WorldGuard Specification

Table of Contents

This is a proposal for an RVIA standard. It is only in inception stage and has not been accepted as the basis of an official task group. Everything might change before standardization.

History

Revision	date	Comments
0.3	2023-04-03	Donated to RVIA
0.3-draft	2022-11-22	Third draft in progress.
0.2	2022-11-20	Second draft, for donation to RVIA
0.1	2022-10-31	Initial draft, from WorldGuard HAS Specification 1.1

Release Notes

- 0.3
  - This is version 0.3, which is being donated to RVIA as a proposal to form a task group.

Term	Meaning
CSR	Control and Status Register
MMU	Memory Management Unit
Privilege modes	The RISC-V privilege specification defines up to five privilege modes: M, [H]S, U, VS, VU
REE	Rich Execution Environment
TEE	Trusted Execution Environment
WID	WorldGuard world identifier
WG	WorldGuard
`XLEN`	Refers to the width of an integer register in bits (either 32 or 64)

1. WorldGuard Overview

WorldGuard (*WG*) provides isolation in a hardware platform by constraining access to system physical addresses. WG provides *Worlds* , which are execution contexts that include agents (such as harts and devices) that can initiate a transaction on a physical address within a world, and resources (such as memories and peripheral devices) that respond to transactions at a physical address within a world. Worlds are created and configured by a trusted execution environment, usually at system boot time.

Worlds are uniquely identified by a hardware *World Identifier* (*WID*) and the maximum number of unique WIDs on a platform is *NWorlds*. Increasing NWorlds increases the hardware cost, and in practice, 2-8 WIDs (requiring 1-3 bits to represent) are sufficient for many use cases.

Hardware agents (harts and devices) that can initiate transactions on physical addresses may support multiple contexts, with each context potentially being in a different world. A software context running on a hart agent is present in one world at a time. A hardware context on a device agent is present in one world at a time.

Resources are identified by system physical addresses. A world may be granted read, write, or both, access permissions on a physical address. A resource can optionally be shared between worlds, with independent access permissions for each world.

WG is designed for the case where the allocation of agent contexts and resources to worlds is performed before or at reset/boot time, and not changed dynamically when the system is running unless there is a system reset. Efficiently changing WG configurations dynamically while the system is running is not a goal of the current specification.

When an agent context initiates a transaction to a physical address, hardware marks the transaction with the WID of the agent context. The transaction is only allowed to complete successfully if the targeted resource has the appropriate access permissions (read or write) on that address for the WID on the transaction. The permission checks might be performed at the agent, at

the resource, or anywhere along the path the transaction takes through the platform's bus hierarchy. The bus transaction carries the WID through the interconnect and all elements on the path toward the targeted resource until access permissions can be checked. The method of propagating and checking the WID on busses is platform-specific, with different bus fabrics supporting WIDs in bus-specific ways.

Theoretically all permissions checks on transactions could be performed at the source agent to prevent any illegal transactions from entering the bus fabric. But in practice, replicating and checking the entire platform permissions map at each agent is prohibitively expensive, particularly when permissions are configurable, and so WG assumes permissions checking is distributed out in the bus fabric and attached resources.

If the permission check fails, the transaction is terminated or modified to avoid violating world isolation and the failure may be reported. Failures may be reported in a number of ways depending on the platform, the agent, the resource, and the transaction type. Failures may be reported to the initiating agent, and optionally one or more other agents. In some cases, the failure cannot be directly reported to the initiating agent and the transaction is modified to be ignored or to return benign data. In these cases, the failure may still be reported to a different agent through an alternate mechanism.

2. RISC-V ISA WorldGuard Extensions

RISC-V harts that support WorldGuard associate a WID with all memory accesses initiated by that hart. The WorldGuard extensions allow different privilege modes on a hart to be tagged with different WIDs.

There are three levels of WG support on RISC-V harts. The first level does not require an ISA extension and fixes the WID for all privilege modes on a hart. The second level is the **Smwg** extension, which enables M-mode to control the WID of lower-privilege modes. The third level is the **Smwgd** extension, which further enables M-mode to delegate to [H]S-mode the ability to assign the WID of lower-privilege modes, thereby adding the **Sswg** extension to [H]S-mode.

All accesses, including implicit memory references such as instruction fetches and page-table walks, must be tagged with the appropriate WID. For the purposes of WG permissions, instruction fetches are treated as memory reads.

Bus fabrics typically do not differentiate instruction fetches from memory reads, and so WG permissions checkers located on the other side of the bus fabric are unable to distinguish these two cases.

WorldGuard does not allow a privilege mode to change its own WID. The WID of M-mode on a hart is set by the external environment and does not change between resets.

Different harts in a system may have different WIDs in M-mode.

2.1. WorldGuard CSRs

The WorldGuard Smwg, Smwgd, and Sswg extensions allow a hart to assign WIDs to its privilege modes. The new CSRs are listed in the Table below.

Table 1. WorldGuard CSRs

Size in bits	Register	Access	Proposed CSR Address	Description
XLEN	mlwid	RW for M	0x390	WID used for lower privilege modes. Ceil(Log$_2$NWorlds) LSBs are used, others are zero.
XLEN	mwiddeleg	RW for M	0x748	Set of WID values delegated to [H]S-mode, represented as a bit vector. NWorlds LSBs are used, others are zero.
XLEN	slwid	RW for [H]S	0x190	WID value used in lower modes (i.e., U, VS, or VU). Ceil(Log$_2$NWorlds) LSBs are used, others are zero.

These extensions supports NWorlds ≤ XLEN.

2.2. One world per hart

In this case, there are no ISA-visible additions to the RISC-V hart. The hart is reset into a single world and all transactions from that hart, regardless of privilege mode, are tagged with the WID of that world. How the hart is assigned to a world, or whether and how a hart is allowed to determine any information about WG configuration is platform-specific.

2.3. Smwg extension

The Smwg extension adds support for M-mode to control the world used by less-privileged modes, and can only be added to harts with at least two privilege modes.

The Smwg extension adds the mlwid CSR, which is an M-mode read-write CSR, whose least-significant bits set the WID to be used by lower-privilege modes.

If the system supports demand-paged virtual memory, then any address-translation caches must ensure that translations are cached separately for each WID. A simple implementation can flush address-translation caches on any mlwid write.

The mlwid CSR is WARL, and if an illegal WID is written, the lowest-numbered legal WID is returned. It is platform-specific which worlds can be used by lower-privilege modes on this hart.

M-mode software can use the WARL property of mlwid to discover which worlds are available to be assigned to lower modes, though in normal use, platform software will have predetermined allocations for the worlds on a platform.

We constrain WARL behavior to reduce compatibilty-testing effort. Also, having a defined value slightly reduces time to dynamically search for valid WIDs.

M-mode may have a different WID than any assignable by `mlwid` to lower-privilege modes.

The platform is not required to allow lower-privilege modes to be in the same world as M-mode.

At reset, `mlwid` must hold a WID that can be allocated to lower modes.

2.4. Smwgd / Sswg extensions

The Smwgd extension requires the Smwg extension and allows M-mode to delegate to [H]S-mode the ability to allocate WIDs to privilege modes lower than [H]S. The Smwgd extension optionally enables the Sswg extension for [H]S-mode.

The Smwgd extension adds the `mwiddeleg` M-mode read-write CSR. The `mwiddeleg` register represents a set of WIDs as a bit vector with WID *i* represented by bit *i* of the register. The `mwiddeleg` CSR is a WARL register where each bit that can be set indicates a WID that is delegated to [H]S-mode. The set of worlds that can be delegated to [H]S-mode on a hart is platform-specific.

Different harts on a platform can have different sets of delegatable worlds.

The Sswg extension adds the [H]S-mode read-write `slwid` CSR, which sets the WID used for modes lower than [H]S-mode. The `slwid` is WARL, with `mwiddeleg` specifying the legal values for `slwid`. If an illegal WID is written to `slwid`, the lowest-numbered legal WID is returned. When Sswg is present, `mlwid` now specifies the WID for [H]S-mode only, and `slwid` specifies the WID for lower-privilege modes (U, VS, VU).

If the value in `mwiddeleg` is non-zero, the Sswg extension is enabled. When `mwiddeleg` is set to a non-zero value, `slwid` is initialized to the lowest-numbered world present in `mwiddeleg`.

If the value in `mwiddeleg` is zero, then the Sswg extension is disabled and accesses to `slwid` raise an illegal instruction exception.

If the system supports demand-paged virtual memory, then any address-translation caches must ensure that translations are cached separately for each WID. A simple implementation can flush address-translation caches on any write to `mwiddeleg` or `slwid`.

When the hypervisor extension is present `slwid` sets the WID for both VS and VU mode, as well as U mode.

The Sswg extension is not available to a guest OS.

There is no requirement for `mwiddeleg` to contain the WID in `mlwid`, i.e., S-mode can be set to a different world than the ones it is allowed to assign to lower modes using `slwid` when Sswg is enabled.

At reset, `mwiddeleg` is set to zero and hence Sswg is disabled.

2.5. Response to permission violations

When a hart attempts an explicit or implicit memory access that fails a WG permissions check, the access may or may not raise an access-fault exception of the appropriate type (i.e., instruction-access fault, load-access fault, or store/AMO-access fault). When an access-fault exception cannot be raised, the instruction performing the memory access can be retired but any writes to the protected physical memory location are ignored and any memory reads return data independent of the value in the protected physical memory location to avoid violating memory isolation.

Secure systems will typically ensure that some agent is notified when an illegal access is attempted, even when an access-fault exception cannot be raised on the hart context.

We cannot require that reads that fail permissions checks but that do not raise access-fault exceptions return a specific value (e.g, zero) to the hart as this is incompatible with some cache-coherence protocols, which may require cache-resident data be modifiable even when the underlying physical memory locations are protected and the bus responses previously returned zero.

3. Non-ISA WorldGuard Hardware Platform Components

The overall structure of a hardware platform supporting WorldGuard is platform-specific. A hardware platform may include *markers* that add WIDs to hardware-agent accesses and *checkers* that check access permissions at some location in the path to resources. All accesses targeting a WG-protected resource must be associated with a WID.

The design of WG markers and checkers may be customized for particular device agents and resources respectively.

One of the motivations for WG is to reduce the cost of securing platform-level accesses by customizing WID checks at the resources instead of requiring that all information is handled in a generic form at the agents.

The configuration of the WG markers and WG checkers can be fixed at design time or be controlled by writable state. Where markers and checkers have dynamic configuration state, it is recommended that the state is reset to a known platform-specific configuration or to a safe state that prevents unauthorized accesses. Dynamic configuration state is typically initialized by trusted boot-time software. A mechanism should be provided to lock this state after initialization until the next reset.

Typically, assignment of agents and resources to worlds is performed once at boot time and locking the configurations provides additional security.

3.1. Generic WG Checker

Although WG checkers can be customized for individual resource types on a platform, a generic interface to a checker is provided here. This design can be used directly for applications requiring flexible world configuration over a large resource address range, and can also be used as a basis for more customized checker designs.

The generic checker monitors a fixed physical address range and allows for software configuration of permissions within that address range. A number of programmable *slots* are provided within the checker. Each slot can specify a *rule* for a contiguous subset of addresses within the checker's overall memory address range, giving the read and write permissions for each world for that address subset.

This section describes only the software interface to such a checker.

Full implementation specifications will require many further platform-specific details on bus interfaces, reset, clocking, and power management, for example.

3.1.1. Configuration Register Memory Map

The configuration registers for the generic checker are memory-mapped into a separate region of physical memory addresses, which is usually protected such that only trusted software can alter the configuration at boot time.

The memory-mapped interface is designed to be operated using only individual 32b accesses.

Table 2. WG Generic Checker Configuration Register

Offset	Bytes	Access	Name	Description
0x00	4	R	`vendor`	Vendor ID
0x04	4	R	`impid`	Implementation revision
0x08	4	R	`nslots`	Number of rule slots
0x0C	4			Reserved
0x10	8	RW	`errcause`	Information about a permissions violation
0x18	8	RW	`erraddr`	Address of a permissions violation
0x20	(nslots+1) *32	RW	`slot[nslots:0]`	Array of slots

The read-only `vendor` and `impid` registers provide details on the vendor and implementation version of the checker.

The read-only `nslots` parameter is encoded as a 4-byte unsigned integer which gives the number of variable rule slots available in this checker. The value of `nslots` must be ≥ 1. The read-only `slot[0]` is not counted in the `nslots` value.

The `errcause` and `erraddr` registers are used to report permissions violations, as described below.

3.1.2. Rule Slot Format

Each slot defines one rule and occupies 32 bytes of address space formatted as follows:

Table 3. WG Generic Checker Slot Configuration (32 bytes total per slot)

Offset	Bytes	Name	Description
0x00	4	`addr[33:2]`	Rule address
0x04	4	`addr[65:34]`	Rule address (RV64 systems only, zero on RV32)
0x08	8	`perm[nWorlds-1:0]`	R and W permissions for up to 32 worlds
0x10	4	`cfg`	Rule configuration
0x14	12		Reserved

The `addr[65:2]` register holds a single physical address for the rule, shifted right by two bits to allow the first 32b register to represent the 34-bit physical address range of an Sv32 system. Address registers may be constrained (WARL) to only hold addresses naturally aligned to a certain protection granule, e.g., 4KiB.

The `perm` register holds two bits per WID that give the WID's read and write permissions for the rule. Bit 2**i* holds the read permission for WID *i*, and bit 2**i*+1 holds the write permission for WID *i*.

The `cfg` register specifies the behavior of the rule, and is formatted into fields as follows:

Table 4. WG Rule Configuration Register

Bits	Name	Description
1:0	`A[1:0]`	Address range configuration
7:2		Reserved (write zero)
8	`ER`	Report read violations as bus errors
9	`EW`	Report write violations as bus errors
10	`IR`	Report read violations as interrupts
11	`IW`	Report write violations as interrupts
30:12		Reserved (write zero)
31	`L`	Lock bit

The `A[1:0]` field specifies how the address range of the rule is constructed, according to the following table:

Table 5. WG A[1:0] encoding

A[1:0]	Name	Description
0	OFF	Rule disabled (grants no access permissions)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region (optional)

A[1:0]	Name	Description
3	NAPOT	Naturally aligned power-of-two region ≥ 8 bytes (optional)

When A=OFF, the rule will grant no access permissions.

When A=TOR, the rule's addr register specifies the top of the rule's address range and the preceding slot's addr register gives the bottom of the rule's address range, when the preceding slot's cfg field is set to OFF or TOR. A TOR rule matches an address y where:

```
slot[i-1].addr <= y < slot[i].addr # slot[i-1].cfg = OFF or TOR
```

The A=NA4 and A=NAPOT encodings are optional. If provided, they are based on the format of the RISC-V PMP encoding, and allow a single addr register to encode a base address and the size of a naturally aligned power-of-2 (NAPOT) region.

The figure below shows how the NAPOT range encoding differs from RISC-V PMPs. The checker typically monitors a fixed NAPOT subset of the entire physical address range. The address bits marked with a b represent the non-writeable bits in addr, which hold the base address of the checker region. The writable address bits in the addr registers are marked as a.

Table 6. WG NAPOT range encoding for a checker covering a NAPOT total address range

addr[65:2]	A[1:0]	**Match type and size**
bb···bb000···0000		*Address of first byte in checker range* (slot[0].addr)
bb···bbaaa···aaaa	NA4	4-byte NAPOT range
bb···bbaaa···aaa0	NAPOT	8-byte NAPOT range
bb···bbaaa···aa01	NAPOT	16-byte NAPOT range
bb···bbaaa···a011	NAPOT	32-byte NAPOT range
...	...	...
bb···bba01···1111	NAPOT	half of checker range
bb···bb011···1111	NAPOT	all of checker range
bb···bb111···1111	NAPOT	all of checker range
bb···bb111···1111		*Address of last four-byte word in checker range* (slot[nslots].addr - 4)

For the A=NA4 configuration , the addr value (composed of fixed b bits and variable a bits) holds the base address of the four-byte NAPOT range.

For the A=NAPOT configuration, a run of contiguous 1 bits in the least-significant addr bits sets the size of the slot's NAPOT region, with the least-significant 0 bit representing the most-significant bit that is not part of the slot's base address (e.g., addr[3] is zero for a 16-byte NAPOT region). The addr bits above the least-significant zero bit (both fixed b bits and variable a bits) set the base address of the slot's NAPOT region. When the highest writable a bit is set to 0, the slot's NAPOT range covers the

entire address range monitored by the checker. When all the writable `a` bits are set to 1, the range is also the entire checker's address range, i.e., the setting of the highest `a` bit is a "don't-care" under a NAPOT encoding when all lower bits are set to 1.

The NAPOT thermometer encoding of the masked address bits was designed to simplify the logic needed to perform the NAPOT address comparison.

In another departure from the RISC-V PMP encoding, when a NAPOT slot is being used as the base address for a TOR configuration in the next-highest numbered slot, the address used in the comparison is one greater than the highest byte address in the NAPOT slot's region, i.e., a TOR rule matches an address `y` where:

```
slot[i-1].addr      <= y < slot[i].addr # slot[i-1].cfg = OFF or TOR
slot[i-1].napot_top <= y < slot[i].addr # slot[i-1].cfg = NA4 or NAPOT

where napot_top is one greater than address of the highest-addressed byte in slot[i-
1]'s NAPOT range
```

Compared to the standardized RISC-V PMP encoding, this optimization reduces by one the number of slots needed to represent a variable-sized region following a NAPOT region in the address map. No significant additional comparator logic is required over the standard PMP design.

The IR and IW bits indicate whether an interrupt should be generated if an access fails global permissions checks but falls within the address range of the rule. The IR bit specifies the response to read transactions, while the IW bit specifies the response to write transactions. If the IR or IW bit is clear, then no interrupts are reported. If the IR or IW bits are set, then interrupts are reported for permission violations for this rule.

The ER and EW bits indicates whether bus-error responses should be returned for bus transactions that fail global permissions checks but fall within the address range of rule. The ER bit specifies the response to read transactions, while the EW bit specifies the response to write transactions. If the ER is set, reads return zero data but also an error response, and if the EW bit is set, writes are ignored but return an error response. The type and encoding of the error responses is bus protocol-specific. If the ER bit is clear, reads return zero data but the transaction response does not indicate an error. If the EW bit is clear, writes are ignored, and the transaction response does not indicate an error.

The IR, IW, ER, and EW bits are used to tailor the checker's behavior for memory regions that are cacheable, idempotent, or non-idempotent.

The L bit indicates that this entry is locked. Once this is set, the slot data cannot be modified or unlocked without a reset of the checker. Note that when two slots are used to specify a range, both slots must be individually locked to prevent a change in the range.

The other bits in the `cfg` register are reserved for future use and should be written as zeros.

3.1.3. Address Range Bounds

The first slot, `slot[0]`, is a special rule slot that follows the same format as other slots but where the `addr` field is set to the address of the first byte of the checker's address range and where `cfg.A[1:0]` is hard-wired to OFF. Slot[0] never provides permissions for an access (`perms[]` are all zero) and the `addr` field is only used to delimit the start of the checker's address range. The IR, IW, ER, EW and L bits are implemented in the `slot[0].cfg` register but are used to determine the interrupt and error response to transactions that do not overlap with any other rule's address range, with the L bit locking this configuration until the next reset.

The last slot `slot[nslots]` has a read-only `addr` field that holds the address of the first byte past the end of the checker's address range. The last slot's `cfg` field can only be set to OFF or TOR. Otherwise the last slot can be configured as with the other writeable slots in the checker.

The smallest possible generic checker with `nslots=1` has only this one writable rule `slot[1]` together with `slot[0]`, and when set to `cfg=TOR` this controls world permissions for the entire checker address range.

When an attempt is made to write an `addr` register with a physical address `y` outside the range of the checker (i.e., `y < slot[0].addr OR y >= slot[nslots].addr`), the corresponding `addr` field is set to the bottom of the checker's range (i.e., `slot[0].addr`).

Storing only addresses within the range of the checker reduces hardware requirements. In normal usage, the checker would be placed in the platform's bus fabric such that only accesses within the range would be presented to the checker.

There is no error reporting if an invalid configuration is written. The software setting up the configuration should read back the values to check they were set correctly if this is a concern.

3.1.4. Error-reporting registers

The checker has two error-reporting registers `errcause` and `erraddr` that are written when an access violation is reported. The `erraddr` register holds the failing physical address shifted right by 2 bits (`addr[65:2]`) to allow the lower 32b register to hold the full 34b physical address space in an RV32 system.

The `errcause` register is formatted as shown in the Table below.

Table 7. WG Error cause register `errcause` format

Bits	Name	Description
7:0	`wid`	The WID that cause the error
8	`r`	If set, a read caused the error
9	`w`	If set, a write cause the error
31:10		Reserved (write zero)
61:32		Reserved (write zero)

Bits	Name	Description
62	be	Bus error generated
63	ip	Interrupt generated

The wid, r, and w fields record the WID and access type of the transaction that caused the error.

The be bit is set if the violation caused a bus error to be reported, cleared otherwise.

The ip bit is set when the violation raises an interrupt.

The ip bit provides a level-sensitive interrupt signal that can be fed to a platform-specific interrupt controller.

Both be and ip bits can be set by the same violation as detailed below.

With either of the be or ip bits is set, further violations do not update the errcause or erraddr registers.

The be and ip bits must be cleared by software to reenable further error reports and new interrupts.

Usually, a software handler will read errcause and erraddr fields before clearing errcause.

Generation of bus-error responses is independent of the errcause value, so bus-error responses will continue to be generated. But new interrupts cannot be generated until the be and ip fields are cleared.

3.1.5. Operation of the Checker

When an access arrives at the checker, each rule is independently evaluated to determine if the access is within the rule's address range and if so, whether the rule grants the WID the necessary read or write permissions.

The checks are done in parallel and combined, which is different from RISC-V PMP checks, which are evaluated in order. PMPs represent the permissions map of single hart and are swapped dynamically between different run-time contexts, while the WG checker is typically initialized once at boot time and represents permissions for several worlds across all cores.

The transaction will proceed normally if any rule permits access. Rules may overlap in address ranges, but permissions are strictly accretive.

Implementations can OR together the permissions of all rules matching the address.

If no rule grants access, the access is prevented. Where the following descriptions say the violation

is recorded in the error registers, the update to the `errcause` and `erraddr` registers only occurs if there is no existing error recorded in the registers (i.e., `errcause.ip` and `errcause.be` are both zero).

When a transaction is prevented, if any rule with an address range including any byte of the transaction has the IR or IW bit set for reads and writes respectively, or if no rules overlap the transaction but the `slot[0].cfg.IR` or `slot[0].cfg.IW` bit is set for reads and writes respectively, the failed access attempt will be recorded in the error registers in the checker and will generate an interrupt from the checker.

When a transaction is prevented, but no rule has an address range that includes any byte of the transaction with the IR or IW bit set for reads and writes respectively, or if no rules overlap the transaction and `slot[0].cfg.IR` or `slot[0].cfg.IW` are clear for reads and writes respectively, no interrupts are generated and no errors are recorded in the error registers of the checker.

Speculative accesses that violate permissions checks should not be reported immediately as errors via interrupts. In some cases, an error response on the bus transaction enables the initiating agent to take appropriate action if the speculative access attempts to commit.

When a transaction is prevented, if any rule with an address range including any byte of the transaction has the ER or EW bit set for reads and writes respectively, or if no rules overlap the transaction but the `slot[0].cfg.ER` or `slot[0].cfg.EW` bit is set for reads and writes respectively, the failed access attempt will report an error in the bus transaction response. The specific error encoding is bus-protocol specific. Reads will return zero data along with the error response, and writes to the target memory location will be ignored. In addition, the error will be recorded in the error registers of the checker.

When a transaction is prevented, but no rule has an address range that includes any byte of the transaction with the ER or EW bit set for reads and writes respectively, or if no rules overlap the transaction and `slot[0].cfg.ER` or `slot[0].cfg.EW` are clear for reads and writes respectively, read transactions return zero data with no error in the response, and writes are ignored but are acknowledged with no error in the bus transaction response. Unless the corresponding IR or IW bit is set, no errors are recorded in the error registers of the checker.

Some agents and protocols will cause the system to fail when an unexpected error response is received, possibly preventing the permissions violation from being reported and contained. In these cases, the ER/EW bits can be cleared but the violation reported using the IR/IW bits.

Some platforms may provide the ability to "poison" data returned by a failing access to ensure it will cause an exception if consumed, even if consumed much later from a cached copy, but this cannot be assumed to be available in a generic checker interface. Returning zero does have the effect of poisoning instruction fetches for RISC-V harts, for which zero is defined to be an illegal instruction.

When a transaction is prevented, and the appropriate IR, IW, ER, EW bits are all clear, then no errors are recorded in the error registers of the checker.

This prevents a speculative access that should not report permissions failure from blocking the report of a true permissions violation.

When a transaction is prevented, it can cause both a bus-error response and raise an interrupt if the appropriate IR or IW and ER or EW bits are set. In this case, both the `errcause.ip` and `errcause.be` bits are set when the error is recorded in the error registers of the checker.

3.1.6. Checker Reset

At reset, all slots must either be initialized to a platform-specific value or the `cfg` registers must be cleared (OFF and unlocked).

The `errcause.ip` and `errcause.be` fields must be cleared at reset.

3.1.7. Safely Modifying Slot Entries

When modifying the rule in a slot, it is important to not allow a stray transaction to observe an inconsistent rule state in the checker.

To provide atomic updates to a rule, software must first turn off the rule using a single 32b write to set the `cfg.A[1:0]` setting to OFF, then perform any necessary updates to the other registers in the slot, before finally renabling the rule with another write to set the `cfg` register to the desired value.

References

- [risc-v-priv-spec] https://riscv.org/technical/specifications/