RISC-V RERI Architecture Specification

RERI Task Group

Version 0.1, 03/2023: This document is in development. Assume everything can change. See http://riscv.org/spec-state for details.

Table of Contents

Preamble

This document is in the Development state

Assume everything can change. This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Copyright and license information

Contributors

This RISC-V specification has been contributed to directly or indirectly by (in alphabetical order):

Aaron Durbin, Allen Baum, Anup Patel, Cameron McNairy, Dimitris Gizopoulos, David Kruckemeyer, Dhaval Sharma, Greg Favor, Himanshu
Chauhan, Mostafa Hadizadeh, Vedvyas Shanbhogue, Xiaohan Ma

Chapter 1. Introduction

The RERI specification augments RAS features in the SoC with a standard mechanism for reporting and logging errors by means of a memory-mapped register interface to enable error logging, provide the facility to log the detected errors (including their severity, nature, and location), and configuring means to report the error to a RAS handler component. The RAS handler may use this information to determine suitable recovery actions that may include terminating the computation (e.g., terminating a process, etc.), restarting parts or all of the system, etc. to recover from the errors. Additionally, this specification shall support software-initiated error logging, reporting, and testing of error handlers. Lastly, this specification shall provide maximal flexibility to implement error handling and shall co-exist with RAS frameworks defined by other standards such as PCIe, CXL, etc.

A system is an entity that interacts with other entities such as other systems, software, operators, etc. to deliver one or more services in its role as a service provider. A system may itself be a consumer of one or more services provided by one or more other systems. A system thus is a collection of interacting components that implement one or more functions to provide a service.

A service is the behavior as perceived by the consumers of the service. A system may implement the service as one or more functions in the system. The functions used to compose the service may be implemented by one or more components in the system.

A service is described as a set of states that can be observed by the consumer of the service. The set of states observed by the consumer of the service may be further dependent on a set of internal states of the functions that implement the service.

A service is said to be correct if the set of states observed by the consumer of the service match the specification of that service. The specifications of a service may include its functional behavior, performance goals, security objectives, and RAS requirements.

Reliability of a system as a function of time is the probability it continues to provide correct service and may be characterized by metrics such as mean time between failures (MTBF). The services provided by a reliable system fail on faults instead of silently producing incorrect results. Reliable systems incorporate methods to detect occurrence of errors and to signal the errors to the consumers of the service.

Availability of a system as a function of time is the probability that the system provides the expected service and is a measure of tolerance of errors. Systems may increase the availability by minimizing the impact of the errors in one part of the system to the rest of the system. These may be achieved by means such as error correction, redundancy, state checkpoints and rollbacks, error prediction, and error containment.

Serviceability is a measure of time to restore the service to correct operation with minimal disruption to the consumers of the service. These may be achieved by means such as identifying and reporting failures and supporting mechanisms to repair and bring the system back online.

1.1. Faults and Errors

Fault is an incorrect state resulting from failures of components or due to interference from the environment in which the system operates. A fault is permanent if it reflects an irreversible change to the observable system state else the fault is transient. A permanent fault may occur due to a physical defect or due to a flaw in the design of the functions implementing the service itself. A transient fault may occur due to temporary environmental conditions (cosmic rays, voltage glitches, etc.) or due to instability (e.g. marginal hardware).

Some faults that occur in a component may be dormant and only affect the internal state of the component. Such dormant faults however may turn into active faults when that internal state is used by the computation process in that component and produce an error. An error is detected when its presence is indicated by an error message or signal. Malicious software, especially software operating at privileged modes of operation of the system, may attempt to cause errors; the RAS capabilities are designed to prevent such software-induced errors.

Software faults may similarly cause errors that cause the service provided by the system to deviate from its specification. Well known software engineering and reliability techniques may be employed to prevent, detect and recover from software errors. Software errors are not in the scope of this specification. Software should not have the ability to induce hardware errors.

A service failure occurs when the service deviates from its specification due to errors.

Errors may propagate from component X to another component Y that consumes the results of the computation in component X and appears as an error that was detected by an external component. Eventually, if the error propagates to the external state of the service implemented by these components then a service failure occurs.

A reliable system deals with errors through one or more of the following techniques:

- Fault prevention
- Error detection and correction
- Error forecasting

1.2. Fault prevention

Fault prevention involves use of techniques that reduce or prevent errors that may occur after the product has been shipped. These may be accomplished through the use of high quality in product design, technology selection, materials selection, and manufacturing time screening for defects. Through the use of systematic design, technology selection, and manufacturing tests many errors such as those induced by electric fields, temperature stress, switching/coupling noise (e.g. DRAM RowHammer effect), incorrect V/F operating points, insufficient guard bands, meta-stability, etc. can be prevented.

Faults that are not prevented may manifest as errors during operation of the system. Errors that are not detected may still lead to a service failure. For example, an undetected error in an adder used to produce the address of a load may produce a bad address which causes the load to incur an exception and lead to a service failure. Some undetected errors however may not manifest as

exceptions and cause a service failure due to silent data corruption. For example, a circuit performing encryption of a database may silently cause an error in the ciphertext produced leading to the entire database being left in a state where it cannot be decrypted. Such undetected errors that do not lead to a service failure are called silent data errors (SDE). The impact of SDE is generally much higher than errors that lead to a service failure. A resilient system attempts to minimize the probability of SDE to the largest extent possible by implementing error detection capabilities.

1.3. Error Detection and Correction

Error detection involves the use of coding and protocols to detect errors. For example, caches with error correcting codes, TLB entries with parity protection, buses with parity protection on transaction fields, circuitry to detect unexpected and/or illegal encodings, gray codes, voltage sensors, clock/PLL monitors, timing margin sensors, etc. Some components such as memory controllers may actively attempt to detect errors using techniques such as periodic background scrubbing or on-demand scrubbing.

Error correction involves the use of techniques to correct the detected errors. Error correction may be performed by employing error correcting codes and protocols. For example, a processor cache may employ error correcting codes (ECC) to detect and correct errors. Some components may recover from errors by using protocols that involve a retry. For example, a TLB that detects an error may invalidate the entry and attempt to refill it from the page tables, a receiver on a bus that detects an error may request the transmitter to retransmit the transaction, etc. Error correction is thus complete when the error is either corrected or it does not recur on retry. Such errors that were corrected by the hardware are called **corrected errors (CE)**.

Errors that could not be corrected are called uncorrected errors. A component that detects an uncorrected error may allow possibly corrupted data to propagate to the requester of the data but associate an indicator (e.g., poison) with the data. Such errors are said to be **deferred errors (DE)** as they allow the component to continue operation and defer dealing with the error to a later point in time if the data corrupted by the error is consumed. Defering errors allows deferring the error handling to an ultimate consumer of the corrupted data that may be able to provide more precise information to an error handler about the contexts affected by the corruption and thus enable more precise error recover actions by the erro handler. The component that detected and deferred the error may signal an error recovery handler by logging the DE but such DE does not need an immediate remedial action to be performed by the error handler. For example, a memory controller may detect an uncorrectable ECC error on a data in memory but since there is no immediate consumer of the data the memory controller may just mark the data as poisoned and defer the error handling to a component that requests the data. If the poisoned data is never consumed then deferred errors are benign. If the poisoned data is completely overwritten with new data then the associated poison is cleared. If the poisoned data is only partially written then the data continues to be marked as poisoned. If the poisoned data is consumed by a component (e.g. a hart, an IOMMU, a device, etc.) then an **urgent error (UE)** occurs and a recovery handler is invoked as immediate remedial actions are required and further deferreral of the error is not possible.

A component that detects an uncorrected error may be unable to defer the handling of the error by techniques such as poisoning and may instead signal an error recovery handler by logging the UE. For example, a cache controller may detect an uncorrectable ECC error on the memory used to hold

cache tags and since such errors cannot be attributed to any particular data element these errors may be classified as UE.

A component that signals a request for execution of an error recovery handler for an UE may indicate that the error has not propagated beyond the boundaries of the component that detected the error and thus may be **containable** through recovery actions (e.g., terminating the computation, etc.) carried out by the error recovery handler.

Some components act as an intermediary through which the data passes through. For example, a PCIe/CXL port is an intermediary component that by itself does not consume the data it receives from memory but forwards the data to the endpoint. In such cases the component may receive the data with a deferred error. Such a component may propagate the error and not log an error by itself. However, if the component to which the data is being propagated (e.g. a PCIe endpoint) is not capable of handling poison then the former component must signal a UE instead of propagating the corrupted data, as the act of propagation breaks containment of the error.

An error detected by a component may lead to a failure mode where the component may not be able to service requests anymore (e.g. colloquially called jammed, wedged, etc.). For example, an error in the hart pipeline may cause the hart to stop committing instructions, a fabric may be in a state where it cannot process any further requests, the link connecting the memory module to the host may have failed, etc. In such cases invoking a software recovery handler may not be useful as the recovery handler itself needs to generate requests to the failed component to perform the recovery actions. Components in such failed states may use an implementation-defined signal to a system recovery controller (e.g., a board management controller (BMC), an on-chip service controller, etc.) to initiate a RAS-handling reset to restart the component, sub-system, or the system itself to restore correct service operations.

1.4. Error Forecasting

Error forecasting involves the use of corrected errors as a predictor of future uncorrectable permanent failures or other systemic issues such as marginality due to aging, etc. A future service failure could be avoided if the corrected errors can be monitored. To support such monitoring components in a resilient system may include counters to count the corrections performed. Such components may further include a threshold or support a programmable threshold to notify error handlers when the number of corrected errors exceeds the threshold. A component may also track history of corrected errors and determine if the corrected errors are being triggered by transient faults or permanent faults. For example, a cache may detect that certain cells are repeatedly causing errors, a bus may detect that a certain lane is stuck at a logic level and causing errors, etc. In such cases the system may be able to continue operation due to error correction ability but may still raise a notification to error handlers such that maintenance can be scheduled to replace the failing components in the system.

1.5. Glossary

Table 1. Terms and definitions

Term	Definition
CE	Corrected error.

Term	Definition
Custom	A register or data structure field designated for custom use. Software that is not aware of the custom use must ignore custom fields and preserve value held in these fields when writing values to other fields in the same register.
CXL	Compute Express Link bus standard.
DE	Deferred error.
GPA	Guest Physical Address. See Priv. specification.
ID	Identifier.
OS	Operating system.
PCIe	Peripheral Component Interconnect Express bus standard.
RAS	Reliability, Availability, and Serviceability.
RERI	RAS error record register interface.
Reserved	A register or data structure field reserved for future use. Reserved fields in data structures must be set to 0 by software. Software must ignore reserved fields in registers and preserve the value held in these fields when writing values to other fields in the same register.
RO	Read-only - Register bits are read-only and cannot be altered by software. Where explicitly defined, these bits are used to reflect changing hardware state, and as a result bit values can be observed to change at run time. If the optional feature that would Set the bits is not implemented, the bits must be hardwired to Zero
RW	Read-Write - Register bits are read-write and are permitted to be either Set or Cleared by software to the desired state. If the optional feature that is associated with the bits is not implemented, the bits are permitted to be hardwired to Zero.
RW1C	Write-1-to-clear status - Register bits indicate status when read. A Set bit indicates a status event which is Cleared by writing a 1b. Writing a 0b to RW1C bits has no effect. If the optional feature that would Set the bit is not implemented, the bit must be read-only and hardwired to Zero
RW1S	Read-Write-1-to-set - register bits indicate status when read. The bit may be Set by writing 1b. Writing a 0b to RW1S bits has no effect. If the optional feature that introduces the bit is not implemented, the bit must be read-only and hardwired to Zero
SOC	System on a chip, also referred as system-on-a-chip and system-on-chip.
SPA	Supervisor Physical Address. See Priv. specification.
VA	Virtual Address. See Priv. specification.

Term	Definition
UE	Urgent error.
WARL	Write Any values, Reads Legal values: Attribute of a register field that is only defined for a subset of bit encodings, but allow any value to be written while guaranteeing to return a legal value whenever read.
WPRI	Writes Preserve values, Reads Ignore values: Attribute of a register field that is reserved for future standard use.

Chapter 2. Error Logging and Signaling

Components (e.g., a RISC-V hart, a memory controller, etc.) in a system that support error detection may implement one or more banks of error records. Each error record corresponds to a hardware unit of the component and reports errors detected by that hardware unit. A hardware unit may implement multiple error records. One or more error records may be valid at any instance of time due to one or more hardware units in the component detecting an error or due to a hardware unit having detected one or more errors.

Each error bank is memory-mapped and are located within a naturally aligned 4-KiB region (a page) of physical address space that exists for each error bank, i.e., one page per bank. Each error bank may include up to 63 error records. Each error record is a set of registers used to control that error record and to report status, address, and other information relevant to the error recorded in that error record.

The behavior for register accesses where the address is not aligned to the size of the access, or if the access spans multiple registers, of if the size of the access is not 4 bytes or 8 bytes, is `UNSPECIFIED`. The atomicity of access to an 8 byte register is `UNSPECIFIED`. An aligned 4 byte access to a RERI register must be single-copy atomic.

The RERI registers have little-endian byte order (even for systems where all harts are big-endian-only).

Big-endian-configured harts that make use of an RERI may implement the `REV8` byte-reversal instruction defined by the Zbb extension. If `REV8` is not implemented, then endianness conversion may be implemented using a sequence of instructions.

An implementation-specific response occurs if the error bank and/or record is unavailable (e.g., powered down) to memory-mapped accesses. For example, an error bank and/or record may respond with all zero data on reads and may ignore writes. Other implementations may for example, signal a error response on the attempted transaction.

A error bank that is otherwise available for memory-mapped accesses must respond with all zero data on reads and must ignore writes to unimplemented registers in the page.

2.1. Register layout

The error bank registers are organized as a 64-byte header providing information about the error bank followed by an array of 64-byte error records. The offset of error record numbered `i` in the bank is (64 + `i` * 64) where `i` may range from 0 to 62.

Table 2. Error bank Memory-mapped register layout

Offset	Name	Size	Description
0	`vendor_n_imp_id`	8	Vendor and implementation ID.
8	`bank_info`	8	Error bank information.
16	`valid_summary`	8	Summary of valid error records.

Offset	Name	Size	Description
24	Reserved	16	Reserved for future standard use.
40	Custom	24	Designated for custom use.
64 + 64 * i	`control_i`	8	Control register of error record i.
72 + 64 * i	`status_i`	8	Status register of error record i.
80 + 64 * i	`addr_i`	8	Address register of error record i.
88 + 64 * i	`info_i`	8	Information register of error record i.
96 + 64 * i	`suppl_info_i`	8	Supplemental information register of error record i.
104 + 64 * i	`timestamp_i`	8	Timestamp register of error record i.
112 + 64 * i	Reserved	8	Reserved for future standard use.
120 + 64 * i	Custom	8	Designated for custom use.

2.2. Reset behavior

The reset value is `UNSPECIFIED` for RERI registers.

The registers of an error bank may preserve their value across certain types of reset. For example, a warm reset or a RAS initiated reset may preserve the register values whereas a cold reset may reset the values back to their initial state.

Under normal circumstances, when an error is signaled, the RAS handler retrieves the logged errors to process the error condition. In some cases, the RAS handler may not be able to do such processing. For example, the system may be unable to support execution of the RAS handler and cause a RAS initiated reset. Preserving the information logged in error records across such resets allows reporting of unhandled errors that occured in a previous boot of the system.

All registers in an error bank must have the same reset behavior.

2.3. Error bank registers

2.3.1. Vendor and implementation ID (`vendor_n_imp_id`)

The `vendor_n_imp_id` register is a read-only register and its layout is:

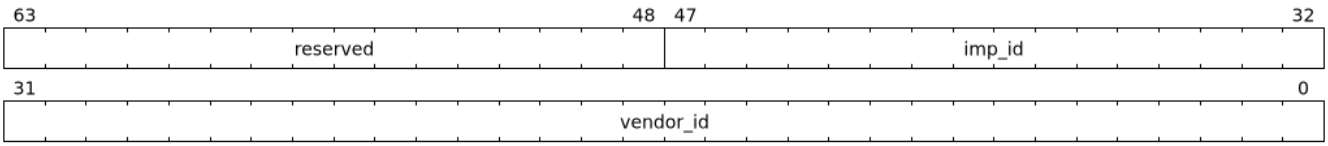

Figure 1. Vendor and implementation ID

The `vendor_id` field follows the encoding as defined by `mvendorid` CSR and provides the JEDEC manufacturer ID of the provider of the component hosting the error bank. A value of 0 may be returned to indicate the field is not implemented or that this is a non-commercial implementation.

The imp_id provides a unique identity, defined by the vendor, to identify the component and revisions of the component implementation hosting the error bank. A value of 0 may be returned to indicate that the field is not implemented. The value returned should reflect the design of the component itself and not of the surrounding system.

The vendor_id and the imp_id are expected to be used as a identifier to determine the format of fields and encodings that UNSPECIFIED by this specification.

2.3.2. Error bank information (bank_info)

The bank_info is a read-only register and its layout is as follows:

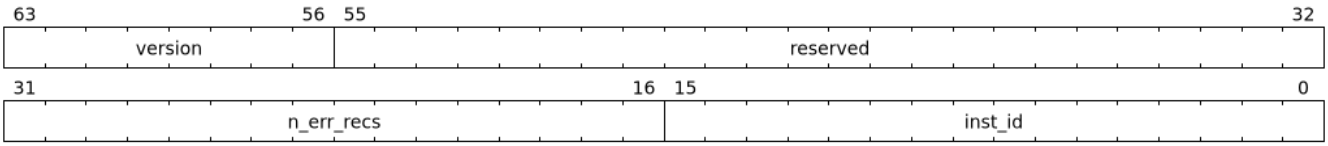

Figure 2. Error bank information

The version field returns the version of the architectural register layout specification implemented by the error bank. The version defined by this specification is 0x01.

The offset of the version field in the error bank shall not change across versions of the register layout. Software should first read the version field and use the value to determine the register layout.

The inst_id field identifies a unique instance, within a package or at least a silicon die, of the component; ideally unique in the whole system. The inst_id are defined by the vendor of the system as a unique identifier for the component. A value of 0 may be returned to indicate the field is not implemented.

The inst_id are expected to be collected and logged as part of the RAS error logs. These may allow the vendor of the silicon to make inferences about the instances of the components that may be vulnerable. As these values differ between vendors of the system and even among systems provided by the same vendor, these are not expected to be useful to the majority of software besides software intimately familiar with that system implementation.

The n_err_recs field indicates the number of error records implemented by the error bank. The field is allowed to have a unsigned value between 1 and 63. The error records of an error bank are located in the 4 KiB memory mapped region reserved for the error bank such that the first error record is at offset 64 and the last error record at offset (64 + 63 * n_err_recs).

2.3.3. Summary of valid error records (valid_summary)

The valid_summary is a read-only register and its layout is as follows:

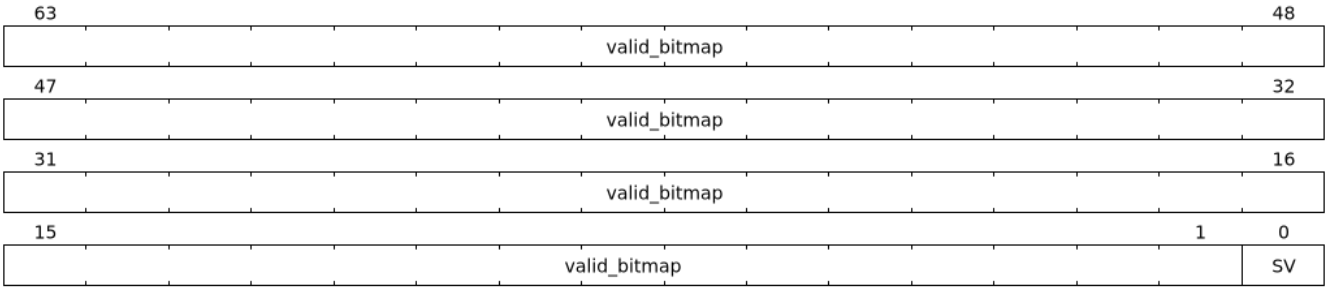

Figure 3. Summary of valid error records

The SV bit when 1 indicates that the valid_bitmap provides a summary of the valid bits from the status registers of this error bank. If this bit is 0 then the error bank does not provide a summary of valid bits and the valid_bitmap is 0.

If SV is 1, then software may use the valid_bitmap to determine which error records in the bank are valid. If this bit is 0 then software must read the status_register_i of each implemented error record in this bank to determine if there is a valid error logged in that error record. The algorithm to determine the records to scan is summarized as follows:

```
if ( valid_summary.SV == 1 ) {
    records_to_scan = valid_summary.valid_bitmap;
} else {
    records_to_scan = (1 << bank_info.n_err_recs) - 1;
}
```

2.4. Error record registers

2.4.1. Control register (control_i)

The control_i is a read/write WARL register used to control error logging by the corresponding error record in the error bank. The layout of this register is as follows:

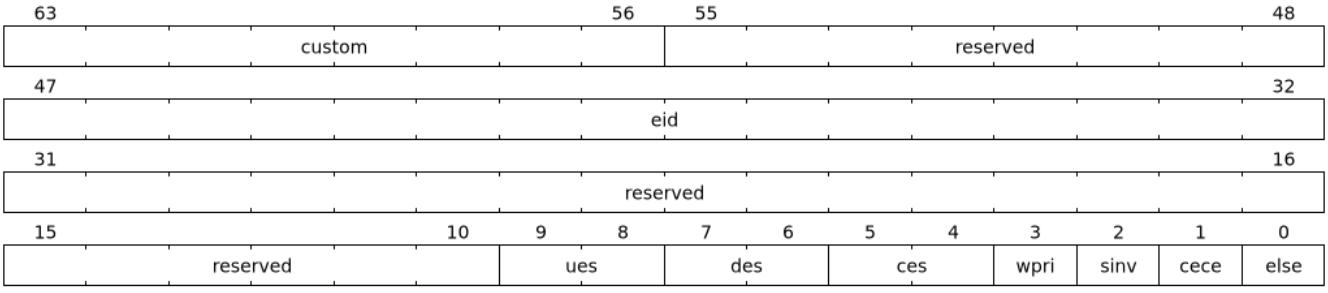

Figure 4. Control register

Error logging and signaling functionality in the error record is enabled if the else field is set to 1. The else field is WARL and may default to 1 or 0 at reset. When else is 1, the hardware unit logs and signals errors in the error record. When else is 0, whether the hardware unit continues detecting and correcting errors is UNSPECIFIED.

When error logging and signaling is disabled, the hardware unit may continue to

silently correct detected errors and when correction is not possible provide corrupt data to the consumers of the data. Alternatively an implementation may disable error detection altogether when logging and signaling are disabled. It is recommended that implementations continue performing error correction even when logging and signaling are disabled.

It is recommended that a hardware component continue to produce error detection and correction codes on data generated by or stored in the hardware component even when logging and signaling is disabled. It is recommended hardware components continue to use containment techniques like data poisoning even when logging and signaling is disabled.

The `ces`, `des`, and `ues` are WARL fields used to enable signaling of UE, DE, and CE respectively when they are logged (i.e. when `else` is 1). Enables for unsupported classes of errors may be hardwired to 0. The encodings of these fields are specified in Table 3.

Table 3. Error signaling enable filed encodings

Encoding	Error signal
0	Signaling is disabled.
1	Signal using a Low-priority RAS signal.
2	Signal using a High-priority RAS signal.
3	Signal using a platform specific RAS signal.

The RAS signals are usually used to notify a RAS error handler. The physical manifestation of the signal is `UNSPECIFIED` by this specification. The information carried by the signal is `UNSPECIFIED` by this specification.

The error signaling enables default to 0 - disabled - at reset to allow a RAS handler an opportunity to initialize itself for handling RAS signals and to initialize the hardware units that generate the RAS signals before error logging and signaling is enabled.

The signal generated by the error record may in addition to causing a interrupt/event notification be also used to carry additional information to aid the RAS error handler in the platform.

The RAS error handler may be implemented by a RISC-V application processor hart in the system, a dedicated RAS handling microcontroller, a finite state machine, etc.

The error signals may be configured, through platform specific means, to notify a RAS error handler in the platform. For example, the High-priority RAS signal may be configured to cause a High-priority RAS local interrupt, an external interrupt, or an NMI and the Low-priority RAS signal may be configured to cause a Low-priority RAS local interrupt or an external interrupt.

If the error record supports corrected-error counting then the corrected-error-counting-enable

(cece) field, when set to 1, enables counting corrected errors in the corrected-error-counter (CEC). The CEC is a counter that holds an unsigned integer count. When cece is 0, the CEC does not count and retains its value. If corrected error counting is not supported by a hardware unit then cece may be hardwired to 0. CEC overflow is signaled using the signal configured in the ces field. When cece is 1, the logging of a CE in does not cause an error signal and an error signal configured in ces occurs only on a CEC overflow.

The sinv bit, when written with a value of 1, causes the v (valid) field and the ceco field in status_i register to be cleared. The sinv field always returns 0 on read.

The error injection delay (eid) field is used to control error record injection. When eid is written with a value greater than 1, the eid starts counting down, at an implementation defined rate, till the value reaches a count of 0. Writing a value of 0 disables the counter. If error injection is not supported by the error record then the eid field may be hardwired to 0. When eid reaches a count of 0, the status register is made valid by setting the status_i.v bit to 1. The status_i.v transition from 0 to 1 generates a RAS signal corresponding to the type of error setup in the status_i register. The counter continues to count even if the status_i register was overwritten by a hardware detected error before the eid counts down to 0.

The error record injection capability only injects an error record and not an error into the hardware itself. The error record injection capability is expected to be used to test the RAS handlers and is not intended to be used for verification of the hardware implementation itself.

Other implementation specific mechanisms may be provided to generate and/or emulate hardware error conditions. When hardware error injection capabilities are implemented, the implementation should ensure that these capabilities cannot be misused to maliciously inject hardware errors that may lead to security issues.

2.4.2. Status register (status_i)

The status_i is a read-write WARL register that reports errors detected by the hardware unit.

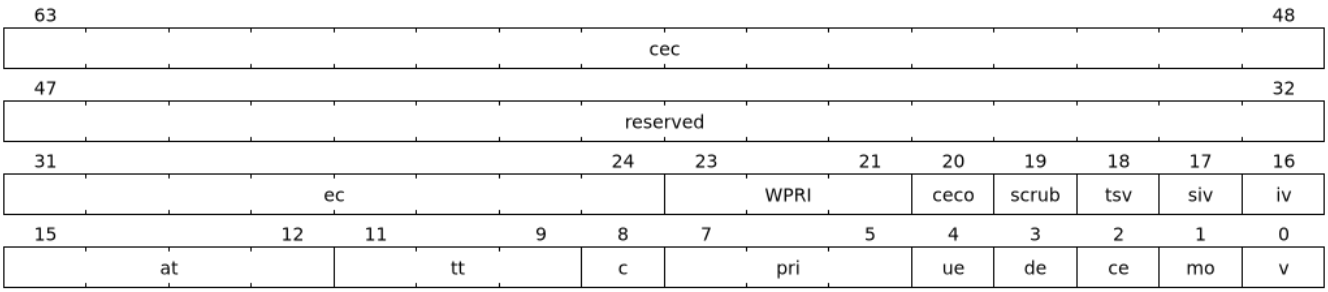

Figure 5. Status register

The error record hold a valid error log if the v field is 1.

If the detected error was deferred then de is set to 1. If the detected error was corrected then ce is set to 1. If the detected error could not be corrected or deferred and thus needs urgent handling by an error handler, then the ue bit is set to 1. If the error record does not log a class of errors (e.g., does not support DE), then the corresponding bit may be hardwired to 0. If the bits corresponding to more than one error class are set to 1 then the error record holds information about the highest

severity error class among the bits set.

When `v` is 1, if more errors of the same class as the error currently logged in the error record occur then the `mo` bit is set to indicate the multiple occurrence of errors of the same severity.

Each error of an error class that may be logged in an error record are associated with a priority which is a number between 0 and 7; zero being the highest priority and 7 being the lowest priority. The `pri` field indicates the priority of the currently logged error in the error record.

When an error occurs the `c` may be set to 1 to indicate that the error has not propagated beyond the boundaries of the hardware unit that detected the error and thus may be **containable** through recovery actions (e.g., terminating the computation, etc.) carried out by the error recovery handler.

For example, a RISC-V hart by causing the precise data corruption exception on attempts to consume corrupted/poisoned data may contain the error to the program currently executing on the hart. A RISC-V RERI by aborting the transaction that caused the corrupted data from being consumed may contain the error to the device initiating the transaction, etc.

While the `c` bit indicates that the error may be containable the RAS handler may or may not be able to recover the system from such errors. The RAS handler must make the recovery determination based on additional information provided in the error record such as the address of the memory where corruption was detected, etc.

The address-type (`at`) field indicates the type of address reported in the `addr_i` register. A error record that does not report addresses may hardwire this field to 0. The encodings of the `at` field are listed in Table 4.

Table 4. Address type encodings

Encoding	Description
0	None. When `at` is 0, the contents of the `addr_i` register are `UNSPECIFIED`.
1	Supervisor physical address (SPA).
2	Guest physical address (GPA).
3	Virtual address (VA).
4-15	Component specific.

The component specific address types may be used to report address such as a local bus address, a DRAM address, etc. The interpretation of such addresses is component specific.

A set of component specific encodings are defined to allow a platform to use an encoding per type of component specific addresses.

The `addr_i` register must hold the address of type determined by the `at` field. Additional non-redundant information about the location accessed using the

address (e.g., cache set and way, etc.) may be reported in the `info_i` register.

The `tt` field reports the type of transaction that detected the error and its encodings are listed in Table 5. A error record that does not report transaction types may hardwire this field to 0.

Table 5. Transaction type encodings

Encoding	Description
0	Unspecified or not applicable.
1-3	Reserved for future standard extensions.
4	Explicit read.
5	Explicit write.
4	Implicit read.
5	Implicit write.

Implicit read and write are accesses that may be implicitly performed by hardware to perform an explicit operation. For example, a load or store instruction executed by the hart may perform implicit memory accesses to page table data structures. Another example, might be where processing a memory transaction may require a fabric component to implicitly access a routing table data structure.

Instruction memory accesses by a hart are termed as implicit accesses by the hart. However for the purposes of error logging only the implicit accesses to data structures like the page tables and guest page tables used to determine the address of the instruction to fetch are termed as implicit accesses. The read to fetch the instruction bytes themselves are termed as explicit reads.

If the detected error reports additional information in the `info_i` register then `iv` field is set to 1. If the detected error reports additional supplemental information in the `suppl_info_i` register then `siv` field is set to 1. The `iv` and/or `siv` fields may be hardwired to 0 if the error record does not provide information in `info_i` and/or `suppl_info_i` registers.

If the error record holds a timestamp of when the last error was logged in the `timestamp_i` register then the `tsv` bit is set to 1. This field may be hardwired to 0 if the error record does not report a timestamp with the error.

The `scrub` bit is valid when a CE is logged and when set to 1 indicates that the storage location that held the data value has been updated with the corrected value (i.e., the data has been scrubbed). An implementation that cannot make this distinction or where the error record is not associated with storage elements (e.g., correcting errors detected on bus transactions) this field may be hardwired to 0. If this property is unconditionally true for a hardware unit then this field may be hardwired to 1.

The `ec` field holds an error code that provides a description of the detected error. Standard `ec` encodings are defined in Table 6. If an error record detects an error that does not correspond to a standard `ec` encoding then such errors may be reported using a custom encoding. The custom

encodings have the most significant bit set to 1 to differentiated them from the standard encodings.

An error record that supports the 1 setting of the cece field in control_i, implements a 16-bit wide corrected-error-counter in the cec field. When cece is 1, the cec is incremented on each CE in addition to logging details of the error in the error record registers. If an integer overflow occurs on cec increment then the corrected-error-counter-overflow (ceco) field is set to 1. The cec continues to count following an overflow. The cec and ceco fields hold valid data and continue to count even when the v field is 0.

Some hardware units may maintain a history of CE and may report a CE and increment the cec only if the error is not identical to a previously reported CE.

Some hardware units may implement low pass filters (e.g., leaky buckets) that throttle the rate which CE are reported and counted.

When a UE or DE error is logged the cec and ceco fields are not modified and retain their values.

Software may determine if the error record was read atomically by first reading the registers of the error record, then clearing the valid in status_i by writing 1 to control_i.sinv and then reading the status_i register again to determine if the value (besides the v field) changed. If a change was detected then the process may be repeated to read the latest reported error.

2.4.3. Address register (addr_i)

The addr_i is a WARL register that reports the address associated with the detected error when status_i.at is not 0. If status_i.at is 0, the value in this register is UNSPECIFIED. An implementation that does not report addresses may hardwire this register to 0. Some fields of the register may be hardwired to zero if the field is unused to report any type of address. In general, to the extent possible, the error record should capture all significant parts of the address. However as a function of the type of error being logged some address fields may be zeroes. Some highest address bits may be fixed or may be sign-extensions or may be zero-extensions of the next lowest address bit depending on the type of address reported.

2.4.4. Information register (info_i)

The info_i field provides additional information about the error when status_i.iv is 1. If status_i.iv is 0, the value in this register is UNSPECIFIED. An implementation that does not report any additional information may hardwire this register to 0.

The format of the register is UNSPECIFIED by this specification. This field may be interpreted using the error code in status_i.ec along with implementation specific and implementation defined format and rules.

This field may be used to report error specific information to help locate the failing component, guide recovery actions, whether error is transient or permanent, etc. The field may be used to report more detailed information about the location of the error within the component. For example, set and way where the error was

> detected, the parity group that was in error, the ECC syndrome, a protocol FSM state, the input that caused an assertion to fail, etc.
>
> Components that are field replaceable units or detect errors in connected field replacement units may log additional information in the `info_i` register to help identify the failing component. For example, a memory controller may log the memory channel associated with the error such as the DIMM channel, bank, column, row, rank, subRank, device ID, etc.

2.4.5. Supplemental information register (`suppl_info_i`)

The `suppl_info_i` field provides additional information about the error when `status_i.siv` is 1. This information may supplement the information provided in `info_i` register. If `status_i.siv` is 0, the value in this register is `UNSPECIFIED`. An implementation that does not report any supplemental information may hardwire this register to 0.

The format of the register is `UNSPECIFIED` by this specification. This field may be interpreted using the error code in `status_i.ec` along with implementation specific and implementation defined format and rules.

2.4.6. Timestamp register (`timestamp_i`)

The `timestamp_i` field provides a timestamp for the last error recorded in the error record if `status_i.tsv` is 1. When `status.tsv` is 0, the value in this register is `UNSPECIFIED`. An implementation that does not report a timestamp may hardwire this register to 0. Some fields of the register may be hardwired to zero if the field is unused to report the timestamp.

The frequency and resolution of the timestamp are `UNSPECIFIED`.

2.5. Error record overwrite rules

When a hardware unit detects an error it may find its error record still valid due to an earlier detected error that has not been consumed yet by software.

The overwrite rules allow a higher severity error to overwrite a lower severity error. UE has the highest severity, followed by DE, and then CE. When the two errors have same severity the priority of the errors is used to determine if the error record is overwritten. Higher priority errors overwrite the lower priority errors. When a error record is overwritten by a higher severity error (DE/CE by UE, DE by UE, or CE by DE), the status bits indicating the severity of the first error are retained (i.e., are sticky).

The rules for writing the error record are as follows:

Listing 1. Error record writing rules

```
    Let new_status be the value to be recorded in status_i register for the new error
    overwrite = FALSE
    if status_i.v == 1
        // There is a valid first error recorded
```

```
        if ( severity(new_error) > severity(status_i) )
            // Severity of second error is higher than first error
            // The DE and CE bits are sticky and retained to provide the
            // overwrite history
            status_i.UE |= new_status.UE
            status_i.DE |= new_status.DE
            status_i.CE |= new_status.CE
            status_i.MO = 0
            overwrite = TRUE
        endif
        if ( severity(new_status) == severity(status_i) )
            // Severity of second error is same as of first error
            // Note multiple occurrences of same severity error
            status_i.MO = 1
            // Overwrite if priority of second error is higher
            if ( new_status.pri > status_i.pri )
                overwrite = TRUE;
            endif
        endif
    else
        // There is a no error valid recorded
        // Note the severity of the new error
        status_i.UE = new_status.UE
        status_i.DE = new_status.DE & ~new_status.UE
        sttaus_i.CE = new_status.CE & ~new_status.UE & ~new_status.DE
        overwrite = TRUE;
    endif

    if ( overwrite = TRUE )
        status_i.pri = new_status.pri
        status_i.c = new_status.c
        status_i.tt = new_status.tt
        status_i.at = new_status.at
        status_i.iv = new_status.iv
        status_i.siv = new_status.siv
        status_i.tsv = new_status.tsv
        status_i.scrub = new_status.scrub
        status_i.ec = new_status.ec
        // Update addr_i, info_i, suppl_info_i, timestam_i appropriately
    endif

    status_i.v = 1
```

When the `status_i.MO` is 1, if the logged error is a UE then the recovery handler should restart the system to bring it to a correct state as an UE record has been lost. If the `status_i.MO` is 1 and the logged error is a DE or a CE then the recovery handler may keep the system operational.

A 0 to 1 transition of the `status_i.v` causes the signal configured in the `control_i` register for the highest severity error recorded in the error record to be generated.

2.6. Error logging defined by other standards

Standards such as PCIe and CXL define standardized error logging architectures such as the PCIe Advanced Error Reporting (AER). Specifications such as CXL define a standardized set of RAS requirements to be complied to by host and devices. The RISC-V RERI extension complements the error reporting architecture defined by these standards with a RISC-V standard for reporting errors for components that are not PCIe/CXL components. There may also be other error logging mechanisms, possibly custom, that are employed alongside the RERI specification.

The RISC-V system components such as PCIe root ports or PCIe Root Complex Event Collectors may themselves implement error logging compliant with the RISC-V RERI extensions and thus provide a unified error reporting mechanism in such systems. For example, a root complex event collector may support an error log to report errors logged in the AER logs.

2.7. Error code encodings

Table 6. Error code encodings

Encoding	Error signal
0	None
1	Other
2	Corrupted data access (e.g. consumption of poison)
3	Cache data error
4	Cache scrubbing detected data error
5	Cache tag or state error
6	Cache unspecified error
7	Snoop-filter/directory tag or state error
8	Snoop-filter/directory unspecified error
9	TLB/Page-walk cache data error
10	TLB/Page-walk cache tag error
11	TLB/Page-walk cache unspecified error
12	Hart architectural state error
13	Interrupt controller/register file error
14	Interconnect data error
15	Interconnect other error
16	Internal watchdog error
17	Internal datapath, memory, or execution units error
18	System memory command/address bus error
19	System memory unspecified error
20	System memory data error

Encoding	Error signal
21	System Memory scrubbing detected data error
22	Protocol Error - illegal input/output error
23	Protocol Error - illegal/unexpected state error
24	Protocol Error - timeout
25	System internal controller (power management, security, etc.) error
26	Deferred error passthrough not supported
27	PCIe/CXL component detected errors.
28 - 127	Reserved for future standard extensions.
128 - 255	Designated for custom use.

Bibliography