 of the corresponding register, and writes change only bits 63-32. The `mhpmevent`*n*`h` CSRs are provided only if the Sscofpmf extension is implemented.

3.1.1.11. Machine Counter-Enable (`mcounteren`) Register

The counter-enable `mcounteren` register is a 32-bit register that controls the availability of the hardware performance-monitoring counters to the next-lower privileged mode.

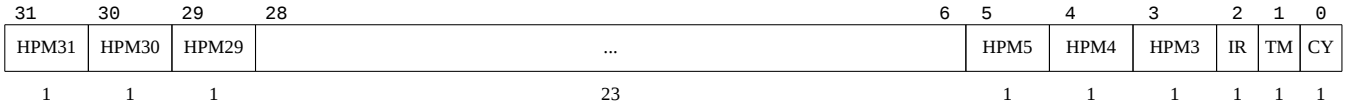


Figure 17. Counter-enable (`mcounteren`) register.

The settings in this register only control accessibility. The act of reading or writing this register does not affect the underlying counters, which continue to increment even when not accessible.

When the CY, TM, IR, or HPM*n* bit in the `mcounteren` register is clear, attempts to read the `cycle`, `time`, `instret`, or `hpmcountern` register while executing in S-mode or U-mode will cause an illegal-instruction exception. When one of these bits is set, access to the corresponding register is permitted in the next implemented privilege mode (S-mode if implemented, otherwise U-mode).

NOTE

The counter-enable bits support two common use cases with minimal hardware. For harts that do not need high-performance timers and counters, machine-mode software can trap accesses and implement all features in software. For harts that need high-performance timers and counters but are not concerned with obfuscating the underlying hardware counters, the counters can be directly exposed to lower privilege modes.

In addition, when the TM bit in the `mcounteren` register is clear, attempts to access the `stimecmp` or `vstimecmp` register while executing in a mode less privileged than M will cause an illegal-instruction exception. When this bit is set, access to the `stimecmp` or `vstimecmp` register is permitted in S-mode if implemented, and access to the `vstimecmp` register (via `stimecmp`) is permitted in VS-mode if implemented and not otherwise prevented by the TM bit in `hcounteren`.

The `cycle`, `instret`, and `hpmcountern` CSRs are read-only shadows of `mcycle`, `minstret`, and `mhpmcounter n`, respectively. The `time` CSR is a read-only shadow of the memory-mapped `mtime` register. Analogously, when XLEN=32, the `cycleh`, `instreth` and `hpmcounternh` CSRs are read-only shadows of `mcycleh`, `minstreth` and `mhpmcounternh`, respectively. When XLEN=32, the `timeh` CSR is a read-only shadow of the upper 32 bits of the memory-mapped `mtime` register, while `time` shadows only the lower 32 bits of `mtime`.

NOTE

Implementations can convert reads of the `time` and `timeh` CSRs into loads to the memory-mapped `mtime` register, or emulate this functionality on behalf of less-privileged modes in M-mode software.

In harts with U-mode, the `mcounteren` must be implemented, but all fields are **WARL** and may be read-only zero, indicating reads to the corresponding counter will cause an illegal-instruction exception when executing in a less-privileged mode. In harts without U-mode, the `mcounteren` register should not exist.

3.1.1.12. Machine Counter-Inhibit (`mcountinhibit`) Register

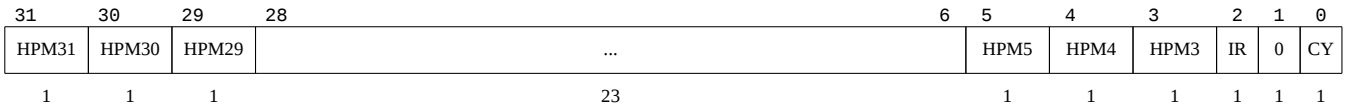


Figure 18. Counter-inhibit `mcountinhibit` register

The counter-inhibit register `mcountinhibit` is a 32-bit **WARL** register that controls which of the hardware performance-monitoring counters increment. The settings in this register only control whether the counters increment; their accessibility is not affected by the setting of this register.

When the CY, IR, or HPM*n* bit in the `mcountinhibit` register is clear, the `mcycle`, `minstret`, or `mhpmcountern` register increments as usual. When the CY, IR, or HPM*n* bit is set, the corresponding counter does not increment.

The `mcycle` CSR may be shared between harts on the same core, in which case the `mcountinhibit.CY` field is also shared between those harts, and so writes to `mcountinhibit.CY` will be visible to those harts.

If the `mcountinhibit` register is not implemented, the implementation behaves as though the register were set to zero.

NOTE

When the `mcycle` and `minstret` counters are not needed, it is desirable to conditionally inhibit them to reduce energy consumption. Providing a single CSR to inhibit all counters also allows the counters to be atomically sampled.

Because the `mtime` counter can be shared between multiple cores, it cannot be inhibited with the `mcountinhibit` mechanism.

3.1.1.13. Machine Scratch (`mscratch`) Register

The `mscratch` register is an MXLEN-bit read/write register dedicated for use by machine mode. Typically, it is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler.

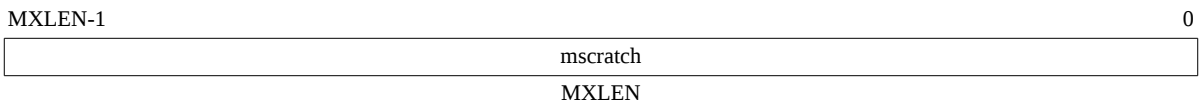


Figure 19. Machine-mode scratch register.

NOTE

The MIPS ISA allocated two user registers (`k0` / `k1`) for use by the operating system. Although the MIPS scheme provides a fast and simple implementation, it also reduces available user registers, and does not scale to further privilege levels, or nested traps. It can also require both registers are cleared before returning to user level to avoid a potential security hole and to provide deterministic debugging behavior.

The RISC-V user ISA was designed to support many possible privileged system environments and so we did not want to infect the user-level ISA with any OS-dependent features. The RISC-V CSR swap instructions can quickly save/restore values to the `mscratch` register. Unlike the MIPS design, the OS can rely on holding a value in the `mscratch` register while the user context is running.

3.1.1.14. Machine Exception Program Counter (`mepc`) Register

`mepc` is an MXLEN-bit read/write register formatted as shown in Machine exception program counter register.. The low bit of `mepc` (`mepc[0]`) is always zero. On implementations that support only IALIGN=32, the two low bits (`mepc[1:0]`) are always zero.

If an implementation allows IALIGN to be either 16 or 32 (by changing CSR `misa` , for example), then, whenever IALIGN=32, bit `mepc[1]` is masked on reads so that it appears to be 0. This masking occurs also for the implicit read by the MRET instruction. Though masked, `mepc[1]` remains writable when IALIGN=32.

`mepc` is a **WARL** register that must be able to hold all valid virtual addresses. It need not be capable of holding all possible invalid addresses. Prior to writing `mepc`, implementations may convert an invalid address into some other invalid address that `mepc` is capable of holding.

NOTE

When address translation is not in effect, virtual addresses and physical addresses are equal. Hence, the set of addresses `mepc` must be able to represent includes the set of physical addresses that can be used as a valid `pc` or effective address.

When a trap is taken into M-mode, `mepc` is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, `mepc` is never written by the implementation, though it may be explicitly written by software.

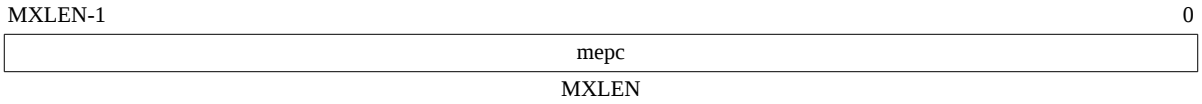


Figure 20. Machine exception program counter register.

3.1.1.15. Machine Cause (`mcause`) Register

The `mcause` register is an MXLEN-bit read-write register formatted as shown in Machine Cause (`mcause`) register.. When a trap is taken into M-mode, `mcause` is written with a code indicating the event that caused the trap. Otherwise, `mcause` is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the `mcause` register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception or interrupt. Machine cause (`mcause`) register values after trap. lists the possible machine-level exception codes.# The Exception Code is a **WLRL** field, so is only guaranteed to hold supported exception codes.

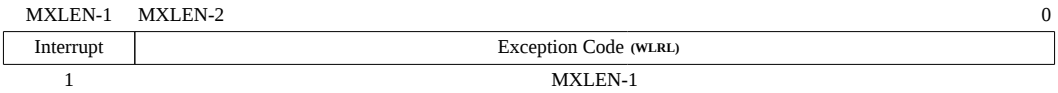


Figure 21. Machine Cause (`mcause`) register.

Note that load and load-reserved instructions generate load exceptions, whereas store, store-conditional, and AMO instructions generate store/AMO exceptions.

NOTE

NOTE

Interrupts can be separated from other traps with a single branch on the sign of the `mcause` register value. A shift left can remove the interrupt bit and scale the exception codes to index into a trap vector table.

We do not distinguish privileged instruction exceptions from illegal-instruction exceptions. This simplifies the architecture and also hides details of which higher-privilege instructions are supported by an implementation. The privilege level servicing the trap can implement a policy on whether these need to be distinguished, and if so, whether a given opcode should be treated as illegal or privileged.

If an instruction may raise multiple synchronous exceptions, the decreasing priority order of Synchronous exception priority in decreasing priority order. indicates which exception is taken and reported in `mcause`. The priority of any custom synchronous exceptions is implementation-defined.

Table 6. Machine cause (`mcause`) register values after trap.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12	*Reserved*
1	13	Counter-overflow interrupt
1	14-15	*Reserved*
1	≥16	*Designated for platform use*

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16	Double trap
0	17	*Reserved*
0	18	Software check
0	19	Hardware error
0	20-23	*Reserved*
0	24-31	*Designated for custom use*
0	32-47	*Reserved*
0	48-63	*Designated for custom use*
0	≥64	*Reserved*

Table 7. Synchronous exception priority in decreasing priority order.

Priority	Exc.Code	Description
Highest	3	Instruction address breakpoint
		During instruction address translation:
	12, 1	First encountered page fault or access fault
		With physical address for instruction:
	1	Instruction access fault
	2	Illegal instruction
	0	Instruction address misaligned
	8,9,11	Environment call
	3	Environment break
	3	Load/store/AMO address breakpoint
		Optionally:
	4,6	Load/store/AMO address misaligned
		During address translation for an explicit memory access:
	13, 15, 5, 7	First encountered page fault or access fault
		With physical address for an explicit memory access:
	5,7	Load/store/AMO access fault
		If not higher priority:
Lowest	4,6	Load/store/AMO address misaligned

When a virtual address is translated into a physical address, the address translation algorithm determines what specific exception may be raised.

Load/store/AMO address-misaligned exceptions may have either higher or lower priority than load/store/AMO page-fault and access-fault exceptions.

NOTE

NOTE

The relative priority of load/store/AMO address-misaligned and page-fault exceptions is implementation-defined to flexibly cater to two design points. Implementations that never support misaligned accesses can unconditionally raise the misaligned-address exception without performing address translation or protection checks. Implementations that support misaligned accesses only to some physical addresses must translate and check the address before determining whether the misaligned access may proceed, in which case raising the page-fault exception or access is more appropriate.

Instruction address breakpoints have the same cause value as, but different priority than, data address breakpoints (a.k.a. watchpoints) and environment break exceptions (which are raised by the EBREAK instruction).

Instruction address-misaligned exceptions are raised by control-flow instructions with misaligned targets, rather than by the act of fetching an instruction. Therefore, these exceptions have lower priority than other instruction address exceptions.

NOTE

A software-check exception is a synchronous exception that is triggered when there are violations of checks and assertions defined by ISA extensions that aim to safeguard the integrity of software assets, including e.g. control-flow and memory-access constraints. When this exception is raised, the `xtval` register is set either to 0 or to an informative value defined by the extension that stipulated the exception be raised. The priority of this exception, relative to other synchronous exceptions, depends on the cause of this exception and is defined by the extension that stipulated the exception be raised.

A hardware-error exception is a synchronous exception triggered when corrupted or uncorrectable data is accessed explicitly or implicitly by an instruction. In this context, "data" encompasses all types of information used within a RISC-V hart. Upon a hardware-error exception, the `xepc` register is set to the address of the instruction that attempted to access corrupted data, while the `xtval` register is set either to 0 or to the virtual address of an instruction fetch, load, or store that attempted to access corrupted data. The priority of hardware-error exception is implementation-defined, but any given occurrence is generally expected to be recognized at the point in the overall priority order at which the hardware error is discovered.

3.1.1.16. Machine Trap Value (`mtval`) Register

The `mtval` register is an MXLEN-bit read-write register formatted as shown in Machine Trap Value (`mtval`) register.. When a trap is taken into M-mode, `mtval` is either set to zero or written with exception-specific information to assist software in handling the trap. Otherwise, `mtval` is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set `mtval` informatively, which may unconditionally set it to zero, and which may exhibit either behavior, depending on the underlying event that caused the exception. If the hardware platform specifies that no exceptions set `mtval` to a nonzero value, then `mtval` is read-only zero.

If `mtval` is written with a nonzero value when a breakpoint, address-misaligned, access-fault, page-fault, or hardware-error exception occurs on an instruction fetch, load, or store, then `mtval` will contain the faulting virtual address.

On a breakpoint exception raised by an EBREAK or C.EBREAK instruction, `mtval` is written with either zero or the virtual address of the instruction.

NOTE

For breakpoint exceptions raised by [C.]EBREAK, the virtual address of the instruction is already recorded in `mepc`. Recording the same address in `mtval` is redundant; the option is provided for backwards compatibility.

When page-based virtual memory is enabled, `mtval` is written with the faulting virtual address, even for physical-memory access-fault exceptions. This design reduces datapath cost for most implementations, particularly those with hardware page-table walkers.

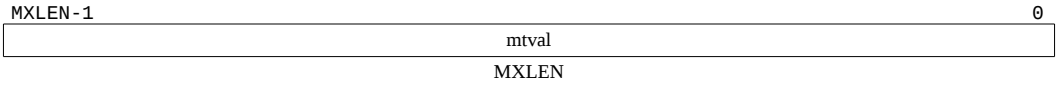


Figure 22. Machine Trap Value (`mtval`) register.

If `mtval` is written with a nonzero value when a misaligned load or store causes an access-fault, page-fault, or hardware-error exception, then `mtval` will contain the virtual address of the portion of the access that caused the fault.

If `mtval` is written with a nonzero value when an instruction access-fault, page-fault, or hardware-error exception occurs on a hart with variable-length instructions, then `mtval` will contain the virtual address of the portion of the instruction that caused the fault, while `mepc` will point to the beginning of the instruction.

The `mtval` register can optionally also be used to return the faulting instruction bits on an illegal-instruction exception (`mepc` points to the faulting instruction in memory). If `mtval` is written with a nonzero value when an illegal-instruction exception occurs, then `mtval` will contain the shortest of:#

- the actual faulting instruction
- the first ILEN bits of the faulting instruction
- the first MXLEN bits of the faulting instruction

The value loaded into `mtval` on an illegal-instruction exception is right-justified and all unused upper bits are cleared to zero.

NOTE

NOTE

Capturing the faulting instruction in `mtval` reduces the overhead of instruction emulation, potentially avoiding several partial instruction loads if the instruction is misaligned, and likely data cache misses or slow uncached accesses when loads are used to fetch the instruction into a data register. There is also a problem of atomicity if another agent is manipulating the instruction memory, as might occur in a dynamic translation system.

A requirement is that the entire instruction (or at least the first MXLEN bits) are fetched into `mtval` before taking the trap. This should not constrain implementations, which would typically fetch the entire instruction before attempting to decode the instruction, and avoids complicating software handlers.

A value of zero in `mtval` signifies either that the feature is not supported, or an illegal zero instruction was fetched. A load from the instruction memory pointed to by `mepc` can be used to distinguish these two cases (or alternatively, the system configuration information can be interrogated to install the appropriate trap handling before runtime).

On a trap caused by a software-check exception, the `mtval` register holds the cause for the exception. The following encodings are defined:

- 0 - No information provided.
- 2 - Landing Pad Fault. Defined by the Zicfilp extension (Landing Pad (Zicfilp)).
- 3 - Shadow Stack Fault. Defined by the Zicfiss extension (Shadow Stack (Zicfiss)).

For other traps, `mtval` is set to zero, but a future standard may redefine `mtval`'s setting for other traps.

If `mtval` is not read-only zero, it is a **WARL** register that must be able to hold all valid virtual addresses and the value zero. It need not be capable of holding all possible invalid addresses. Prior to writing `mtval`, implementations may convert an invalid address into some other invalid address that `mtval` is capable of holding. If the feature to return the faulting instruction bits is implemented, `mtval` must also be able to hold all values less than 2^N, where N is the smaller of MXLEN and ILEN.

3.1.1.17. Machine Configuration Pointer (`mconfigptr`) Register

The `mconfigptr` register is an MXLEN-bit read-only CSR formatted as shown in Machine Configuration Pointer (`mconfigptr`) register., that holds the physical address of a configuration data structure. Software can traverse this data structure to discover information about the harts, the platform, and their configuration.

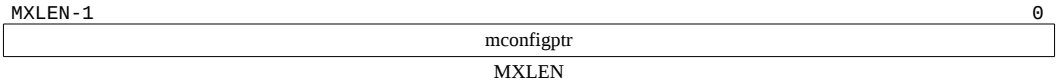


Figure 23. Machine Configuration Pointer (`mconfigptr`) register.

The pointer alignment in bits must be no smaller than MXLEN: i.e., if MXLEN is 8×*n*, then `mconfigptr`[$\log_{2n}$-1:0] must be zero.

The `mconfigptr` register must be implemented, but it may be zero to indicate the configuration data structure does not exist or that an alternative mechanism must be used to locate it.

NOTE

The format and schema of the configuration data structure have yet to be standardized.

While the `mconfigptr` register will simply be hardwired in some implementations, other implementations may provide a means to configure the value returned on CSR reads. For example, `mconfigptr` might present the value of a memory-mapped register that is programmed by the platform or by M-mode software towards the beginning of the boot process.

3.1.1.18. Machine Environment Configuration (`menvcfg`) Register

The `menvcfg` CSR is a 64-bit read/write register, formatted as shown in Machine environment configuration (`menvcfg`) register., that controls certain characteristics of the execution environment for modes less privileged than M.

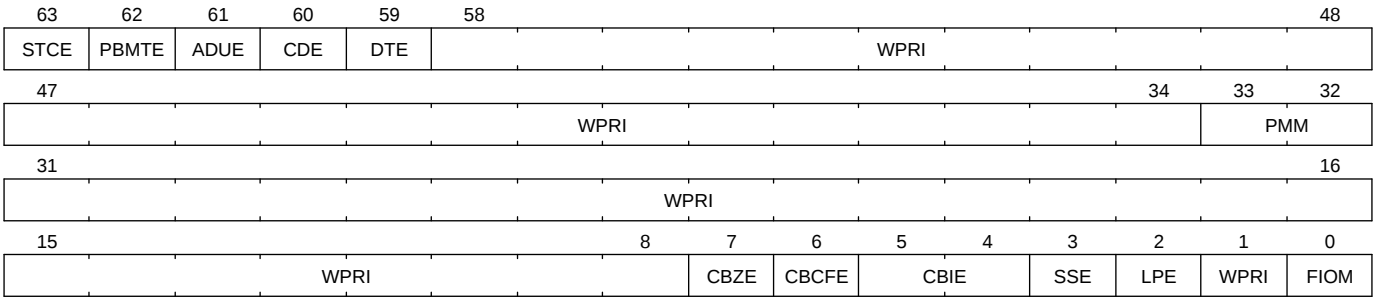


Figure 24. Machine environment configuration (`menvcfg`) register.

If bit FIOM (Fence of I/O implies Memory) is set to one in `menvcfg`, FENCE instructions executed in modes less privileged than M are modified so the requirement to order accesses to device I/O implies also the requirement to order main memory accesses. Modified interpretation of FENCE predecessor and successor sets for modes less privileged than M when FIOM=1. details the modified interpretation of FENCE instruction bits PI, PO, SI, and SO for modes less privileged than M when FIOM=1.

Similarly, for modes less privileged than M when FIOM=1, if an atomic instruction that accesses a region ordered as device I/O has its *aq* and/or *rl* bit set, then that instruction is ordered as though it accesses both device I/O and memory.

If S-mode is not supported, or if `satp`.MODE is read-only zero (always Bare), the implementation may make FIOM read-only zero.

Table 8. Modified interpretation of FENCE predecessor and successor sets for modes less privileged than M when FIOM=1.

Instruction bit	Meaning when set
PI	Predecessor device input and memory reads (PR implied)
PO	Predecessor device output and memory writes (PW implied)
SI	Successor device input and memory reads (SR implied)
SO	Successor device output and memory writes (SW implied)

NOTE

Bit FIOM is needed in `menvcfg` so M-mode can emulate the hypervisor extension of "H" Extension for Hypervisor Support, which has an equivalent FIOM bit in the hypervisor CSR `henvcfg`.

The PBMTE bit controls whether the Svpbmt extension is available for use in S-mode and G-stage address translation (i.e., for page tables pointed to by `satp` or `hgatp`). When PBMTE=1, Svpbmt is available for S-mode and G-stage address translation. When PBMTE=0, the implementation behaves as though Svpbmt were not implemented. If Svpbmt is not implemented, PBMTE is read-only zero. Furthermore, for implementations with the hypervisor extension, `henvcfg`.PBMTE is read-only zero if `menvcfg`.PBMTE is zero.

After changing `menvcfg`.PBMTE, executing an SFENCE.VMA instruction with *rs1*=`x0` and *rs2*=`x0` suffices to synchronize address-translation caches with respect to the altered interpretation of page-table entries' PBMT fields. See Memory-Management Fences for additional synchronization requirements when the hypervisor extension is implemented.

If the Svadu extension is implemented, the ADUE bit controls whether hardware updating of PTE A/D bits is enabled for S-mode and G-stage address translations. When ADUE=1, hardware updating of PTE A/D bits is enabled during S-mode address translation, and the implementation behaves as though the Svade extension were not implemented for S-mode address translation. When the hypervisor extension is implemented, if ADUE=1, hardware updating of PTE A/D bits is enabled during G-stage address translation, and the implementation behaves as though the Svade extension were not implemented for G-stage address translation. When ADUE=0, the implementation behaves as though Svade were implemented for S-mode and G-stage address translation. If Svadu is not implemented, ADUE is read-only zero. Furthermore, for implementations with the hypervisor extension, `henvcfg`.ADUE is read-only zero if `menvcfg`.ADUE is zero.

After changing `menvcfg`.ADUE, executing an SFENCE.VMA instruction with *rs1*=`x0` and *rs2*=`x0` suffices to synchronize address-translation caches with respect to the altered interpretation of page-table entries' A/D bits. See Memory-Management Fences for additional synchronization requirements when the hypervisor extension is implemented.

NOTE

NOTE

The Svade extension requires page-fault exceptions be raised when PTE A/D bits need be set, hence Svade is implemented when ADUE=0.

If the Smcdeleg extension is implemented, the CDE (Counter Delegation Enable) bit controls whether Zicntr and Zihpm counters can be delegated to S-mode. When CDE=1, the Smcdeleg extension is enabled, see "Smcdeleg/Ssccfg" Counter Delegation Extensions. When CDE=0, the Smcdeleg and Ssccfg extensions appear to be not implemented. If Smcdeleg is not implemented, CDE is read-only zero.

The Sstc extension adds the `STCE` (STimecmp Enable) bit to `menvcfg` CSR. When the Sstc extension is not implemented, `STCE` is read-only zero. The `STCE` bit enables `stimecmp` for S-mode when set to one. When this extension is implemented and `STCE` in `menvcfg` is zero, an attempt to access `stimecmp` in a mode other than M-mode raises an illegal-instruction exception, `STCE` in `henvcfg` is read-only zero, and `STIP` in `mip` and `sip` reverts to its defined behavior as if this extension is not implemented. Further, if the H extension is implemented, then `hip`.VSTIP also reverts its defined behavior as if this extension is not implemented.

The Zicboz extension adds the `CBZE` (Cache Block Zero instruction enable) field to `menvcfg`. When the `CBZE` field is set to 1, it enables execution of the cache block zero instruction, `CBO.ZERO`, in modes less privileged than M. Otherwise, the instruction raises an illegal-instruction exception in modes less privileged than M. When the Zicboz extension is not implemented, `CBZE` is read-only zero.

The Zicbom extension adds the `CBCFE` (Cache Block Clean and Flush instruction Enable) field to `menvcfg`. When the `CBCFE` field is set to 1, it enables execution of the cache block clean instruction (`CBO.CLEAN`) and the cache block flush instruction (`CBO.FLUSH`) in modes less privileged than M. Otherwise, these instructions raise an illegal-instruction exception in modes less privileged than M. When the Zicbom extension is not implemented, `CBCFE` is read-only zero.

The Zicbom extension adds the `CBIE` (Cache Block Invalidate instruction Enable) WARL field to `menvcfg` to control execution of the cache block invalidate instruction (`CBO.INVAL`) in modes less privileged than M. When `CBIE` is set to `00b`, the instruction raises an illegal-instruction exception in modes less privileged than M. When the Zicbom extension is not implemented, `CBIE` is read-only zero. The encoding `10b` is reserved. When `CBIE` is set to `01b` or `11b`, and when enabled for execution in modes less privileged than M, it behaves as follows:

- `01b` — The instruction is executed and performs a flush operation, even if configured by a mode less privileged than M to perform an invalidate operation.
- `11b` — The instruction is executed and performs an invalidate operation, unless configured by a mode less privileged than M to perform a flush operation.

If the Smnpm extension is implemented, the `PMM` field enables or disables pointer masking (see Pointer Masking Extensions) for the next-lower privilege mode (S-/HS-mode if S-mode is implemented, or U-mode otherwise), according to the values in Legal values of `PMM` WARL field. If Smnpm is not implemented, `PMM` is read-only zero. The `PMM` field is read-only zero for RV32.

Table 9. Legal 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)

The Zicfilp extension adds the `LPE` field in `menvcfg`. When the `LPE` field is set to 1 and S-mode is implemented, the Zicfilp extension is enabled in S-mode. If `LPE` field is set to 1 and S-mode is not implemented, the Zicfilp extension is enabled in U-mode. When the `LPE` field is 0, the Zicfilp extension is not enabled in S-mode, and the following rules apply to S-mode. If the `LPE` field is 0 and S-mode is not implemented, then the same rules apply to U-mode.

- The hart does not update the `ELP` state; it remains as `NO_LP_EXPECTED`.
- The `LPAD` instruction operates as a no-op.

The Zicfiss extension adds the `SSE` field to `menvcfg`. When the `SSE` field is set to 1 the Zicfiss extension is activated in S-mode. When `SSE` field is 0, the following rules apply to privilege modes that are less than M:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- The `pte.xwr=010b` encoding in VS/S-stage page tables becomes reserved.
- `SSAMOSWAP.W/D` raises an illegal-instruction exception.

When `menvcfg.SSE` is 0, the `henvcfg.SSE` and `senvcfg.SSE` fields are read-only zero.

The Ssdbltrp extension adds the double-trap-enable (`DTE`) field in `menvcfg`. When `menvcfg.DTE` is zero, the implementation behaves as though Ssdbltrp is not implemented. When Ssdbltrp is not implemented `sstatus.SDT`, `vsstatus.SDT`, and `henvcfg.DTE` bits are read-only zero.

When XLEN=32, `menvcfgh` is a 32-bit read/write register that aliases bits 63:32 of `menvcfg`. The `menvcfgh` register does not exist when XLEN=64.

If U-mode is not supported, then registers `menvcfg` and `menvcfgh` do not exist.

3.1.1.19. Machine Security Configuration (`mseccfg`) Register

`mseccfg` is a 64-bit read/write register, formatted as shown in Machine security configuration (`mseccfg`) register., that controls security features. It exists if any extension that adds a field to `mseccfg` is implemented. Otherwise, it is reserved.

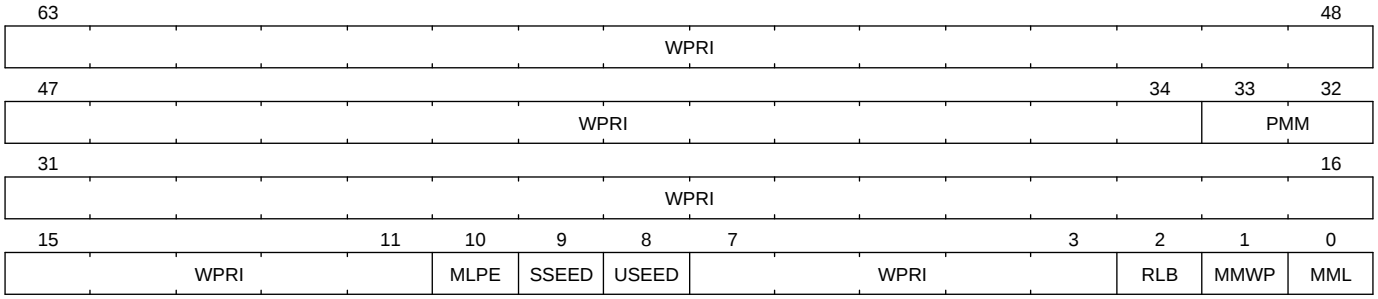


Figure 25. Machine security configuration (`mseccfg`) register.

The Zkr extension adds the `SSEED` and `USEED` fields to the `mseccfg` CSR to control access to the `seed` CSR from modes less privileged than M.

When `USEED` is 0, access to the `seed` CSR in U-mode raises an illegal-instruction exception. When `USEED` is 1, read-write access to the `seed` CSR from U-mode is allowed; all other types of accesses raise an illegal-instruction exception. If Zkr or U-mode is not implemented, `USEED` is read-only zero.

When `SSEED` is 0, access to the `seed` CSR from S-/HS-mode raises an illegal-instruction exception. When `SSEED` is 1, read-write access to the `seed` CSR from S-/HS-mode is allowed; all other types of accesses raise an illegal-instruction exception. If Zkr or S-mode is not implemented, `SSEED` is read-only zero.

When the H extension is also implemented, access to the `seed` CSR from an HS-qualified instruction leads to a virtual-instruction exception in VS and VU modes; all other types of accesses raise an illegal-instruction exception.

Table 10. Entropy Source Access Control.

Mode	SSEED	USEED	Description

Mode	`SSEED`	`USEED`	Description
M	-	-	The `seed` CSR is always available in machine mode as normal (with a CSR read-write instruction.) Attempted read without a write raises an illegal-instruction exception regardless of mode and access control bits.
U	-	`0`	Any `seed` CSR access raises an illegal-instruction exception.
U	-	`1`	The `seed` CSR is accessible as normal. No exception is raised for read-write.
S/HS	`0`	-	Any `seed` CSR access raises an illegal-instruction exception.
S/HS	`1`	-	The `seed` CSR is accessible as normal. No exception is raised for read-write.
VS/VU	`0`	-	Any `seed` CSR access raises an illegal-instruction exception.
VS/VU	`1`	-	A read-write `seed` access raises a virtual-instruction exception, while other access conditions raise an illegal-instruction exception.

The Smepmp extension adds the `RLB`, `MMWP`, and the `MML` fields in `mseccfg`.

When `mseccfg.RLB` (Rule Locking Bypass) a WARL field that provides a mechanism to temporarily modify **Locked** PMP rules. When `mseccfg.RLB` is 1, locked PMP rules may be removed or modified and locked PMP rules may be edited. When `mseccfg.RLB` is 0 and `pmpcfg.L` is 1 in any rule or entry (including disabled entries), then `mseccfg.RLB` remains 0 and any further modifications to `mseccfg.RLB` are ignored until a **PMP reset**.

NOTE

NOTE

This feature is intended to be used as a debug mechanism, or as a temporary workaround during the boot process for simplifying software, and optimizing the allocation of memory and PMP rules. Using this functionality under normal operation, after the boot process is completed, should be avoided since it weakens the protection of *M-mode-only* rules. Vendors who don't need this functionality may hardwire this field to 0.

The terminology used to specify the fields introduced by the Smepmp extension is listed in "Smepmp" Extension for PMP Enhancements for memory access and execution prevention in Machine mode.

The `mseccfg.MMWP` (Machine-Mode Allowlist Policy) is a WARL field. This field changes the default PMP policy for Machine mode when accessing memory regions that don't have a matching PMP rule. This is a sticky bit, meaning that once set it cannot be unset until a **PMP reset**. When set it changes the default PMP policy for M-mode when accessing memory regions that don't have a matching **PMP rule**, to **denied** instead of **ignored**.

The `mseccfg.MML` (Machine Mode Lockdown) is a WARL field. The `MML` bit changes the interpretation of the `pmpcfg.L` bit defined in 3.1.7.1.2. Locking and Privilege Mode. This is a sticky bit, meaning that once set it cannot be unset until a **PMP reset**. When `mseccfg.MML` is set the system's behavior changes in the following way:

1. The meaning of `pmpcfg.L` changes: Instead of marking a rule as **locked** and **enforced** in all modes, it now marks a rule as **M-mode-only** when set and **S/U-mode-only** when unset. The formerly reserved encoding of `pmpcfg.RW=01`, and the encoding `pmpcfg.LRWX=1111`, now encode a **Shared-Region**.

   An *M-mode-only* rule is **enforced** on Machine mode and **denied** in Supervisor or User mode. It also remains **locked** so that any further modifications to its associated configuration or address registers are ignored until a **PMP reset**, unless `mseccfg.RLB` is set.

   An *S/U-mode-only* rule is **enforced** on Supervisor and User modes and **denied** on Machine mode.

   A *Shared-Region* rule is **enforced** on all modes, with restrictions depending on the `pmpcfg.L` and `pmpcfg.X` bits:

   - A *Shared-Region* rule where `pmpcfg.L` is not set can be used for sharing data between M-mode and S/U-mode, so is not executable. M-mode has read/write access to that region, and S/U-mode has read access if `pmpcfg.X` is not set, or read/write access if `pmpcfg.X` is set.
   - A *Shared-Region* rule where `pmpcfg.L` is set can be used for sharing code between M-mode and S/U-mode, so is not writable. Both M-mode and S/U-mode have execute access on the region, and M-mode also has read access if `pmpcfg.X` is set. The rule remains **locked** so

that any further modifications to its associated configuration or address registers are ignored until a **PMP reset**, unless `mseccfg.RLB` is set.

- The encoding `pmpcfg.LRWX=1111` can be used for sharing data between M-mode and S/U mode, where both modes only have read-only access to the region. The rule remains **locked** so that any further modifications to its associated configuration or address registers are ignored until a **PMP reset**, unless `mseccfg.RLB` is set.

2. Adding a rule with executable privileges that either is **M-mode-only** or a **locked Shared-Region** is not possible and such `pmpcfg` writes are ignored, leaving `pmpcfg` unchanged. This restriction can be temporarily lifted by setting `mseccfg.RLB` e.g. during the boot process.
3. Executing code with Machine mode privileges is only possible from memory regions with a matching **M-mode-only** rule or a **locked Shared-Region** rule with executable privileges. Executing code from a region without a matching rule or with a matching *S/U-mode-only* rule is **denied**.
4. If `mseccfg.MML` is not set, the combination of `pmpcfg.RW=01` remains reserved for future standard use.

If the Smmpm extension is implemented, the `PMM` field enables or disables pointer masking (see ZPM) for M-mode according to the values in Legal values of `PMM` WARL field. If Smmpm is not implemented, `PMM` is read-only zero. The `PMM` field is read-only zero for RV32.

Table 11. Legal 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)

NOTE

`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.

The Zicfilp extension adds the `MLPE` field in `mseccfg`. When `MLPE` field is 1, Zicfilp extension is enabled in M-mode. When the `MLPE` field is 0, the Zicfilp extension is not enabled in M-mode and the following rules apply to M-mode.

- The hart does not update the `ELP` state; it remains as `NO_LP_EXPECTED`.
- The `LPAD` instruction operates as a no-op.

When XLEN=32 only, `mseccfgh` is a 32-bit read/write register that aliases bits 63:32 of `mseccfg`. Register `mseccfgh` exists when XLEN=32 and `mseccfg` is implemented; it does not exist when XLEN=64.

3.1.2. Machine-Level Memory-Mapped Registers

3.1.2.1. Machine Timer (`mtime` and `mtimecmp`) Registers

Platforms provide a real-time counter, exposed as a memory-mapped machine-mode read-write register, `mtime`. `mtime` must increment at constant frequency, and the platform must provide a mechanism for determining the period of an `mtime` tick. The `mtime` register will wrap around if the count overflows.

The `mtime` register has a 64-bit precision on all RV32 and RV64 systems. Platforms provide a 64-bit memory-mapped machine-mode timer compare register (`mtimecmp`). A machine timer interrupt becomes pending whenever `mtime` contains a value greater than or equal to `mtimecmp`, treating the values as unsigned integers. The interrupt remains posted until `mtimecmp` becomes greater than `mtime` (typically as a result of writing `mtimecmp`). The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the `mie` register.

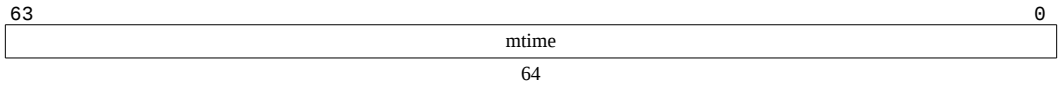


Figure 26. Machine time register (memory-mapped control register).

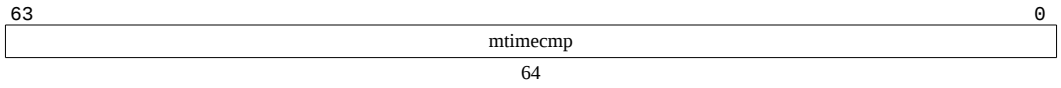


Figure 27. Machine time compare register (memory-mapped control register).

NOTE

The timer facility is defined to use wall-clock time rather than a cycle counter to support modern processors that run with a highly variable clock frequency to save energy through dynamic voltage and frequency scaling.

Accurate real-time clocks (RTCs) are relatively expensive to provide (requiring a crystal or MEMS oscillator) and have to run even when the rest of system is powered down, and so there is usually only one in a system located in a different frequency/voltage domain from the processors. Hence, the RTC must be shared by all the harts in a system and accesses to the RTC will potentially incur the penalty of a voltage-level-shifter and clock-domain crossing. It is thus more natural to expose `mtime` as a memory-mapped register than as a CSR.

Lower privilege levels do not have their own `timecmp` registers. Instead, machine-mode software can implement any number of virtual timers on a hart by multiplexing the next timer interrupt into the `mtimecmp` register.

Simple fixed-frequency systems can use a single clock for both cycle counting and wall-clock time.

If the result of the comparison between `mtime` and `mtimecmp` changes, it is guaranteed to be reflected in MTIP eventually, but not necessarily immediately.

NOTE

A spurious timer interrupt might occur if an interrupt handler increments `mtimecmp` then immediately returns, because MTIP might not yet have fallen in the interim. All software should be written to assume this event is possible, but most software should assume this event is extremely unlikely. It is almost always more performant to incur an occasional spurious timer interrupt than to poll MTIP until it falls.

In RV32, memory-mapped writes to `mtimecmp` modify only one 32-bit part of the register. The following code sequence sets a 64-bit `mtimecmp` value without spuriously generating a timer interrupt due to the intermediate value of the comparand:

For RV64, naturally aligned 64-bit memory accesses to the `mtime` and `mtimecmp` registers are additionally supported and are atomic.

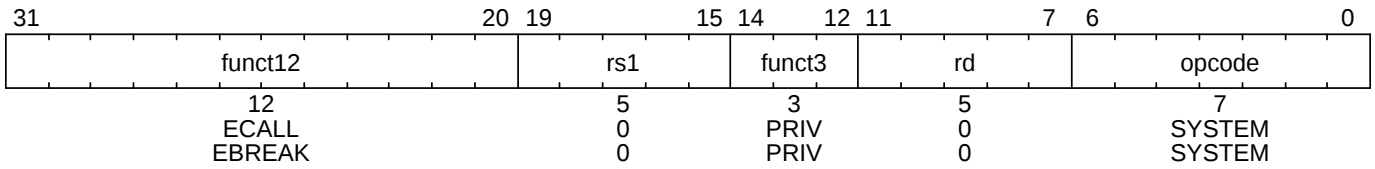
Sample code for setting the 64-bit time comparand in RV32 assuming a little-endian memory system and that the registers live in a strongly ordered I/O region. Storing -1 to the low-order bits of `mtimecmp` prevents `mtimecmp` from temporarily becoming smaller than the lesser of the old and new values.

```
# New comparand is in a1:a0.
li t0, -1
la t1, mtimecmp
sw t0, 0(t1)     # No smaller than old value.
sw a1, 4(t1)     # No smaller than new value.
sw a0, 0(t1)     # New value.
```

The `time` CSR is a read-only shadow of the memory-mapped `mtime` register. When XLEN=32, the `timeh` CSR is a read-only shadow of the upper 32 bits of the memory-mapped `mtime` register, while `time` shadows only the lower 32 bits of `mtime`. When `mtime` changes, it is guaranteed to be reflected in `time` and `timeh` eventually, but not necessarily immediately.

3.1.3. Machine-Mode Privileged Instructions

3.1.3.1. Environment Call and Breakpoint



The ECALL instruction is used to make a request to the supporting execution environment. When executed in U-mode, S-mode, or M-mode, it generates an environment-call-from-U-mode exception,

environment-call-from-S-mode exception, or environment-call-from-M-mode exception, respectively, and performs no other operation.

NOTE

ECALL generates a different exception for each originating privilege mode so that environment call exceptions can be selectively delegated. A typical use case for Unix-like operating systems is to delegate to S-mode the environment-call-from-U-mode exception but not the others.

The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment. Unless overridden by an external debug environment, EBREAK raises a breakpoint exception and performs no other operation.

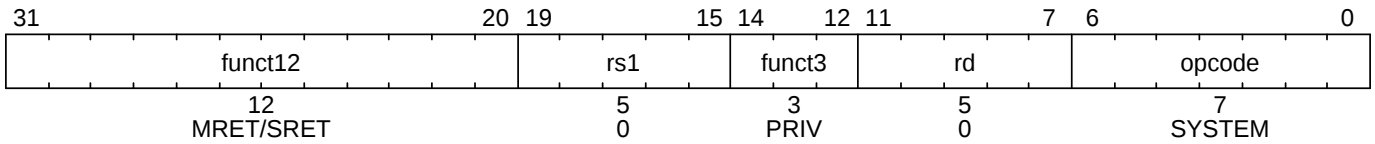
NOTE

As described in the "C" Standard Extension for Compressed Instructions in Volume I of this manual, the C.EBREAK instruction performs the same operation as the EBREAK instruction.

ECALL and EBREAK cause the receiving privilege mode's `epc` register to be set to the address of the ECALL or EBREAK instruction itself, *not* the address of the following instruction. As ECALL and EBREAK cause synchronous exceptions, they are not considered to retire, and should not increment the `minstret` CSR.

3.1.3.2. Trap-Return Instructions

Instructions to return from trap are encoded under the PRIV minor opcode.



To return after handling a trap, there are separate trap return instructions per privilege level, MRET and SRET. MRET is always provided. SRET must be provided if supervisor mode is supported, and should raise an illegal-instruction exception otherwise. SRET should also raise an illegal-instruction exception when TSR=1 in `mstatus`, as described in 3.1.1.6.6. Virtualization Support in `mstatus` Register. An *x*RET instruction can be executed in privilege mode *x* or higher, where executing a lower-privilege *x*RET instruction will pop the relevant lower-privilege interrupt enable and privilege mode stack. Attempting to execute an *x*RET instruction in a mode less privileged than *x* will raise an illegal-instruction exception.

In addition to manipulating the privilege stack as described in 3.1.1.6.1. Privilege and Global Interrupt-Enable Stack in `mstatus` register, *x*RET sets the `pc` to the value stored in the `xepc` register.

If the A extension is supported, the *x*RET instruction is allowed to clear any outstanding LR address reservation but is not required to. Trap handlers should explicitly clear the reservation if required (e.g., by using a dummy SC) before executing the *x*RET.

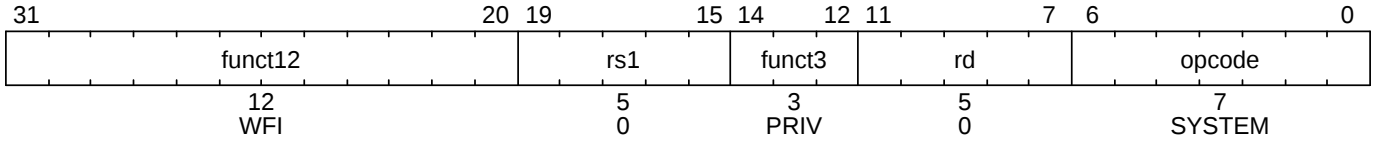
NOTE

NOTE

If *x*RET instructions always cleared LR reservations, it would be impossible to single-step through LR/SC sequences using a debugger.

3.1.3.3. Wait for Interrupt

The Wait for Interrupt instruction (WFI) informs the implementation that the current hart can be stalled until an interrupt might need servicing. Execution of the WFI instruction can also be used to inform the hardware platform that suitable interrupts should preferentially be routed to this hart. WFI is available in all privileged modes, and optionally available to U-mode. This instruction may raise an illegal-instruction exception when TW=1 in `mstatus`, as described in 3.1.1.6.6. Virtualization Support in `mstatus` Register.



If an enabled interrupt is present or later becomes present while the hart is stalled, the interrupt trap will be taken on the following instruction, i.e., execution resumes in the trap handler and `mepc` = `pc` + 4.

NOTE

The following instruction takes the interrupt trap so that a simple return from the trap handler will execute code after the WFI instruction.

Implementations are permitted to resume execution for any reason, even if an enabled interrupt has not become pending. Hence, a legal implementation is to simply implement the WFI instruction as a NOP.

NOTE

If the implementation does not stall the hart on execution of the instruction, then the interrupt will be taken on some instruction in the idle loop containing the WFI, and on a simple return from the handler, the idle loop will resume execution.

The WFI instruction can also be executed when interrupts are disabled. The operation of WFI must be unaffected by the global interrupt bits in `mstatus` (MIE and SIE) and the delegation register `mideleg` (i.e., the hart must resume if a locally enabled interrupt becomes pending, even if it has been delegated to a less-privileged mode), but should honor the individual interrupt enables (e.g, MTIE) (i.e., implementations should avoid resuming the hart if the interrupt is pending but not individually enabled). WFI is also required to resume execution for locally enabled interrupts pending at any privilege level, regardless of the global interrupt enable at each privilege level.

If the event that causes the hart to resume execution does not cause an interrupt to be taken, execution will resume at `pc` + 4, and software must determine what action to take, including looping back to

repeat the WFI if there was no actionable event.

NOTE

By allowing wake-up when interrupts are disabled, an alternate entry point to an interrupt handler can be called that does not require saving the current context, as the current context can be saved or discarded before the WFI is executed.

As implementations are free to implement WFI as a NOP, software must explicitly check for any relevant pending but disabled interrupts in the code following an WFI, and should loop back to the WFI if no suitable interrupt was detected. The `mip` or `sip` registers can be interrogated to determine the presence of any interrupt in machine or supervisor mode respectively.

The operation of WFI is unaffected by the delegation register settings.

WFI is defined so that an implementation can trap into a higher privilege mode, either immediately on encountering the WFI or after some interval to initiate a machine-mode transition to a lower power state, for example.

The same "wait-for-event" template might be used for possible future extensions that wait on memory locations changing, or message arrival.

3.1.3.4. Custom SYSTEM Instructions

The subspace of the SYSTEM major opcode shown in SYSTEM instruction encodings designated for custom use. is designated for custom use. It is recommended that these instructions use bits 29:28 to designate the minimum required privilege mode, as do other SYSTEM instructions.

31 26	25 15	14 12	11 7	6 0	
funct6	custom	funct3	custom	opcode	Recommended Purpose
6	11	3	5	7	
100011	*custom*	0	*custom*	SYSTEM	Unprivileged or User-Level
110011	*custom*	0	*custom*	SYSTEM	Unprivileged or User-Level
100111	*custom*	0	*custom*	SYSTEM	Supervisor-Level
110111	*custom*	0	*custom*	SYSTEM	Supervisor-Level
101011	*custom*	0	*custom*	SYSTEM	HS-Level
111011	*custom*	0	*custom*	SYSTEM	HS-Level
101111	*custom*	0	*custom*	SYSTEM	Machine-Level
111111	*custom*	0	*custom*	SYSTEM	Machine-Level

Figure 28. SYSTEM instruction encodings designated for custom use.

3.1.4. Reset

Upon reset, a hart's privilege mode is set to M. The `mstatus` fields MIE and MPRV are reset to 0. If little-endian memory accesses are supported, the `mstatus`/`mstatush` field MBE is reset to 0. The `misa` register is reset to enable the maximal set of supported extensions, as described in 3.1.1.1.

Machine ISA (`misa`) Register. For implementations with the "A" standard extension, there is no valid load reservation. The `pc` is set to an implementation-defined reset vector. The `mcause` register is set to a value indicating the cause of the reset. Writable PMP registers' A and L fields are set to 0, unless the platform mandates a different reset value for some PMP registers' A and L fields. If the hypervisor extension is implemented, the `hgatp`.MODE and `vsatp`.MODE fields are reset to 0. If the Smrnmi extension is implemented, the `mnstatus`.NMIE field is reset to 0. No **WARL** field contains an illegal value. If the Zicfilp extension is implemented, the `mseccfg`.MLPE field is reset to 0. All other hart state is UNSPECIFIED.

The `MML`, `MMWP`, and `RLB` fields of the `mseccfg` register are set to 0, unless the platform mandates a different reset value.

The `mcause` values after reset have implementation-specific interpretation, but the value 0 should be returned on implementations that do not distinguish different reset conditions. Implementations that distinguish different reset conditions should only use 0 to indicate the most complete reset.

The `USEED` and `SSEED` fields of the `mseccfg` CSR must have defined reset values. The system must not allow them to be in an undefined state after reset.

NOTE

Some designs may have multiple causes of reset (e.g., power-on reset, external hard reset, brownout detected, watchdog timer elapse, sleep-mode wake-up), which machine-mode software and debuggers may wish to distinguish.

To avoid ambiguity, `mcause` reset values may alias `mcause` values following synchronous exceptions. There should be no ambiguity in this overlap, since on reset the `pc` is typically set to a different value than on other traps.

3.1.5. Non-Maskable Interrupts

Non-maskable interrupts (NMIs) are only used for hardware error conditions, and cause an immediate jump to an implementation-defined NMI vector running in M-mode regardless of the state of a hart's interrupt enable bits. The `mepc` register is written with the virtual address of the instruction that was interrupted, and `mcause` is set to a value indicating the source of the NMI. The NMI can thus overwrite state in an active machine-mode interrupt handler.

The values written to `mcause` on an NMI are implementation-defined. The high Interrupt bit of `mcause` should be set to indicate that this was an interrupt. An Exception Code of 0 is reserved to mean "unknown cause" and implementations that do not distinguish sources of NMIs via the `mcause` register should return 0 in the Exception Code.

Unlike resets, NMIs do not reset processor state, enabling diagnosis, reporting, and possible containment of the hardware error.

3.1.6. Physical Memory Attributes

The physical memory map for a complete system includes various address ranges, some corresponding to memory regions and some to memory-mapped control registers, portions of which might not be accessible. Some memory regions might not support reads, writes, or execution; some might not support subword or subblock accesses; some might not support atomic operations; and some might not support cache coherence or might have different memory models. Similarly, memory-mapped control registers vary in their supported access widths, support for atomic operations, and whether read and write accesses have associated side effects. In RISC-V systems, these properties and capabilities of each region of the machine's physical address space are termed *physical memory attributes* (PMAs). This section describes RISC-V PMA terminology and how RISC-V systems implement and check PMAs.

PMAs are inherent properties of the underlying hardware and rarely change during system operation. Unlike physical memory protection values described in 3.1.7. Physical Memory Protection, PMAs do not vary by execution context. The PMAs of some memory regions are fixed at chip design time—for example, for an on-chip ROM. Others are fixed at board design time, depending, for example, on which other chips are connected to off-chip buses. Off-chip buses might also support devices that could be changed on every power cycle (cold pluggable) or dynamically while the system is running (hot pluggable). Some devices might be configurable at run time to support different uses that imply different PMAs—for example, an on-chip scratchpad RAM might be cached privately by one core in one end-application, or accessed as a shared non-cached memory in another end-application.

Most systems will require that at least some PMAs are dynamically checked in hardware later in the execution pipeline after the physical address is known, as some operations will not be supported at all physical memory addresses, and some operations require knowing the current setting of a configurable PMA attribute. While many other architectures specify some PMAs in the virtual memory page tables and use the TLB to inform the pipeline of these properties, this approach injects platform-specific information into a virtualized layer and can cause system errors unless attributes are correctly initialized in each page-table entry for each physical memory region. In addition, the available page sizes might not be optimal for specifying attributes in the physical memory space, leading to address-space fragmentation and inefficient use of expensive TLB entries.

For RISC-V, we separate out specification and checking of PMAs into a separate hardware structure, the *PMA checker*. In many cases, the attributes are known at system design time for each physical address region, and can be hardwired into the PMA checker. Where the attributes are run-time configurable, platform-specific memory-mapped control registers can be provided to specify these attributes at a granularity appropriate to each region on the platform (e.g., for an on-chip SRAM that can

be flexibly divided between cacheable and uncacheable uses). PMAs are checked for any access to physical memory, including accesses that have undergone virtual to physical memory translation. To aid in system debugging, we strongly recommend that, where possible, RISC-V processors precisely trap physical memory accesses that fail PMA checks. Precisely trapped PMA violations manifest as instruction, load, or store access-fault exceptions, distinct from virtual-memory page-fault exceptions. Precise PMA traps might not always be possible, for example, when probing a legacy bus architecture that uses access failures as part of the discovery mechanism. In this case, error responses from peripheral devices will be reported as imprecise bus-error interrupts.

PMAs must also be readable by software to correctly access certain devices or to correctly configure other hardware components that access memory, such as DMA engines. As PMAs are tightly tied to a given physical platform's organization, many details are inherently platform-specific, as is the means by which software can learn the PMA values for a platform. Some devices, particularly legacy buses, do not support discovery of PMAs and so will give error responses or time out if an unsupported access is attempted. Typically, platform-specific machine-mode code will extract PMAs and ultimately present this information to higher-level less-privileged software using some standard representation.

Where platforms support dynamic reconfiguration of PMAs, an interface will be provided to set the attributes by passing requests to a machine-mode driver that can correctly reconfigure the platform. For example, switching cacheability attributes on some memory regions might involve platform-specific operations, such as cache flushes, that are available only to machine-mode.

3.1.6.1. Main Memory versus I/O Regions

The most important characterization of a given memory address range is whether it holds regular main memory or I/O devices. Regular main memory is required to have a number of properties, specified below, whereas I/O devices can have a much broader range of attributes. Memory regions that do not fit into regular main memory, for example, device scratchpad RAMs, are categorized as I/O regions.

NOTE

What previous versions of this specification termed *vacant* regions are no longer a distinct category; they are now described as I/O regions that are not accessible (i.e. lacking read, write, and execute permissions). Main memory regions that are not accessible are also allowed.

3.1.6.2. Supported Access Type PMAs

Access types specify which access widths, from 8-bit byte to long multi-word burst, are supported, and also whether misaligned accesses are supported for each access width.

NOTE

NOTE

Although software running on a RISC-V hart cannot directly generate bursts to memory, software might have to program DMA engines to access I/O devices and might therefore need to know which access sizes are supported.

Main memory regions always support read and write of all access widths required by the attached devices, and can specify whether instruction fetch is supported.

NOTE

Some platforms might mandate that all of main memory support instruction fetch. Other platforms might prohibit instruction fetch from some main memory regions.

In some cases, the design of a processor or device accessing main memory might support other widths, but must be able to function with the types supported by the main memory.

I/O regions can specify which combinations of read, write, or execute accesses to which data widths are supported.

For systems with page-based virtual memory, I/O and memory regions can specify which combinations of hardware page-table reads and hardware page-table writes are supported.

NOTE

Unix-like operating systems generally require that all of cacheable main memory supports page-table walks.

3.1.6.3. Atomicity PMAs

Atomicity PMAs describes which atomic instructions are supported in this address region. Support for atomic instructions is divided into two categories: *LR/SC* and *AMOs*.

NOTE

Some platforms might mandate that all of cacheable main memory support all atomic operations required by the attached processors.

3.1.6.3.1. AMO PMA

Within AMOs, there are four levels of support: *AMONone*, *AMOSwap*, *AMOLogical*, and *AMOArithmetic*. AMONone indicates that no AMO operations are supported. AMOSwap indicates that only `amoswap` instructions are supported in this address range. AMOLogical indicates that swap instructions plus all the logical AMOs (`amoand`, `amoor`, `amoxor`) are supported. AMOArithmetic indicates that all RISC-V AMOs defined by the A extension are supported. For each level of support, naturally aligned AMOs of a given width are supported if the underlying memory region supports reads and writes of that width. Main

memory and I/O regions may only support a subset or none of the processor-supported atomic operations.

Table 12. Classes of AMOs supported by I/O regions.

AMO Class	Supported Operations
AMONone	*None*
AMOSwap	`amoswap`
AMOLogical	above + `amoand`, `amoor`, `amoxor`
AMOArithmetic	above + `amoadd`, `amomin`, `amomax`, `amominu`, `amomaxu`

NOTE

We recommend providing at least AMOLogical support for I/O regions where possible.

The Zacas extension defines three additional levels of support: AMOCASW, AMOCASD, and AMOCASQ.

AMOCASW indicates that in addition to instructions indicated by AMOArithmetic level support, the `AMOCAS.W` instruction is supported. AMOCASD indicates that in addition to instructions indicated by AMOCASW level support, the `AMOCAS.D` instruction is supported. AMOCASQ indicates that in addition to instructions indicated by AMOCASD level support, the `AMOCAS.Q` instruction is supported.

NOTE

`AMOCASW/D/Q` require AMOArithmetic level support as the `AMOCAS.W/D/Q` instructions require ability to perform an arithmetic comparison and a swap operation.

The AMOs specified by the Zabha extension require the same level of support as the corresponding instructions in the A standard extension or the Zacas extension.

3.1.6.3.2. Reservability PMA

For *LR/SC*, there are three levels of support indicating combinations of the reservability and eventuality properties: *RsrvNone*, *RsrvNonEventual*, and *RsrvEventual*. RsrvNone indicates that no LR/SC operations are supported (the location is non-reservable). RsrvNonEventual indicates that the operations are supported (the location is reservable), but without the eventual success guarantee described in the unprivileged ISA specification. RsrvEventual indicates that the operations are supported and provide the eventual success guarantee.

NOTE

NOTE

We recommend providing RsrvEventual support for main memory regions where possible. Most I/O regions will not support LR/SC accesses, as these are most conveniently built on top of a cache-coherence scheme, but some may support RsrvNonEventual or RsrvEventual.

When LR/SC is used for memory locations marked RsrvNonEventual, software should provide alternative fall-back mechanisms used when lack of progress is detected.

3.1.6.4. Misaligned Atomicity Granule PMA

The misaligned atomicity granule PMA provides constrained support for misaligned AMOs. This PMA, if present, specifies the size of a *misaligned atomicity granule*, a naturally aligned power-of-two number of bytes. Specific supported values for this PMA are represented by MAG*NN*, e.g., MAG16 indicates the misaligned atomicity granule is at least 16 bytes.

The misaligned atomicity granule PMA applies only to AMOs, loads and stores defined in the base ISAs, and loads and stores of no more than XLEN bits defined in the F, D, and Q extensions, and compressed encodings thereof. For an instruction in that set, if all accessed bytes lie within the same misaligned atomicity granule, the instruction will not raise an exception for reasons of address alignment, and the instruction will give rise to only one memory operation for the purposes of RVWMO—i.e., it will execute atomically.

If a misaligned AMO accesses a region that does not specify a misaligned atomicity granule PMA, or if not all accessed bytes lie within the same misaligned atomicity granule, then an exception is raised. For regular loads and stores that access such a region or for which not all accessed bytes lie within the same atomicity granule, then either an exception is raised, or the access proceeds but is not guaranteed to be atomic. Implementations may raise access-fault exceptions instead of address-misaligned exceptions for some misaligned accesses, indicating the instruction should not be emulated by a trap handler.

NOTE

LR/SC instructions are unaffected by this PMA and so always raise an exception when misaligned. Vector memory accesses are also unaffected, so might execute non-atomically even when contained within a misaligned atomicity granule. Implicit accesses are similarly unaffected by this PMA.

3.1.6.5. Memory-Ordering PMAs

Regions of the address space are classified as either *main memory* or *I/O* for the purposes of ordering by the FENCE instruction and atomic-instruction ordering bits.

Accesses by one hart to main memory regions are observable not only by other harts but also by other devices with the capability to initiate requests in the main memory system (e.g., DMA engines).

Coherent main memory regions always have either the RVWMO or RVTSO memory model. Incoherent main memory regions have an implementation-defined memory model.

Accesses by one hart to an I/O region are observable not only by other harts and bus mastering devices but also by the targeted I/O devices, and I/O regions may be accessed with either *relaxed* or *strong* ordering. Accesses to an I/O region with relaxed ordering are generally observed by other harts and bus mastering devices in a manner similar to the ordering of accesses to an RVWMO memory region, as discussed in the I/O Ordering section in the RVWMO Explanatory Material appendix of Volume I of this specification. By contrast, accesses to an I/O region with strong ordering are generally observed by other harts and bus mastering devices in program order.

Each strongly ordered I/O region specifies a numbered ordering channel, which is a mechanism by which ordering guarantees can be provided between different I/O regions. Channel 0 is used to indicate point-to-point strong ordering only, where only accesses by the hart to the single associated I/O region are strongly ordered.

Channel 1 is used to provide global strong ordering across all I/O regions. Any accesses by a hart to any I/O region associated with channel 1 can only be observed to have occurred in program order by all other harts and I/O devices, including relative to accesses made by that hart to relaxed I/O regions or strongly ordered I/O regions with different channel numbers. In other words, any access to a region in channel 1 is equivalent to executing a `fence io,io` instruction before and after the instruction.

Other larger channel numbers provide program ordering to accesses by that hart across any regions with the same channel number

Systems might support dynamic configuration of ordering properties on each memory region.

NOTE

Strong ordering can be used to improve compatibility with legacy device driver code, or to enable increased performance compared to insertion of explicit ordering instructions when the implementation is known to not reorder accesses.

Local strong ordering (channel 0) is the default form of strong ordering as it is often straightforward to provide if there is only a single in-order communication path between the hart and the I/O device.

Generally, different strongly ordered I/O regions can share the same ordering channel without additional ordering hardware if they share the same interconnect path and the path does not reorder requests.

3.1.6.6. Coherence and Cacheability PMAs

Coherence is a property defined for a single physical address, and indicates that writes to that address by one agent will eventually be made visible to other coherent agents in the system. Coherence is not to be confused with the memory consistency model of a system, which defines what values a memory

read can return given the previous history of reads and writes to the entire memory system. In RISC-V platforms, the use of hardware-incoherent regions is discouraged due to software complexity, performance, and energy impacts.

The cacheability of a memory region should not affect the software view of the region except for differences reflected in other PMAs, such as main memory versus I/O classification, memory ordering, supported accesses and atomic operations, and coherence. For this reason, we treat cacheability as a platform-level setting managed by machine-mode software only.

Where a platform supports configurable cacheability settings for a memory region, a platform-specific machine-mode routine will change the settings and flush caches if necessary, so the system is only incoherent during the transition between cacheability settings. This transitory state should not be visible to lower privilege levels.

NOTE

Coherence is straightforward to provide for a shared memory region that is not cached by any agent. The PMA for such a region would simply indicate it should not be cached in a private or shared cache.

Coherence is also straightforward for read-only regions, which can be safely cached by multiple agents without requiring a cache-coherence scheme. The PMA for this region would indicate that it can be cached, but that writes are not supported.

Some read-write regions might only be accessed by a single agent, in which case they can be cached privately by that agent without requiring a coherence scheme. The PMA for such regions would indicate they can be cached. The data can also be cached in a shared cache, as other agents should not access the region.

If an agent can cache a read-write region that is accessible by other agents, whether caching or non-caching, a cache-coherence scheme is required to avoid use of stale values. In regions lacking hardware cache coherence (hardware-incoherent regions), cache coherence can be implemented entirely in software, but software coherence schemes are notoriously difficult to implement correctly and often have severe performance impacts due to the need for conservative software-directed cache-flushing. Hardware cache-coherence schemes require more complex hardware and can impact performance due to the cache-coherence probes, but are otherwise invisible to software.

For each hardware cache-coherent region, the PMA would indicate that the region is coherent and which hardware coherence controller to use if the system has multiple coherence controllers. For some systems, the coherence controller might be an outer-level shared cache, which might itself access further outer-level cache-coherence controllers hierarchically.

Most memory regions within a platform will be coherent to software, because they will be fixed as either uncached, read-only, hardware cache-coherent, or only accessed by one agent.

If a PMA indicates non-cacheability, then accesses to that region must be satisfied by the memory itself, not by any caches.

NOTE

NOTE

For implementations with a cacheability-control mechanism, the situation may arise that a program uncacheably accesses a memory location that is currently cache-resident. In this situation, the cached copy must be ignored. This constraint is necessary to prevent more-privileged modes' speculative cache refills from affecting the behavior of less-privileged modes' uncacheable accesses.

3.1.6.7. Idempotency PMAs

Idempotency PMAs describe whether reads and writes to an address region are idempotent. Main memory regions are assumed to be idempotent. For I/O regions, idempotency on reads and writes can be specified separately (e.g., reads are idempotent but writes are not). If accesses are non-idempotent, i.e., there is potentially a side effect on any read or write access, then speculative or redundant accesses must be avoided.

For the purposes of defining the idempotency PMAs, changes in observed memory ordering created by redundant accesses are not considered a side effect.

NOTE

While hardware should always be designed to avoid speculative or redundant accesses to memory regions marked as non-idempotent, it is also necessary to ensure software or compiler optimizations do not generate spurious accesses to non-idempotent memory regions.

Non-idempotent regions might not support misaligned accesses. Misaligned accesses to such regions should raise access-fault exceptions rather than address-misaligned exceptions, indicating that software should not emulate the misaligned access using multiple smaller accesses, which could cause unexpected side effects.

For non-idempotent regions, implicit reads and writes must not be performed early or speculatively, with the following exceptions. When a non-speculative implicit read is performed, an implementation is permitted to additionally read any of the bytes within a naturally aligned power-of-2 region containing the address of the non-speculative implicit read. Furthermore, when a non-speculative instruction fetch is performed, an implementation is permitted to additionally read any of the bytes within the *next* naturally aligned power-of-2 region of the same size (with the address of the region taken modulo 2^{XLEN}). The results of these additional reads may be used to satisfy subsequent early or speculative implicit reads. The size of these naturally aligned power-of-2 regions is implementation-defined, but, for systems with page-based virtual memory, must not exceed the smallest supported page size.

3.1.7. Physical Memory Protection

To support secure processing and contain faults, it is desirable to limit the physical addresses accessible by software running on a hart. An optional physical memory protection (PMP) unit provides per-hart machine-mode control registers to allow physical memory access privileges (read, write,

execute) to be specified for each physical memory region. The PMP values are checked in parallel with the PMA checks described in 3.1.6. Physical Memory Attributes.

The granularity of PMP access control settings are platform-specific, but the standard PMP encoding supports regions as small as four bytes. Certain regions' privileges can be hardwired—for example, some regions might only ever be visible in machine mode but in no lower-privilege layers.

NOTE

Platforms vary widely in demands for physical memory protection, and some platforms may provide other PMP structures in addition to or instead of the scheme described in this section.

PMP checks are applied to all accesses whose effective privilege mode is S or U, including instruction fetches and data accesses in S and U mode, and data accesses in M-mode when the MPRV bit in `mstatus` is set and the MPP field in `mstatus` contains S or U. PMP checks are also applied to page-table accesses for virtual-address translation, for which the effective privilege mode is S. Optionally, PMP checks may additionally apply to M-mode accesses, in which case the PMP registers themselves are locked, so that even M-mode software cannot change them until the hart is reset. In effect, PMP can *grant* permissions to S and U modes, which by default have none, and can *revoke* permissions from M-mode, which by default has full permissions.

PMP violations are always trapped precisely at the processor.

3.1.7.1. Physical Memory Protection CSRs

PMP entries are described by an 8-bit configuration register and one MXLEN-bit address register. Some PMP settings additionally use the address register associated with the preceding PMP entry. Up to 64 PMP entries are supported. Implementations may implement zero, 16, or 64 PMP entries; the lowest-numbered PMP entries must be implemented first. All PMP CSR fields are **WARL** and may be read-only zero. PMP CSRs are only accessible to M-mode.

The PMP configuration registers are densely packed into CSRs to minimize context-switch time. For RV32, sixteen CSRs, `pmpcfg0` – `pmpcfg15`, hold the configurations `pmp0cfg` – `pmp63cfg` for the 64 PMP entries, as shown in RV32 PMP configuration CSR layout.. For RV64, eight even-numbered CSRs, `pmpcfg0`, `pmpcfg2`, …, `pmpcfg14`, hold the configurations for the 64 PMP entries, as shown in RV64 PMP configuration CSR layout.. For RV64, the odd-numbered configuration registers, `pmpcfg1`, `pmpcfg3`, …, `pmpcfg15`, are illegal.

NOTE

RV64 harts use `pmpcfg2`, rather than `pmpcfg1`, to hold configurations for PMP entries 8-15. This design reduces the cost of supporting multiple MXLEN values, since the configurations for PMP entries 8-11 appear in `pmpcfg2` [31:0] for both RV32 and RV64.

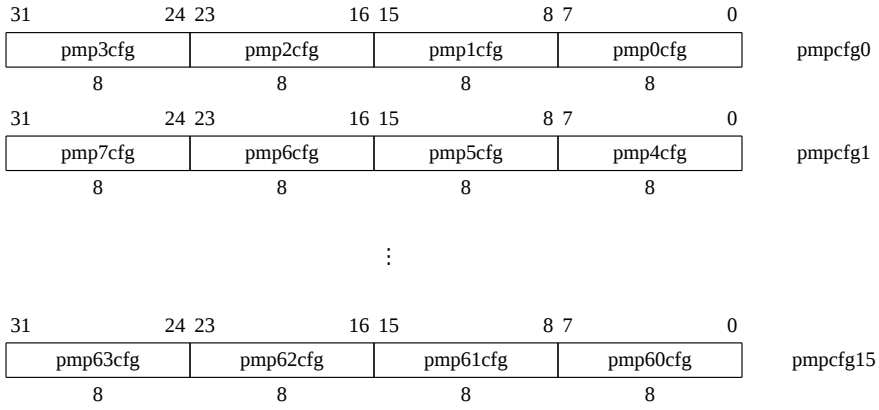


Figure 29. RV32 PMP configuration CSR layout.

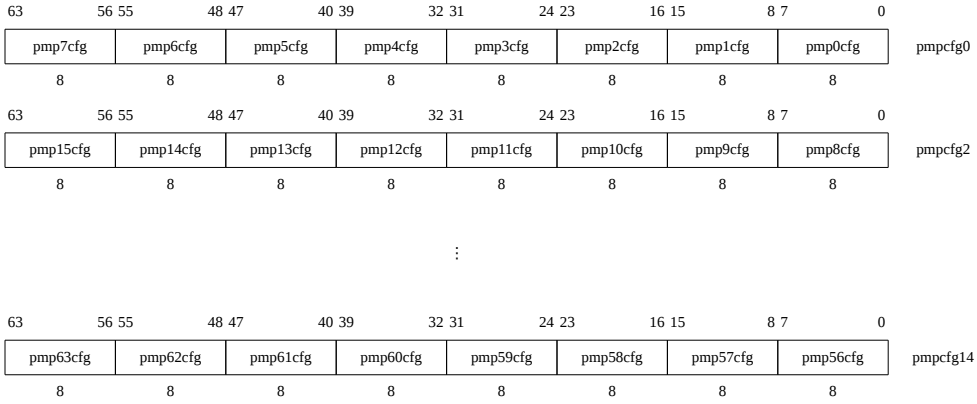


Figure 30. RV64 PMP configuration CSR layout.

The PMP address registers are CSRs named `pmpaddr0`-`pmpaddr63`. Each PMP address register encodes bits 33-2 of a 34-bit physical address for RV32, as shown in PMP address register format, RV32.. For RV64, each PMP address register encodes bits 55-2 of a 56-bit physical address, as shown in PMP address register format, RV64.. Not all physical address bits may be implemented, and so the `pmpaddr` registers are **WARL**.

NOTE

The Sv32 page-based virtual-memory scheme described in Sv32: Page-Based 32-bit Virtual-Memory Systems supports 34-bit physical addresses for RV32, so the PMP scheme must support addresses wider than XLEN for RV32. The Sv39 and Sv48 page-based virtual-memory schemes described in Sv32: Page-Based 39-bit Virtual-Memory Systems and Sv48: Page-Based 48-bit Virtual-Memory Systems support a 56-bit physical address space, so the RV64 PMP address registers impose the same limit.

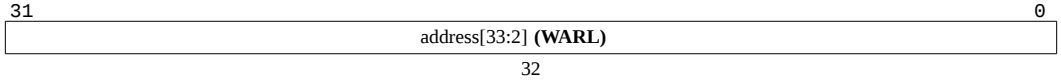


Figure 31. PMP address register format, RV32.

Figure 32. PMP address register format, RV64.

PMP configuration register format. shows the layout of a PMP configuration register. The R, W, and X bits, when set, indicate that the PMP entry permits read, write, and instruction execution, respectively. When one of these bits is clear, the corresponding access type is denied. The R, W, and X fields form a collective **WARL** field for which the combinations with R=0 and W=1 are reserved. The remaining two fields, A and L, are described in the following sections.

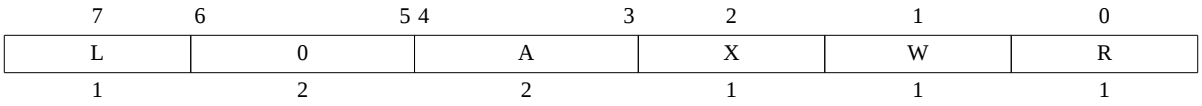


Figure 33. PMP configuration register format.

Attempting to fetch an instruction from a PMP region that does not have execute permissions raises an instruction access-fault exception. Attempting to execute a load, load-reserved, or cache-block management instruction which accesses a physical address within a PMP region without read permissions raises a load access-fault exception. Attempting to execute a store, store-conditional, AMO, or cache-block zero instruction which accesses a physical address within a PMP region without write permissions raises a store access-fault exception.

3.1.7.1.1. Address Matching

The A field in a PMP entry's configuration register encodes the address-matching mode of the associated PMP address register. The encoding of this field is shown in Encoding of A field in PMP configuration registers.. When A=0, this PMP entry is disabled and matches no addresses. Two other address-matching modes are supported: naturally aligned power-of-2 regions (NAPOT), including the special case of naturally aligned four-byte regions (NA4); and the top boundary of an arbitrary range (TOR). These modes support four-byte granularity.

Table 13. Encoding of A field in PMP configuration registers.

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥8 bytes

NAPOT ranges make use of the low-order bits of the associated address register to encode the size of the range, as shown in `NAPOT` range encoding in PMP address and configuration registers..

Table 14. NAPOT range encoding in PMP address and configuration registers.

`pmpaddr`	`pmpcfg`.A	Match type and size

`pmpaddr`	`pmpcfg`.A	Match type and size
`yyyy…yyyy`	NA4	4-byte NAPOT range
`yyyy…yyy0`	NAPOT	8-byte NAPOT range
`yyyy…yy01`	NAPOT	16-byte NAPOT range
`yyyy…y011`	NAPOT	32-byte NAPOT range
…	…	…
`yy01…1111`	NAPOT	2^{XLEN}-byte NAPOT range
`y011…1111`	NAPOT	2^{XLEN+1}-byte NAPOT range
`0111…1111`	NAPOT	2^{XLEN+2}-byte NAPOT range
`1111…1111`	NAPOT	2^{XLEN+3}-byte NAPOT range

If TOR is selected, the associated address register forms the top of the address range, and the preceding PMP address register forms the bottom of the address range. If PMP entry *i*'s A field is set to TOR, the entry matches any address *y* such that $\texttt{pmpaddr}_{i-1} \le y < \texttt{pmpaddr}_i$ (irrespective of the value of $\texttt{pmpcfg}_{i-1}$). If PMP entry 0's A field is set to TOR, zero is used for the lower bound, and so it matches any address $y < \texttt{pmpaddr}_0$.

NOTE

If $\texttt{pmpaddr}_{i-1} \ge \texttt{pmpaddr}_i$ and $\texttt{pmpcfg}_i\texttt{.A}$ =TOR, then PMP entry *i* matches no addresses.

NOTE

It is not possible to represent the address 2^{XLEN+2} as the top of a range, so, for example, in an RV32 system with 34-bit physical addresses, a TOR PMP cannot be used to give less-privileged modes access to the uppermost word of memory. Either a NAPOT PMP can be used, or that memory can be left inaccessible to less-privileged modes. If the supervisor manages memory at base-page granularity, just 0.00002% of the 16 GiB address space is lost.

Although the PMP mechanism supports regions as small as four bytes, platforms may specify coarser PMP regions. In general, the PMP grain is 2^{G+2} bytes and must be the same across all PMP regions. When G ≥ 1, the NA4 mode is not selectable. When G ≥ 2 and pmpcfg_i.A[1] is set, i.e. the mode is NAPOT, then bits pmpaddr_i[G-2:0] read as all ones. When G ≥ 1 and pmpcfg_i.A[1] is clear, i.e. the mode is OFF or TOR, then bits pmpaddr_i[G-1:0] read as all zeros. Bits pmpaddr_i[G-1:0] do not affect the TOR address-matching logic. Although changing pmpcfg_i.A[1] affects the value read from pmpaddr_i, it does not affect the underlying value stored in that register—in particular, pmpaddr_i[G-1] retains its original value when pmpcfg_i.A is changed from NAPOT to TOR/OFF then back to NAPOT.

NOTE

Software may determine the PMP granularity by writing zero to `pmp0cfg`, then writing all ones to `pmpaddr0`, then reading back `pmpaddr0`. If *G* is the index of the least-significant bit set, the PMP granularity is 2^{G+2} bytes.

3.1.7.1.2. Locking and Privilege Mode

The L bit indicates that the PMP entry is locked, i.e., writes to the configuration register and associated address registers are ignored. Locked PMP entries remain locked until the hart is reset. If PMP entry *i* is locked, writes to `pmp`*i*`cfg` and `pmpaddr`*i* are ignored. Additionally, if PMP entry *i* is locked and `pmp`*i*`cfg.A` is set to TOR, writes to `pmpaddr`*i*-1 are ignored.

NOTE

Setting the L bit locks the PMP entry even when the A field is set to OFF.

In addition to locking the PMP entry, the L bit indicates whether the R/W/X permissions are additionally enforced on M-mode accesses. When the L bit is set, these permissions are enforced for all privilege modes. When the L bit is clear, any M-mode access matching the PMP entry will succeed; the R/W/X permissions apply only to S and U modes.

3.1.7.1.3. Priority and Matching Logic

On some implementations, misaligned loads, stores, and instruction fetches may be decomposed into multiple memory operations, some of which may succeed before an access-fault exception occurs, as described in the RVWMO specification. PMP checking is performed on each memory operation independently. In particular, a portion of a misaligned store that passes the PMP check may become visible, even if another portion fails the PMP check. The same behavior may manifest for stores wider than XLEN bits (e.g., the FSD instruction in RV32D), even when the store address is naturally aligned.

PMP entries are statically prioritized. The lowest-numbered PMP entry that matches any byte of a memory operation determines whether that operation succeeds or fails. The matching PMP entry must match all bytes of a memory operation, or the operation fails, irrespective of the L, R, W, and X bits. For example, if a PMP entry is configured to match the four-byte range `0xC`–`0xF`, then an 8-byte access to the range `0x8`–`0xF` will fail, assuming that PMP entry is the highest-priority entry that matches those addresses.

If a PMP entry matches all bytes of a memory operation, then the L, R, W, and X bits determine whether the operation succeeds or fails. If the L bit is clear and the privilege mode of the access is M, the operation succeeds. Otherwise, if the L bit is set or the privilege mode of the access is S or U, then the operation succeeds only if the R, W, or X bit corresponding to the access type is set.

If no PMP entry matches an M-mode memory operation, the operation succeeds. If no PMP entry matches an S-mode or U-mode memory operation, but at least one PMP entry is implemented, the operation fails.

NOTE

If at least one PMP entry is implemented, but all PMP entries' A fields are set to OFF, then all S-mode and U-mode memory accesses will fail.

Failed memory operations generate an instruction, load, or store access-fault exception. Note that a single instruction may generate multiple memory operations, which may not be mutually atomic. An access-fault exception is generated if at least one memory operation generated by an instruction fails, though other memory operations generated by that instruction may succeed with visible side effects. Notably, instructions that reference virtual memory are decomposed into multiple memory operations.

3.1.7.2. Physical Memory Protection and Paging

The Physical Memory Protection mechanism is designed to compose with the page-based virtual memory systems described in Supervisor. When paging is enabled, instructions that access virtual memory may result in multiple physical-memory accesses, including implicit references to the page tables. The PMP checks apply to all of these accesses. The effective privilege mode for implicit page-table accesses is S.

Implementations with virtual memory are permitted to perform address translations speculatively and earlier than required by an explicit memory access, and are permitted to cache them in address translation cache structures—including possibly caching the identity mappings from effective address to physical address used in Bare translation modes and M-mode. The PMP settings for the resulting physical address may be checked (and possibly cached) at any point between the address translation and the explicit memory access. Hence, when the PMP settings are modified, M-mode software must synchronize the PMP settings with the virtual memory system and any PMP or address-translation caches. This is accomplished by executing an SFENCE.VMA instruction with *rs1*= `x0` and *rs2*= `x0`, after the PMP CSRs are written. See Supervisor Memory-Management Fence Instruction for additional synchronization requirements when the hypervisor extension is implemented.

If page-based virtual memory is not implemented, memory accesses check the PMP settings synchronously, so no SFENCE.VMA is needed.