RISC-V Capacity and Bandwidth QoS Register Interface

RISC-V CMQRI Task Group

Version v0.9.1, 2023-10-31: Draft

Table of Contents

Preamble

This document is in the Stable 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:

Aaron Durbin, Adrien Ricciardi, Allen Baum, Allison Randal, Ambika Krishnamoorthy, Daniel Gracia Pérez, David Kruckemeyer, David Weaver, Derek Hower, Drew Fustini, Eric Shiu, Greg Favor, Ravi Sahita, Vedvyas Shanbhogue

Chapter 1. Introduction

Quality of Service (QoS) is defined as the minimal end-to-end performance that is guaranteed in advance by a service level agreement (SLA) to a workload. A workload may be a single application, a group of applications, a virtual machine, a group of virtual machines, or a combination of those. The performance may be measured in the form of metrics such as instructions per cycle (IPC), latency of servicing work, etc.

Various factors such as the available cache capacity, memory bandwidth, interconnect bandwidth, CPU cycles, system memory, etc. affect the performance in a computing system that runs multiple workloads concurrently. Furthermore, when there is arbitration for shared resources, the prioritization of the workloads' requests against other competing requests may also affect the performance of the workload. Such interference due to resource sharing may lead to unpredictable workload performance [1].

When multiple workloads are running concurrently on modern processors with large core counts, multiple cache hierarchies, and multiple memory controllers, the performance of a workload becomes less deterministic or even non-deterministic. This is because the performance depends on the behavior of all the other workloads in the machine that contend for the shared resources, leading to interference. In many deployment scenarios, such as public cloud servers, the workload owner may not be in control of the type and placement of other workloads in the platform.

System software can control some of these resources available to the workload, such as the number of hardware threads made available for execution, the amount of system memory allocated to the workload, the number of CPU cycles provided for execution, etc.

System software needs additional tools to control interference to a workload and thereby reduce the variability in performance experienced by one workload due to other workloads' cache capacity usage, memory bandwidth usage, interconnect bandwidth usage, etc. through a resource allocation capability. The resource allocation capability enables system software to reserve capacity and/or bandwidth to meet the performance goals of the workload. Such controls enable improving the utilization of the system by collocating workloads while minimizing the interference caused by one workload to another [2].

Effective use of the resource allocation capability requires hardware to provide a resource monitoring capability by which the resource requirements of a workload needed to meet a certain performance goal can be characterized. A typical use model involves profiling the resource usage of the workload using the resource monitoring capability and to establish resource allocations for the workload using the resource allocation capability.

The allocation may be in the form of capacity or bandwidth, depending on the type of resource. For caches, TLBs, and directories, the resource allocation is in the form of storage capacity. For interconnects and memory controllers, the resource allocation is in the form of bandwidth.

Workloads generate different types of accesses to shared resources. For example, some of the requests may be for accessing instructions and others may be to access data operated on by the workload. Certain data accesses may not have temporal locality whereas others may have a high probability of reuse. In some cases it is desirable to provide differentiated treatment to each type of access by providing unique resource allocation to each access type.

RISC-V Capacity and Bandwidth Controller QoS Register Interface (CBQRI) specification specifies:

1. QoS identifiers to identify workloads that originate requests to the shared resources. These QoS identifiers include an identifier for resource allocation configurations and an identifier for the monitoring counters used to monitor resource usage. These identifiers accompany each request made by the workload to the shared resource. Chapter 2 specifies the mechanism to associate the identifiers with workloads.
2. Access-type identifiers to accompany request to access a shared resource to allow differentiated treatment of each access-type (e.g., code vs. data, etc.). The access-types are defined in Chapter 2.
3. Register interface for capacity allocation in controllers such as shared caches, directories, etc. The capacity allocation register interface is specified in Chapter 3.
4. Register interface for capacity usage monitoring. The capacity usage monitoring register interface is specified in Chapter 3.
5. Register interface for bandwidth allocation in controllers such as interconnect and memory controllers. The bandwidth allocation register interface is specified in Chapter 4.
6. Register interface for bandwidth usage monitoring. The bandwidth usage monitoring register interface is specified in Chapter 4.

The capacity and bandwidth controller register interfaces for resource allocation and usage monitoring are defined as memory-mapped registers. Each controller that supports CBQRI provides a set of registers that are located at a range of physical address space that is a multiple of 4-KiB and the lowest address of the range is aligned to 4-KiB. The memory-mapped registers may be accessed using naturally aligned 4-byte or 8-byte memory accesses. The controller behavior for register accesses where the address is not aligned to the size of the access, or if the access spans multiple registers, or if the size of the access is not 4 bytes or 8 bytes, is `UNSPECIFIED`. A 4-byte access to a register must be single-copy atomic. Whether an 8-byte access to a CBQRI register is single-copy atomic is `UNSPECIFIED`, and such an access may appear, internally to the CBQRI implementation, as if two separate 4-byte accesses were performed.

The CBQRI registers are defined in such a way that software can perform two individual 4 byte accesses, or hardware can perform two independent 4 byte transactions resulting from an 8 byte access, to the high and low halves of the register as long as the register semantics, with regards to side-effects, are respected between the two software accesses, or two hardware transactions, respectively.

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

Big-endian-configured harts that make use of the register interface 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.

A controller may support a subset of capabilities defined by CBQRI. When a capability is not supported the registers and/or fields used to configure and/or control such capabilities are hardwired to 0. Each controller supports a capabilities register to enumerate the supported capabilities.

Chapter 2. QoS Identifiers

Monitoring or allocation of resources requires a way to identify the originator of the request to access the resource.

CBQRI and the Ssqosid extension provides a mechanism by which a workload can be associated with a resource control ID (RCID) and a monitoring counter ID (MCID) that accompany each request made by the workload to shared resources.

To provide differentiated services to workloads, CBQRI defines a mechanism to configure resource usage limits, in the form of capacity or bandwidth, per supported access type, for an RCID in the resource controllers that control accesses to such shared resources.

To monitor the resource utilization by a workload, CBQRI defines a mechanism to configure counters identified by the MCID to count events in the resource controllers that control accesses to such shared resources.

Guidelines for sizing the QoS IDs and need for differentiated IDs for monitoring is discussed in Section 6.1.

2.1. Associating RCID and MCID with requests

The RCID in the request is used by the resource controllers to determine the resource allocations (e.g., cache occupancy limits, memory bandwidth limits, etc.) to enforce. The MCID in the request is used by the resource controllers to identify the ID of a counter to monitor resource usage (e.g., cache occupancy, memory bandwidth, etc.).

2.1.1. RISC-V hart initiated requests (Ssqosid)

The Ssqosid extension [3] introduces a read/write S/HS-mode register (sqoscfg) to configure QoS Identifiers to be used with requests made by the hart to shared resources.

2.1.2. Device initiated requests

Devices may be configured with an RCID and MCID for requests originated from the device if the device implementation supports such capability. The method to configure the QoS identifiers into devices is UNSPECIFIED.

Where the device does not natively support being configured with an RCID and MCID, the implementation may provide a shim at the device interface that may be configured with the RCID and MCID that are associated with requests originating from the device. The method to configure such QoS identifiers into a shim is UNSPECIFIED.

If the system supports an IOMMU, then the IOMMU may be configured with the RCID and MCID to associate requests from the device with QoS identifiers. The RISC-V IOMMU [4] extension to support configuring QoS identifiers is specified in Chapter 5.

2.2. Access-type (AT)

In some usages, in addition to providing differentiated service among workloads, the ability to differentiate between resource usage for accesses made by the same workload may be required. For example, the capacity allocated in a shared cache for code storage may be differentiated from the capacity allocated for data storage and thereby avoid code from being evicted from such shared cache due to a data access.

When differentiation based on access type (e.g. code vs. data) is supported the requests also carry an access-type (AT) indicator. The resource controllers may be configured with separate capacity and/or bandwidth allocations for each supported access-type. CBQRI defines a 3-bit AT field, encoded as specified in Table 1, in the register interface to configure differentiated resource allocation and monitoring for each AT.

Table 1. Encodings of AT field

Value	Name	Description
0	Data	Requests to access data.
1	Code	Requests for code execution.
2-5	Reserved	Reserved for future standard use.
6-7	Custom	Designated for custom use.

For unsupported AT values the resource controller behaves as if AT was 0.

Chapter 3. Capacity-controller QoS Register Interface

Controllers, such as cache controllers, that support capacity allocation and usage monitoring provide a memory-mapped capacity-controller QoS register interface.

The capacity controller allocates capacity in fixed multiples of *capacity units*. A group of these *capacity units* is referred to as a *capacity block*. One or more *capacity blocks* may be allocated to a workload. When a workload requests capacity allocation, the capacity is allocated using *capacity units* situated within the *capacity blocks* assigned to the workload. Capacity blocks can also be shared among one or more workloads. Optionally, the capacity controller may allow configuration of a limit on the maximum number of *capacity units* that can be occupied in the *capacity blocks* allocated to a given workload.

For example, a cache controller may allocate capacity in multiples of cache blocks. In this context, a cache block serves as a capacity unit, and a group of cache blocks forms a capacity block. A cache controller supporting capacity allocation by ways might define a capacity block to be the cache blocks in one way of the cache.

Table 2. Capacity-controller QoS register layout

Offset	Name	Size	Description	Optional?
0	cc_capabilities	8	Capabilities	No
8	cc_mon_ctl	8	Usage monitoring control	Yes
16	cc_mon_ctr_val	8	Monitoring counter value	Yes
24	cc_alloc_ctl	8	Capacity allocation control	Yes
32	cc_block_mask	M * 8	Capacity block mask	Yes
N	cc_cunits	8	Capacity units count	Yes

The size and offset in Table 2 are specified in bytes.

The size of the cc_block_mask register is determined by the NCBLKS field of the cc_capabilities register but is always a multiple of 8 bytes.

The reset value is 0 for the following register fields.

- cc_mon_ctl.BUSY field
- cc_alloc_ctl.BUSY field

The reset value is UNSPECIFIED for all other registers fields.

The capacity controllers at reset must allocate all available capacity to RCID value of 0. When the capacity controller supports capacity allocation per access-type, then all available capacity is shared by all the access-type for RCID=0. The capacity allocation for all other RCID values is UNSPECIFIED. The capacity controller behavior for handling a request with a non-zero RCID value before configuring the capacity controller with capacity allocation for that RCID is UNSPECIFIED.

3.1. Capacity-controller capabilities (cc_capabilities)

The cc_capabilities register is a read-only register that holds the capacity-controller capabilities.

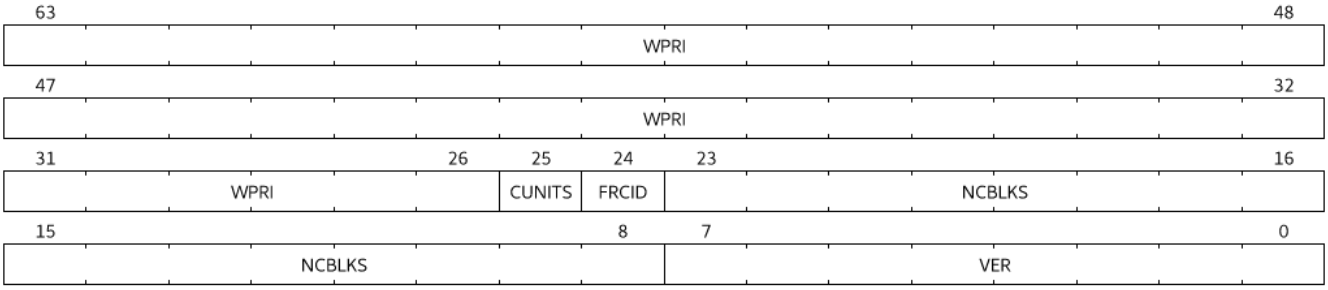

Figure 1. Capacity-controller capabilities register fields

The VER field holds the version of the specification implemented by the capacity controller. The low nibble is used to hold the minor version of the specification and the upper nibble is used to hold the major version of the specification. For example, an implementation that supports version 1.0 of the specification reports 0x10.

The NCBLKS field holds the total number of allocatable *capacity blocks* in the controller. The capacity represented by an allocatable *capacity block* is UNSPECIFIED. The capacity controllers support allocating capacity in fixed multiples of an allocatable *capacity block*.

For example, a cache controller that defines a way of the cache as a capacity block may report the number of ways as the number of allocatable capacity blocks.

If CUNITS is 1, the controller supports specifying a limit on the *capacity units* that can be occupied by an RCID in *capacity blocks* allocated to it.

If FRCID is 1, the controller supports an operation to flush and deallocate the *capacity blocks* occupied by an RCID.

3.2. Capacity usage monitoring control (cc_mon_ctl)

The cc_mon_ctl register is used to control monitoring of capacity usage by a MCID. When the controller does not support capacity usage monitoring the cc_mon_ctl register is read-only zero.

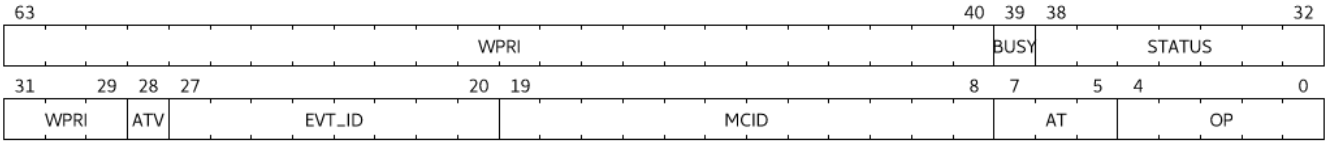

Figure 2. Capacity usage monitoring control (cc_mon_ctl)

Capacity controllers that support capacity usage monitoring implement a usage monitoring counter for each supported MCID. The usage monitoring counter may be configured to count a monitoring event. When an event matching the event configured for the MCID occurs then the monitoring counter is updated. The event matching may optionally be filtered by the access-type.

The OP, AT, ATV, MCID, and EVT_ID fields of the register are WARL fields.

The OP field is used to instruct the controller to perform an operation listed in Table 3.

Table 3. Capacity usage monitoring operations (OP)

Operation	Encoding	Description
—	0	Reserved for future standard use.
CONFIG_EVENT	1	Configure the counter selected by MCID to count the event selected by EVT_ID, AT, and ATV. The EVT_ID encodings are listed in Table 4.
READ_COUNTER	2	Snapshot the value of the counter selected by MCID into cc_mon_ctr_val register. The EVT_ID, AT, and ATV fields are not used by this operation.
—	3-23	Reserved for future standard use.
—	24-31	Designated for custom use.

The EVT_ID field is used to program the identifier of the event to count in the monitoring counter selected by MCID. The AT field (See Table 1) is used to program the access-type to count, and its validity is indicated by the ATV field. When ATV is 0, the counter counts requests with all access-types, and the AT value is ignored.

Table 4. Capacity usage monitoring event ID (EVT_ID)

Event ID	Encoding	Description
None	0	Counter does not count and retains its value.
Occupancy	1	Counter is incremented by 1 when a request with a matching MCID and AT allocates a unit of capacity. The counter is decremented by 1 when a unit of capacity is de-allocated.
—	2-127	Reserved for future standard use.
—	128-256	Designated for custom use.

When the EVT_ID for a MCID is programmed with a non-zero and legal value using the CONFIG_EVENT operation, the counter is reset to 0 and starts counting matching events for requests with the matching MCID and AT (if ATV is 1). However, if the EVT_ID is programmed to 0, the counter retains its current value but stops counting.

A controller that does not support monitoring by access-type can hardwire the ATV and the AT fields to 0, indicating that the counter counts requests with all access-types.

When the cc_mon_ctl register is written, the controller may need to perform several actions that may not complete synchronously with the write. A write to the cc_mon_ctl sets the read-only BUSY bit to 1 indicating the controller is performing the requested operation. When the BUSY bit reads 0, the operation is complete, and the read-only STATUS field provides a status value (see Table 5 for details). Written values to the BUSY and the STATUS fields are ignored. An implementation that can complete the operation synchronously with the write may hardwire the BUSY bit to 0. The state of the BUSY bit, when not hardwired to 0, shall only change in response to a write to the register. The STATUS field remains valid until a subsequent write to the cc_mon_ctl register.

Table 5. cc_mon_ctl.STATUS field encodings

STATUS	Description
0	Reserved
1	The operation was successfully completed.
2	An invalid operation (OP) was requested.
3	An operation was requested for an invalid MCID.
4	An operation was requested for an invalid EVT_ID.
5	An operation was requested for an invalid AT.
6-63	Reserved for future standard use.
64-127	Designated for custom use.

When the BUSY bit is set to 1, the behavior of writes to the cc_mon_ctl is UNSPECIFIED. Some implementations may ignore the second write, while others may perform the operation determined by the second write. To ensure proper operation, software must first verify that the BUSY bit is 0 before writing the cc_mon_ctl register.

3.3. Capacity usage monitoring counter value (cc_mon_ctr_val)

The cc_mon_ctr_val is a read-only register that holds a snapshot of the counter selected by the READ_COUNTER operation. When the controller does not support capacity usage monitoring, the cc_mon_ctr_val register always reads as zero.

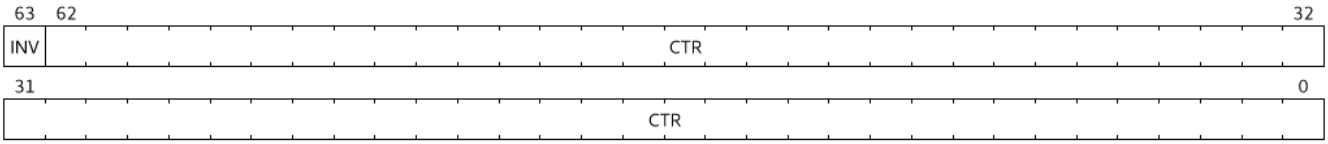

Figure 3. Capacity usage monitoring counter value (cc_mon_ctr_val)

The counter is valid if the INV field is 0. The counter may be marked INV if the controller, for UNSPECIFIED reasons determine the count to be not valid. The counters marked INV may become valid in future.

The counter shall not decrement below zero. If an event should occur that would otherwise result in a negative value, the counter will continue to hold a value of 0.

Following a reset of the counter to zero, a capacity de-allocation may attempt to drive its value below zero. This scenario may occur when the MCID is reassigned to a new workload, yet the capacity controller continues to hold capacity initially allocated by the previous workload. In such cases, the counter shall not decrement below zero and shall remain at zero. After a brief period of execution for the new workload post-counter reset, the counter value is expected to stabilize to reflect the capacity usage of this new workload.

Some implementations may not store the MCID of the request that caused the capacity to be allocated with every unit of capacity in the controller to optimize on the storage overheads. Such controllers may in turn rely on statistical sampling to report the capacity usage by tagging only a subset of the capacity units.

Set-sampling is a technique commonly used in caches to estimate the cache occupancy with a relatively small sample size. The basic idea behind set-sampling is to select a subset of the cache sets and monitor only those sets. By keeping track of the hits and misses in the monitored sets, it is possible to estimate the overall cache occupancy with a high degree of accuracy. The size of the subset needed to obtain accurate estimates depends on various factors, such as the size of the cache, the cache access patterns, and the desired accuracy level. Research [5] has shown that set-sampling can provide statistically accurate estimates with a relatively small sample size, such as 10% or less, depending on the cache properties and sampling technique used.

When the controller has not observed enough samples to provide an accurate value in the monitoring counter, it may report the counter as being INV until more accurate measurements are available. This helps to prevent inaccurate or misleading data from being used in capacity planning or other decision-making processes.

3.4. Capacity allocation control (cc_alloc_ctl)

The cc_alloc_ctl register is used to configure allocation of capacity to an RCID per access-type (AT). The OP, RCID and AT fields in this register are WARL. If a controller does not support capacity allocation then this register is read-only zero. If the controller does not support capacity allocation per access-type then the AT field is read-only zero.

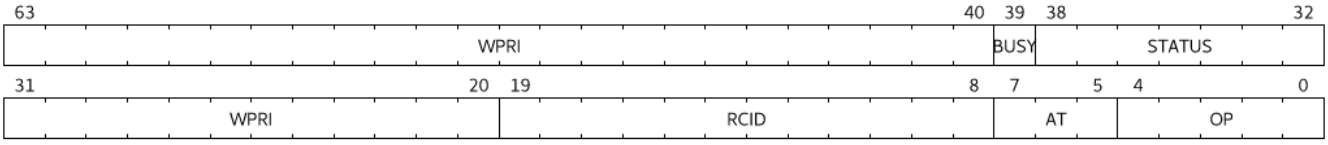

Figure 4. Capacity allocation control (cc_alloc_ctl)

The OP field is used to instruct the capacity controller to perform an operation listed in Table 6. Some operations necessitate the specification of the *capacity blocks* to act upon. For such operations, the targeted *capacity blocks* are designated in the form of a bitmask in the cc_block_mask register. Additionally, certain operations require the *capacity unit* limit to be defined in the cc_cunits register. To execute operations that require a capacity block mask and/or a capacity unit limit, software must first program the cc_block_mask and/or the cc_cunits register, followed by initiating the operation via the cc_alloc_ctl register.

Table 6. Capacity allocation operations (OP)

Operation	Encoding	Description
—	0	Reserved for future standard use.
CONFIG_LIMIT	1	Configure a capacity allocation for requests by RCID and of access-type AT. The *capacity blocks* allocation is specified in the cc_block_mask register, and a limit on capacity units is specified in the cc_cunits register.

Operation	Encoding	Description
READ_LIMIT	2	Read back the previously configured capacity allocation for requests by RCID and of access-type AT. The configured *capacity block* allocation is returned as a bit-mask in the cc_block_mask register, and the configured limit on *capacity units* is available in the cc_cunits register on successful completion of the operation.
FLUSH_RCID	3	Flushes the *capacity units* used by the specified RCID and access-type AT. This operation is supported if the capabilities.FRCID bit is 1. The cc_block_mask and cc_cunits registers are not used for this operation. The configured *capacity block* allocation or the *capacity unit* limit is not changed by this operation.
—	4-23	Reserved for future standard use.
—	24-31	Designated for custom use.

Capacity controllers enumerate the allocatable *capacity blocks* in the NCBLKS field of the cc_capabilities register. The cc_block_mask register is programmed with a bit-mask where each bit represents a *capacity block* for the operation. A limit on the *capacity unit*, if configuration of such limits is supported (i.e., cc_capabilities.CUNIT=1), that can be occupied in the allocated *capacity blocks* may be programmed in the cc_cunits register. If configuration of a limit on the *capacity units* is not supported, then the controller allows the use of all *capacity units* in the allocated *capacity blocks*. A value of zero programmed into cc_cunits indicates that no limits should be enforced on *capacity unit* allocation.

A capacity allocation must be configured for each supported access-type by the controller. An implementation that does not support capacity allocation per access-type may hardwire the AT field to 0 and associate the same capacity allocation configuration for requests with all access-types. When capacity allocation per access-type is supported, identical limits may be configured for two or more access-types if different capacity allocation per access-type is not required. If capacity is not allocated for each access-type supported by the controller, the behavior is UNSPECIFIED.

A cache controller that supports capacity allocation indicates the number of allocatable capacity blocks in cc_capabilities.NCBLKS field. For example, let's consider a cache with NCBLKS=8. In this example, the RCID=5 has been allocated capacity blocks numbered 0 and 1 for requests with access-type AT=0, and has been allocated capacity blocks numbered 2 for requests with access-type AT=1. The RCID=3 in this example has been allocated capacity blocks numbered 3 and 4 for both AT=0 and AT=1 access-types as separate capacity allocation by access-type is not required for this workload. Further in this example, the RCID=6 has been configured with the same capacity block allocations as RCID=3. This implies that they share a common capacity allocation in this cache but may have been associated with different RCID to allow differentiated treatment in another capacity and/or bandwidth controller.

	7	6	5	4	3	2	1	0
RCID=3, AT=0	0	0	0	1	1	0	0	0
RCID=3, AT=1	0	0	0	1	1	0	0	0
RCID=5, AT=0	0	0	0	0	0	0	1	1
RCID=5, AT=1	0	0	0	0	0	1	0	0
RCID=6, AT=0	0	0	0	1	1	0	0	0
RCID=6, AT=1	0	0	0	1	1	0	0	0

Some controllers allow setting a limit on capacity units in allocated capacity blocks. In exclusive allocations, like for RCID=5, the limit can be the capacity block's maximum capacity. For shared allocations, such as between RCID=3 and RCID=6, individual limits can be set. For example, if two capacity blocks represent 100 units and RCID=3 has a 30-unit limit while RCID=6 has a 70-unit limit, they can use 30% and 70% of the shared capacity blocks, respectively.

The FLUSH_RCID operation may incur a long latency to complete. New requests to the controller by the RCID being flushed are allowed. Additionally, the controller is allowed to deallocate capacity that was allocated after the operation was initiated.

For cache controllers, the FLUSH_RCID operation may perfom an operation similar to that performed by the RISC-V CBO.FLUSH instruction on each cache block that is part of the allocation configured for the RCID.

The FLUSH_RCID operation can be used as part of reclaiming a previously allocated RCID and associating it with a new workload. When such a reallocation is performed, the capacity controllers may have capacity allocated by the old workload and thus for a short warmup duration the capacity controller may be enforcing capacity allocation limits that reflect the usage by the old workload. Such warmup durations are typically not statistically significant, but if that is not desired, then the FLUSH_RCID operation can be used to flush and evict capacity allocated by the old workload.

When the cc_alloc_ctl register is written, the controller may need to perform several actions that may not complete synchronously with the write. A write to the cc_alloc_ctl sets the read-only BUSY bit to 1 indicating the controller is performing the requested operation. When the BUSY bit reads 0, the operation is complete, and the read-only STATUS field provides a status value (Table 7) of the requested

operation. Values written to the BUSY and the STATUS fields are always ignored. An implementation that can complete the operation synchronously with the write may hardwire the BUSY bit to 0. The state of the BUSY bit, when not hardwired to 0, shall only change in response to a write to the register. The STATUS field remains valid until a subsequent write to the cc_alloc_ctl register.

Table 7. cc_alloc_ctl.STATUS field encodings

STATUS	Description
0	Reserved
1	The operation was successfully completed.
2	An invalid or unsupported operation (OP) requested.
3	An operation was requested for an invalid RCID.
4	An operation was requested for an invalid AT.
5	An invalid *capacity block* mask was specified.
6-63	Reserved for future standard use.
64-127	Designated for custom use.

When the BUSY bit is set to 1, the behavior of writes to the cc_alloc_ctl register, cc_cunits register, or to the cc_block_mask register is UNSPECIFIED. Some implementations may ignore the second write and others may perform the operation determined by the second write. To ensure proper operation, software must verify that BUSY bit is 0 before writing any of these registers.

3.5. Capacity block mask (cc_block_mask)

The cc_block_mask is a WARL register. If the controller does not support capacity allocation, i.e., NCBLKS is 0, then this register is read-only 0.

The register has NCBLKS bits each corresponding to one allocatable *capacity block* in the controller. The width of this register is variable but always a multiple of 64 bits. The bitmap width in bits (BMW) is determined by the equation below. The division operation in this equation is an integer division.

$$BMW = \lfloor \frac{NCBLKS + 63}{64} \rfloor \times 64 \qquad (1)$$

Bits NCBLKS-1:0 are read-write, and the bits BMW-1:NCBLKS are read-only zero.

The process of configuring capacity allocation for an RCID and AT begins by programming the cc_block_mask register with a bit-mask that identifies the *capacity blocks* to be allocated and, if supported, by programming the cc_cunits register with a limit on the capacity units that may be occupied in those capacity blocks. Next, the cc_alloc_ctl register is written to request a CONFIG_LIMIT operation for the RCID and AT. Once a capacity allocation limit has been established, a request may be allocated capacity in the *capacity blocks* allocated to the RCID and AT associated with the request. It is important to note that at least one *capacity block* must be allocated using cc_block_mask when allocating capacity, or else the operation will fail with STATUS=5. Overlapping *capacity block* masks among RCID and/or AT are allowed to be configured.

A set-associative cache controller that supports capacity allocation by ways can advertise NCBLKS as the number of ways per set in the cache. To Allocate capacity in such a cache for an RCID and AT, a subset of ways must be selected and mask of the selected ways must be programmed in cc_block_mask when requesting the CONFIG_LIMIT operation.

To read the *capacity block* allocation for an RCID and AT, the controller provides the READ_LIMIT operation which can be requested by writing to the cc_alloc_ctl register. Upon successful completion of the operation, the cc_block_mask register holds the configured *capacity block* allocation.

3.6. Capacity units (cc_cunits)

The cc_cunits register is a read-write WARL register. If the controller does not support capacity allocation (i.e., NCBLKS is set to 0), this register shall be read-only zero.

If the controller does not support configuring limits on *capacity units* that may be occupied in the allocated *capacity blocks* (i.e., cc_capabilities.CUNITS=0) then this register shall be read-only zero. In such cases the controller will allow utilization of all available *capacity units* by an RCID within the *capacity blocks* allocated to it.

If the controller supports configuring limits on *capacity units* that may be occupied in the allocated *capacity blocks* (i.e., cc_capabilities.CUNITS=1) then this register sets an upper limit on the number of *capacity units* that can be occupied by an RCID in the *capacity blocks* allocated for an AT. A value of zero specified in the cc_cunits register indicates that no limit is configured.

The sum of the cc_cunits configured for the RCID sharing a *capacity block* allocation may exceed the *capacity units* represented by that *capacity block* allocation.

When multiple RCID instances share a capacity block allocation, the cc_cunits register may be employed to set an upper limit on the number of capacity units each RCID can occupy.

For instance, consider a group of four RCID instances configured to share a set of capacity blocks, representing a total of 100 capacity units. Each RCID could be configured with a limit of 30 capacity units, ensuring that no individual RCID exceeds 30% of the total shared capacity units.

The capacity controller may enforce these limits through various techniques. Examples include:

1. *Refraining from allocating new capacity units to an RCID that has reached its limit.*
2. *Evicting previously allocated capacity units when a new allocation is required.*

These methods are not exhaustive and can be applied either individually or in combination to maintain capacity unit limits.

When the limit on the capacity units is reached or is about to be reached, the capacity controller may initiate additional operations. These could include throttling certain activities (e.g., prefetches) of the corresponding workload requests.

To read the *capacity unit* limit for an RCID and AT, the controller provides the READ_LIMIT operation which can be requested by writing to the cc_alloc_ctl register. Upon successful completion of the operation, the cc_cunits register holds the configured *capacity unit* allocation limit.

Chapter 4. Bandwidth-controller QoS Register Interface

Controllers, such as memory controllers, that support bandwidth allocation and bandwidth usage monitoring provide a memory-mapped bandwidth-controller QoS register interface.

Table 8. Bandwidth-controller QoS register layout

Offset	Name	Size	Description	Optional?
0	bc_capabilities	8	Capabilities	No
8	bc_mon_ctl	8	Usage monitoring control	Yes
16	bc_mon_ctr_val	8	Monitoring counter value	Yes
24	bc_alloc_ctl	8	Bandwidth allocation control	Yes
32	bc_bw_alloc	8	Bandwidth allocation	Yes

The reset value is 0 for the following register fields.

- bc_mon_ctl.BUSY field
- bc_alloc_ctl.BUSY field

The reset value is UNSPECIFIED for all other register fields.

The bandwidth controllers at reset must allocate all available bandwidth to RCID value of 0. When the bandwidth controller supports bandwidth allocation per access-type, the access-type value of 0 of RCID=0 is allocated all available bandwidth, while all other access-types associated with that RCID share the bandwidth allocation with AT=0. The bandwidth allocation for all other RCID values is UNSPECIFIED. The bandwidth controller behavior in handling a request with a non-zero RCID value before configuring the bandwidth controller with bandwidth allocation for that RCID is also UNSPECIFIED.

4.1. Capabilities (bc_capabilities)

The bc_capabilities register is a read-only register that holds the bandwidth-controller capabilities.

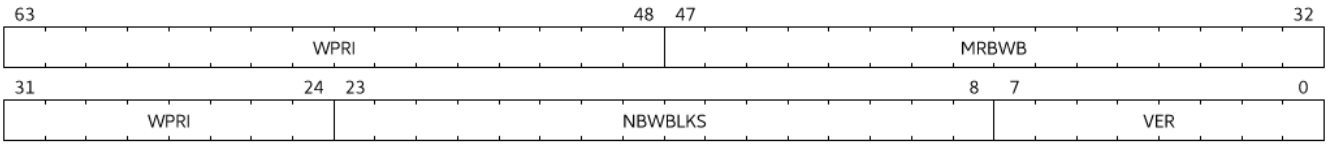

Figure 5. Capabilities register fields

The VER field holds the version of the specification implemented by the bandwidth controller. The low nibble is used to hold the minor version of the specification and the upper nibble is used to hold the major version of the specification. For example, an implementation that supports version 1.0 of the specification reports 0x10.

The NBWBLKS field holds the total number of available bandwidth blocks in the controller. The bandwidth represented by each bandwidth block is UNSPECIFIED. The bandwidth controller supports

reserving bandwidth in multiples of a bandwidth block, which enables proportional allocation of bandwidth.

Bandwidth controllers may limit the maximum bandwidth that may be reserved to a value smaller than NBWBLKS. The MRBWB field reports the maximum number of bandwidth blocks that can be reserved.

The bandwidth controller needs to meter the bandwidth usage by a workload to determine if it is exceeding its allocations and, if necessary, take necessary measures to throttle the workload's bandwidth usage. Therefore, the instantaneous bandwidth used by a workload may either exceed or fall short of the configured allocation. QoS capabilities are statistical in nature and are typically designed to enforce the configured bandwidth over larger time windows. By not allowing all available bandwidth blocks to be reserved for allocation, the bandwidth controller can handle such transient inaccuracies.

4.2. Bandwidth usage monitoring control (bc_mon_ctl)

The bc_mon_ctl register is used to control monitoring of bandwidth usage by a MCID. When the controller does not support bandwidth usage monitoring, the bc_mon_ctl register is read-only zero.

Figure 6. Bandwidth usage monitoring control (bc_mon_ctl)

Bandwidth controllers that support bandwidth usage monitoring implement a usage monitoring counter for each supported MCID. The usage monitoring counter may be configured to count a monitoring event. When an event matching the event configured for the MCID occurs then the monitoring counter is updated. The event matching may optionally be filtered by the access-type. The monitoring counter for bandwidth usage counts the number of bytes transferred by requests matching the monitoring event as the requests go past the monitoring point.

The OP, AT, MCID, and EVT_ID fields of the register are WARL fields.

The OP field is used to instruct the controller to perform an operation listed in Table 9.

Table 9. Usage monitoring operations (OP)

Operation	Encoding	Description
—	0	Reserved for future standard use.
CONFIG_EVENT	1	Configure the counter selected by MCID to count the event selected by EVT_ID, AT, and ATV. The EVT_ID encodings are listed in Table 10.
READ_COUNTER	2	Snapshot the value of the counter selected by MCID into bc_mon_ctr_val register. The EVT_ID, AT, and ATV fields are not used by this operation.
—	3-23	Reserved for future standard use.

Operation	Encoding	Description
—	24-31	Designated for custom use.

The EVT_ID field is used to program, using the CONFIG_EVENT operation, the identifier of the event to count in the monitoring counter selected by MCID. The AT field is used to program the access-type to count, and its validity is indicated by the ATV field. When ATV is 0, the counter counts requests with all access-types, and the AT value is ignored.

Table 10. Bandwidth monitoring event ID (EVT_ID)

Event ID	Encoding	Description
None	0	Counter does not count and retains its value.
Total Read and Write byte count	1	Counter is incremented by the number of bytes transferred by a matching read or write request as the requests go past the monitor.
Total Read byte count	2	Counter is incremented by the number of bytes transferred by a matching read request as the requests go past the monitor.
Total Write byte count	3	Counter is incremented by the number of bytes transferred by a matching write request as the requests go past the monitor.
—	4-127	Reserved for future standard use.
—	128-256	Designated for custom use.

When the EVT_ID for a MCID is programmed with a non-zero and legal value, the counter is reset to 0 and starts counting matching events for requests with the matching MCID and AT (if ATV is 1). However, if the EVT_ID is configured as 0, the counter retains its current value but stops counting.

A controller that does not support monitoring by access-type can hardwire the ATV and the AT fields to 0, indicating that the counter counts requests with all access-types.

When the bc_mon_ctl register is written, the controller may need to perform several actions that may not complete synchronously with the write. A write to the bc_mon_ctl sets the read-only BUSY bit to 1 indicating the controller is performing the requested operation. When the BUSY bit reads 0, the operation is complete, and the read-only STATUS field provides a status value (see Table 11 for details). Written values to the BUSY and the STATUS fields are ignored. An implementation that can complete the operation synchronously with the write may hardwire the BUSY bit to 0. The state of the BUSY bit, when not hardwired to 0, shall only change in response to a write to the register. The STATUS field remains valid until a subsequent write to the bc_mon_ctl register.

Table 11. bc_mon_ctl.STATUS field encodings

STATUS	Description
0	Reserved
1	The operation was successfully completed.
2	An invalid operation (OP) was requested.
3	An operation was requested for an invalid MCID.
4	An operation was requested for an invalid EVT_ID.

STATUS	Description
5	An operation was requested for an invalid AT.
6-63	Reserved for future standard use.
64-127	Designated for custom use.

When the BUSY bit is set to 1, the behavior of writes to the bc_mon_ctl is UNSPECIFIED. Some implementations may ignore the second write, while others may perform the operation determined by the second write. To ensure proper operation, software must first verify that the BUSY bit is 0 before writing the bc_mon_ctl register.

4.3. Bandwidth monitoring counter value (bc_mon_ctr_val)

The bc_mon_ctr_val is a read-only register that holds a snapshot of the counter selected by a READ_COUNTER operation. When the controller does not support bandwidth usage monitoring, the bc_mon_ctr_val register always reads as zero.

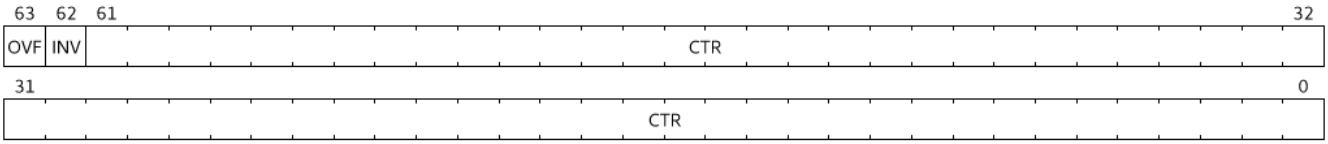

Figure 7. Bandwidth monitoring counter value (bc_mon_ctr_val)

The counter is valid if the INV field is 0. The counter may be marked INV if, for UNSPECIFIED reasons, the controller determines the count to be not valid. Such counters may become valid in the future. Additionally, if an unsigned integer overflow of the counter occurs, then the OVF bit is set to 1.

A counter may be marked as INV if the controller has not been able to establish an accurate counter value for the monitored event.

The counter provides the number of bytes transferred by requests matching the EVT_ID as they go past the monitoring point. A bandwidth value may be determined by reading the byte count value at two instances of time T1 and T2 (see equation (2)). If the value of the counter at time T1 was B1, and at time T2 is B2, then the bandwidth can be calculated as follows. The frequency of the time source is represented by T_{freq}.

$$Bandwidth = T_{freq} \times \frac{B2 - B1}{T2 - T1} \qquad (2)$$

The width of the counter is UNSPECIFIED but the unimplemented bits must be read-only zero.

While the width of the counter is UNSPECIFIED, it is recommended to be wide enough to prevent more than one overflow per sample when the sampling frequency is 1 Hz.

If an overflow was detected then software may discard that sample and reset the counter and overflow indication by reprogramming the event using CONFIG_EVENT operation.

4.4. Bandwidth allocation control (bc_alloc_ctl)

The bc_alloc_ctl register is used to control the allocation of bandwidth to an RCID per AT. If a controller does not support bandwidth allocation, then the register is read-only zero. If the controller does not support bandwidth allocation per access-type, then the AT field is read-only zero.

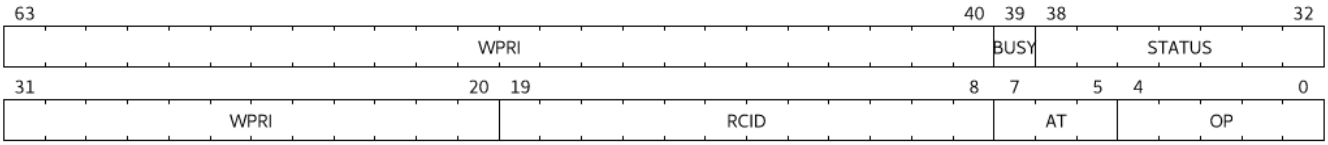

Figure 8. Bandwidth allocation control (bc_alloc_ctl)

The OP field instructs the bandwidth controller to perform an operation listed in Table 12. The bc_alloc_ctl register is used in conjunction with the bc_bw_alloc register to perform bandwidth allocation operations. If the requested operation uses the operands configured in bc_bw_alloc, software must first program the bc_bw_alloc register with the operands for the operation before requesting the operation.

Table 12. Bandwidth allocation operations (OP)

Operation	Encoding	Description
—	0	Reserved for future standard use.
CONFIG_LIMIT	1	Establishes reserved bandwidth allocation for requests by RCID and of access-type AT. The bandwidth allocation is specified in bc_bw_alloc register.
READ_LIMIT	2	Reads back the previously configured bandwidth allocation for requests by RCID and of access-type AT. The current configured allocation is written to bc_bw_alloc register on completion of the operation.
—	3-23	Reserved for future standard use.
—	24-31	Designated for custom use.

A bandwidth allocation must be configured for each access-type supported by the controller. When differentiated bandwidth allocation based on access-type is not required, one of the access-types may be designated to hold a default bandwidth allocation, and the other access-types can be configured to share the allocation with the default access-type. If bandwidth is not allocated for each access-type supported by the controller, the behavior is UNSPECIFIED.

When the bc_alloc_ctl register is written, the controller may need to perform several actions that may not complete synchronously with the write. A write to the bc_alloc_ctl sets the read-only BUSY bit to 1 indicating the controller is performing the requested operation. When the BUSY bit reads 0, the operation is complete, and the read-only STATUS field provides a status value (see Table 13 for details). Written values to the BUSY and the STATUS fields are ignored. An implementation that can complete the operation synchronously with the write may hardwire the BUSY bit to 0. The state of the BUSY bit, when not hardwired to 0, shall only change in response to a write to the register. The STATUS field remains valid until a subsequent write to the bc_alloc_ctl register.

Table 13. bc_alloc_ctl.STATUS field encodings

STATUS	Description
0	Reserved
1	The operation was successfully completed.
2	An invalid operation (OP) was requested.
3	An operation was requested for an invalid RCID.
4	An operation was requested for an invalid AT.
5	An invalid or unsupported reserved bandwidth block was requested.
6-63	Reserved for future standard use.
64-127	Designated for custom use.

4.5. Bandwidth allocation configuration (bc_bw_alloc)

The bc_bw_alloc is used to program reserved bandwidth blocks (Rbwb) for an RCID for requests of access-type AT using the CONFIG_LIMIT operation. If a controller does not support bandwidth allocation, then the bc_bw_alloc register is read-only zero.

The bc_bw_alloc holds the previously configured reserved bandwidth blocks for an RCID and AT on successful completion of the READ_LIMIT operation.

Bandwidth is allocated in multiples of bandwidth blocks, and the value in Rbwb must be at least 1 and must not exceed MRBWB. Otherwise, the CONFIG_LIMIT operation fails with STATUS=5. Additionally, the sum of Rbwb allocated across all RCIDs must not exceed MRBWB, or the CONFIG_LIMIT operation fails with STATUS=5.

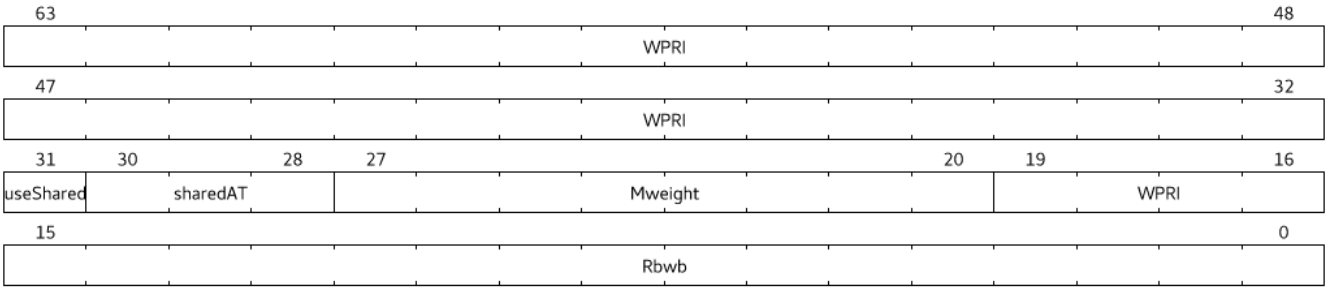

Figure 9. Bandwidth allocation configuration (bc_bw_alloc)

The Rbwb, Mweight, sharedAT, and useShared are all WARL fields.

Bandwidth allocation is typically enforced by the bandwidth controller over finite accounting windows. The process involves measuring the bandwidth consumption over an accounting window and using the measured bandwidth to determine if an RCID is exceeding its bandwidth allocations for each access-types. The specifics of how the accounting window is implemented are UNSPECIFIED, but is expected to provide a statistically accurate control of the bandwidth usage over a few accounting intervals.

The Rbwb represents the bandwidth that is made available to a RCID for requests matching AT, even when all other RCID are using their full allocation of bandwidth.

If there is non-reserved or unused bandwidth available in an accounting interval, RCIDs may compete for additional bandwidth. The non-reserved or unused bandwidth is proportionately shared among the competing RCIDs using the configured Mweight parameter, which is a number between 0 and 255. A larger weight implies a greater fraction of the bandwidth. A weight of 0 implies that the configured limit is a hard limit, and the use of unused or non-reserved bandwidth is not allowed.

The sharing of non-reserved bandwidth is not differentiated by access-type. Therefore, the Mweight parameter must be programmed identically for all access-types. If this parameter is programmed differently for each access-type, then the controller may use the parameter configured for any of the access-types, but the behavior is otherwise well defined.

When the Mweight parameter is not set to 0, the amount of unused bandwidth allocated to RCID=x during contention with another RCID that is also permitted to use unused bandwidth is determined by dividing the Mweight of RCID=x by the sum of the Mweight of all other contending RCIDs. This ratio P is determined by equation (3).

$$P = \frac{Mweight_x}{\sum_{r=1}^{r=n} Mweight_r} \qquad (3)$$

The bandwidth enforcement is typically work-conserving, meaning that it allows unused bandwidth to be used by requestors enabled to use it even if they have consumed their Rbwb.

When contending for unused bandwidth, the weighted share is typically computed among the RCIDs that are actively generating requests in that accounting interval and have a non-zero weight programmed.

If unique bandwidth allocation is not required for an access-type, then the useShared parameter may be set to 1 for a CONFIG_LIMIT operation. When useShared is set to 1, the sharedAT field specifies the access-type with which the bandwidth allocation is shared by the access-type in bc_alloc_ctl.AT. In this case, the Rbwb and Mweight fields are ignored, and the configurations of the access-type in sharedAT are applied. If the access-type specified by sharedAT does not have unique bandwidth allocation, meaning that it has not been configured with useShared=0, then the behavior is UNSPECIFIED.

The useShared and sharedAT fields are read-only zero if the bandwidth controller does not support bandwidth allocation per access-type.

When unique bandwidth allocation for an access-type is not required then one or more access-types may be configured with a shared bandwidth allocation. For example, consider a bandwidth controller that supports 3 access-types. The access-type 0 and 1 of RCID 3 are configured with unique bandwidth allocations and the access-type 2 is configured to share bandwidth allocation with access-type 1. The example configuration is illustrated as follows:

	Rbwb	Mweight	useShared	sharedAT
RCID=3, AT=0	100	16	0	N/A
RCID=3, AT=1	50	16	0	N/A
RCID=3, AT=2	N/A	N/A	1	1

Chapter 5. IOMMU extension for QoS ID

Monitoring or allocation of resources accessed by the IOMMU and devices governed by the IOMMU requires a way to associate QoS IDs with such requests. This section specifies a RISC-V IOMMU [4] extension to:

- Configure and associate QoS IDs for device-originated requests.
- Configure and associate QoS IDs for IOMMU-originated requests.

5.1. IOMMU registers

The specified memory-mapped register layout defines a new IOMMU register named `iommu_qosid`. This register is used to configure the Quality of Service (QoS) IDs associated with IOMMU-originated requests. The register has a size of 4 bytes and is located at an offset of 624 from the beginning of the memory-mapped region.

Table 14. IOMMU Memory-mapped register layout

Offset	Name	Size	Description	Is Optional?
624	`iommu_qosid`	4	QoS IDs for IOMMU requests.	Yes
628	Reserved	60	Reserved for future use (`WPRI`)	

5.1.1. Reset behavior

If the reset value for `ddtp.iommu_mode` field is `Bare`, then the `iommu_qosid.RCID` field must have a reset value of 0.

At reset, it is required that the `RCID` field of `iommu_qosid` be set to 0 if the IOMMU is in `Bare` mode, as typically the resource controllers in the SoC default to a reset behavior of associating all capacity or bandwidth to the `RCID` value of 0.

5.1.2. IOMMU capabilities (`capabilities`)

The IOMMU capabilities register has been extended with a new field, `QOSID`, which enumerates support for associating QoS IDs with requests made through the IOMMU.

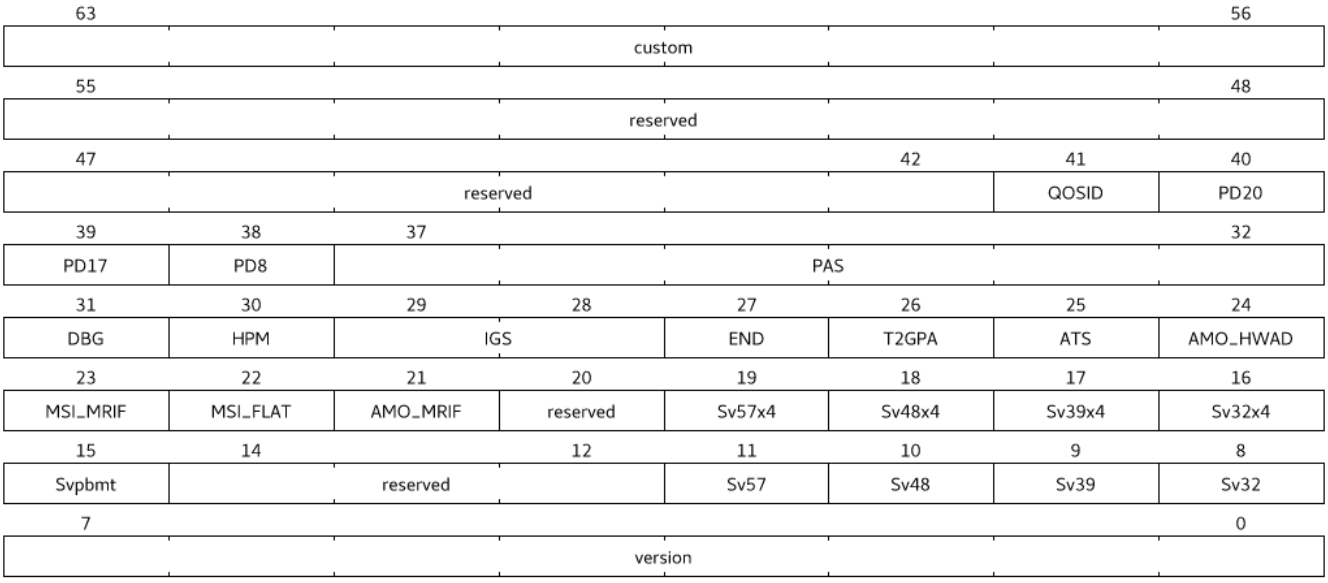

Figure 10. IOMMU capabilities register fields

Bits	Field	Attribute	Description
41	QOSID	RO	Associating QoS IDs with requests is supported.

5.1.3. IOMMU QoS ID (iommu_qosid)

The iommu_qosid register fields are defined as follows:

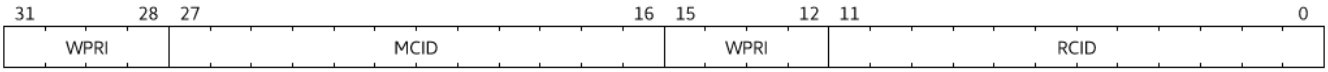

Figure 11. iommu_qosid register fields

Bits	Field	Attribute	Description
11:0	RCID	WARL	RCID for IOMMU initiated requests.
15:12	reserved	WPRI	Reserved for standard use.
27:16	MCID	WARL	MCID for IOMMU initiated requests.
31:28	reserved	WPRI	Reserved for standard use.

IOMMU-initiated requests for accessing the following data structures use the value programmed in the RCID and MCID fields of the iommu_qosid register.

- Device directory table (DDT)
- Fault queue (FQ)
- Command queue (CQ)
- Page-request queue (PQ)
- IOMMU-initiated MSI (Message-signaled interrupts)

When ddtp.iommu_mode == Bare, all device-originated requests are associated with the QoS IDs configured in the iommu_qosid register.

5.2. Device-context fields

The ta field of the device context is extended with two new fields, RCID and MCID, to configure the QoS IDs to associate with requests originated by the devices.

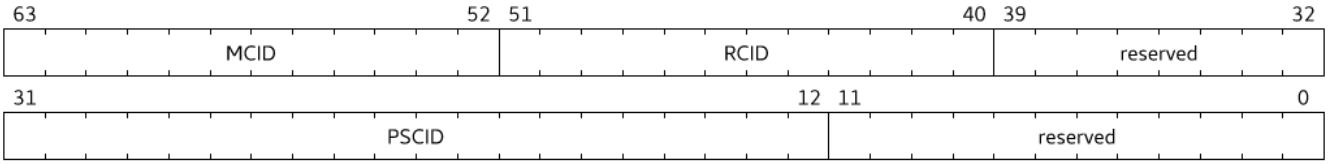

Figure 12. Translation attributes (ta) field

IOMMU-initiated requests for accessing the following data structures use the value configured in the RCID and MCID fields of DC.ta.

- Process directory table (PDT)
- Second-stage page table
- First-stage page table
- MSI page table
- Memory-resident interrupt file (MRIF)

The RCID and MCID configured in DC.ta are provided to the IO bridge on successful address translations. The IO bridge should associate these QoS IDs with device-initiated requests.

If capabilities.QOSID is 1 and DC.ta.RCID or DC.ta.MCID is wider than that supported by the IOMMU, a DC with DC.tc.V=1 is considered misconfigured. In this case, the IOMMU should stop and report "DDT entry misconfigured" (cause = 259).

5.3. IOMMU ATC capacity allocation and monitoring

The IOMMU may support capacity allocation and usage monitoring in the IOMMU address translation cache (IOATC) by implementing a capacity controller register interface.

Some IOMMU may support multiple IOATC where the capacity of each such IOATC may not be the same. When multiple IOATC are implemented, (e.g., corresponding to each page sizes supported), the IOMMU may implement a capacity controller register interface for each IOATC to enable capacity allocation in each IOATC.

Chapter 6. Hardware Guidelines

6.1. Sizing QoS Identifiers

In a typical implementation the number of RCID bits implemented (e.g., to support 10s of RCIDs) may be smaller than the number of MCID bits implemented (e.g., to support 100s of MCIDs).

It is a typical usage to associate a group of applications/VMs with a common RCID and thus sharing a common pool of resource allocations. The resource allocations for the RCID is established to meet the SLA objectives of all members of the group. If SLA objectives of one or more members of the group stop being met, the resource usage of one or more members of the group may be monitored by associating them with a unique MCID and this iterative analysis process used to determine the optimal strategy - increasing resources allocated to the RCID, moving some members to a different RCID, migrating some members away to another machine, etc. - for restoring the SLA. Having a sufficiently large pool of MCID speeds up this analysis.

To achieve maximum flexibility in the allocation of QoS IDs to workloads, it is recommended that all resource controllers in the system support an identical number of RCID and MCID. Requests typically need to be processed by multiple controllers, such as caches, fabrics, and memory controllers. In such cases, if not all controllers accessed by a request support an identical number of IDs, software will be constrained, using only the IDs supported by all controllers.

6.2. Sizing monitoring counters

Typically software samples the monitoring counters periodically to monitor capacity and bandwidth usage. The width of the monitoring counters is recommended to be wide enough to not cause more than one overflow per sample when sampled at a frequency of 1 Hz.

Chapter 7. Software Guidelines

7.1. Reporting capacity and bandwidth controllers

The capability and bandwidth controllers that are present in the system should be reported to operating systems using methods such as ACPI and/or device tree. For each capacity and bandwidth controller, the following information should be reported using these methods:

- Type of controller (e.g, cache, interconnect, memory, etc.)
- Location of the register programming interface for the controller
- Placement and topology describing the hart and IO bridges that share the resources controlled by the controller
- The number of QoS identifiers supported by the controller

7.2. Context switching QoS Identifiers

Typically, the contents of the `sqoscfg` CSR are updated with a new `RCID` and/or `MCID` by the HS/S-mode scheduler if the `RCID` and/or `MCID` of the new workload (a process or a VM) is not same as that of the old workload.

A context switch usually involves saving the context associated with the workload being switched away from and restoring the context of the workload being switched to. Such a context switch may be invoked in response to an explicit call from the workload (i.e, as a function of an `ECALL` invocation) or may be done asynchronously (e.g., in response to a timer interrupt). In such cases the scheduler may want to execute with the `sqoscfg` configuration of the workload being switched away from such that this execution is attributed to the workload being switched away from and then prior to restoring the new workloads context, first switch to the `sqoscfg` configuration appropriate for the workload being switched to such that all of that execution is attributed to the new workload. Further in this context switch process, if the scheduler intends some of its execution to be attributed to neither the outgoing workload nor the incoming workload, then the scheduler may switch to a new `sqoscfg` configuration that is different from that of either of the workloads for the duration of such execution.

7.3. QoS configurations for virtual machines

Usually for virtual machines the resource allocations are configured by the hypervisor. Usually the Guest OS in a virtual machine does not participate in the QoS flows as the Guest OS does not know the physical capabilities of the platform or the resource allocations for other virtual machines in the system.

If a use case requires it, a hypervisor may virtualize the QoS capability to a VM by virtualizing the memory-mapped CBQRI register interface and virtualizing the virtual-instruction exception on access to `sqoscfg` CSR by the Guest OS.

If the use of directly selecting among a set of `RCID` *and/or* `MCID` *by a VM becomes more prevalent and the overhead of virtualizing the* `sqoscfg` *CSR using the virtual instruction exception is not acceptable then a future extension may be introduced where the* `RCID` */*`MCID` *attempted to be written by VS mode are used as a selector for a set of* `RCID`/`MCID` *that the hypervisor configures in a set of HS mode CSRs.*

7.4. QoS Identifiers for supervisor and machine mode

The `RCID` and `MCID` configured in `sqoscfg` also apply to execution in S/HS-mode, but this is typically not an issue. Usually, S/HS-mode execution occurs to provide services, such as through an ABI, to software executing at lower privilege. Since the S/HS-mode invocation was to provide a service for the lower privilege mode, the S/HS-mode software may opt not to modify the `sqoscfg` CSR.

Similarly, The `RCID` and `MCID` configured in `sqoscfg` also apply to execution in M-mode, but this is typically not an issue either. Usually, M-mode execution occurs to provide services, such as through the SBI interface, to software executing at lower privilege. Since the M-mode invocation was to provide a service for the lower privilege mode, the M-mode software may opt not to modify the `sqoscfg` CSR.

If separate `RCID` and/or `MCID` are needed during software execution in M/S/HS-mode, then the M/S/HS-mode software may update the `sqoscfg` CSR and restore it before returning to lower privilege mode execution. The statistical nature of QoS capabilities means that the brief duration, such as the few instructions in the M/S/HS-mode trap handler entry point, during which the trap handler may execute with the `RCID` and/or `MCID` established for lower privilege mode operation may not have a significant statistical impact.

Bibliography

[1] K. Du Bois, S. Eyerman, and L. Eeckhout, “Per-Thread Cycle Accounting in Multicore Processors,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, Jan. 2013, doi: 10.1145/2400682.2400688.

[2] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, “Heracles: Improving Resource Efficiency at Scale,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, New York, NY, USA, 2015, pp. 450–462, doi: 10.1145/2749469.2749475.

[3] “RISC-V Quality-of-Service (QoS) Identifiers.” [Online]. Available: github.com/riscv/riscv-ssqosid.

[4] “RISC-V IOMMU Architecture Specification.” [Online]. Available: github.com/riscv-non-isa/riscv-iommu.

[5] N. C. Thornock and J. K. Flanagan, “Facilitating level three cache studies using set sampling,” in *2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165)*, 2000, vol. 1, pp. 471–479 vol.1, doi: 10.1109/WSC.2000.899754.