RISC-V Capacity and Bandwidth QoS Register Interface

RISC-V CMQRI Task Group

Version 0.1, 11/2022: 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:

Aaron Durbin, Allen Baum, Allison Randal, Ambika Krishnamoorthy, David Kruckemeyer, David Weaver, Derek Hower, Eric Shiu, Greg Favor, Vedvyas Shanbhogue

Chapter 1. Introduction

Quality of Service (QoS) is the minimal end-to-end performance that is guaranteed in advance by a service level agreement (SLA) to an workload. 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. Further 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 an workload becomes less deterministic or even non-deterministic as 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.

A workload may be a single application, a group of application, a virtual machine, a group of virtual machines, or some combination of above.

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 workload's cache capacity usage, memory bandwidth usage, interconnect bandwidth usage, etc. through a resource allocation extension. The resource allocation extension 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 colocating workloads while minimizing the interference caused by one workload to another [2].

Effective use of the resource allocation extension requires hardware to provide a resource monitoring extension 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 extensions and to establish resource allocations for the workload using the resource allocation extensions.

The allocation may be in the form of capacity or bandwidth depending on the type of resource. For caches 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 typically generate a 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) extension specifies:

1. QoS identifiers to identify workloads that originate requests to the shared resources. These QoS identifiers include an identifier for resource allocation configuration 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 specfies the mechanism to associate the identfiers 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 capacity controllers such as shared caches and directories. 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 bandwidth controllers such as interconnect and memory controllers. The bandwidth allocation register interface is specfied 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 the CBQRI extension 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, 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`. The implementation may observe the 8-byte access as two 4-byte accesses. A 4-byte access to a register must be single-copy atomic.

> If an implementation may observe a 8-byte register access as two 4-byte accesses then such implementations must preserve the semantics of the 8-byte access and must cause any side effects only after both accesses have been observed.

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 are expected to 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.

The CBQRI 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, the CBQRI extension defines a mechanism to configure resource usage limits, in the form of capacity or bandwidth, and optionally a priority for an `RCID` in the resource controllers that control accesses to such shared resources.

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

2.1. Associating `RCID` and `MCID` with requests

2.1.1. RISC-V hart initiated requests

The `sqoscfg` CSR is a 32-bit S/HS-mode read/write register to configure a resource control ID (`RCID`) and a monitoring counter ID (`MCID`). The `RCID` and `MCID` accompany each request made by the hart to shared resources such as interconnects, caches, memory, etc.

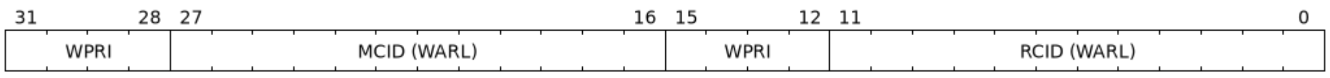

Figure 1. `sqoscfg` register for RV32 and RV64

The type of request made to the shared resource controller depends on the type of shared resource. In case of resources such as caches or memory these may be a memory access request. In case of resources such as CPU dispatch slots and retirement bandwidth the request may be to allocate such resources for execution.

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.).

Access to `sqoscfg` when `V=1` causes a virtual-instruction exception.

At reset it is suggested that the `RCID` field of `sqoscfg` be set to 0 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.

The value of `MCID` at reset, unlike the `RCID`, does not affect functional behavior. Implementations may choose a convenient legal value for the `MCID` reset value.

The `RCID` and `MCID` configured in the `sqoscfg` CSR apply to all privilege modes of software execution on that hart.

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`. The group being associated with a common `RCID` thus shares a common pool of resource allocations. The resource allocation 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 use 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. The usage motivates separate IDs for allocation and monitoring.

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

The `RCID` and `MCID` configured in `sqoscfg` also apply to execution in S/HS-mode but is typically not an issue. Usually, S/HS-mode execution occurs to provide services, such as through the SBI, 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 not modify the `sqoscfg` CSR.

If a use case requires use of separate `RCID` and/or `MCID` for software execution in S/HS-mode, then the S/HS-mode SW may update the `sqoscfg` CSR and restore it prior to returning to the lower privilege mode execution.

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 using the virtual-instruction exception on access to `sqoscfg` CSR.

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.

A Hypervisor may cause a context switch from one virtual machine to another. The context switch usually involves saving the context associated with the VM being switched away from and restoring the context of the VM being switched to. Such context switch may be invoked in response to an explicit call from the VM (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 hypervisor may want to execute with the `sqoscfg` configurations of the VM being switched away from such that the execution is attributed to the VM being switched from and then prior to executing the context switch code associated with restoring the new VMs context first switch to the `sqoscfg` appropriate for the new VM being switched to such that all of that execution is attributed to the new VM. Further in this context switch process, if the hypervisor intends some of the execution to be attributed to neither the outgoing VM nor the incoming VM, then the hypervisor may switch to a new configuration that is different from the configuration of either of the VMs for the duration of such execution. QoS extensions are statistical in nature and the small duration, such as the few instructions in the hypervisor trap handler entrypoint, for which the HS-mode may execute with the RCID/ MCID established for lower privilege mode operation may not be statistically significant.

The RCID and MCID configured in sqoscfg also apply to execution in M-mode but is typically not an issue. 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 not modify the sqoscfg CSR. If a use case requires use of a separate RCID and/or MCID for software execution in M-mode, then the M-mode SW may update the sqoscfg CSR and restore it prior to returning to lower privilege mode execution.

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 the QoS indetifiers. The RISC-V IOMMU specification may support a future standard extension to support this configuration.

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. The CBQRI extension 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.

Value	Name	Description
6-7	Custom	Designated for custom use.

If a request is received with an unsupported `AT` value then the resource controller behaves as if the `AT` value was 0.

Chapter 3. Capacity-controller QoS Register Interface

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

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

The size of the `cc_block_mask` register is determined by the `NCBLKS` field of the `cc_capabilities` register.

The reset value is 0 for the following registers fields.

- `cc_mon_ctl` - `BUSY` and `STATUS` fields
- `cc_alloc_ctl` - `BUSY` and `STATUS` fields

The reset value is `UNSPECIFIED` for all other registers and/or 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 allocated to each supported access-type for `RCID=0`. The capacity allocation for all other `RCID` values is `UNSPECIFIED`. The behavior of handling a request with a non-zero `RCID` value before configuring the capacity controller with capacity allocation for that `RCID` is `UNSPECIFIED`.

3.1. Capabilities (`cc_capabilities`)

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

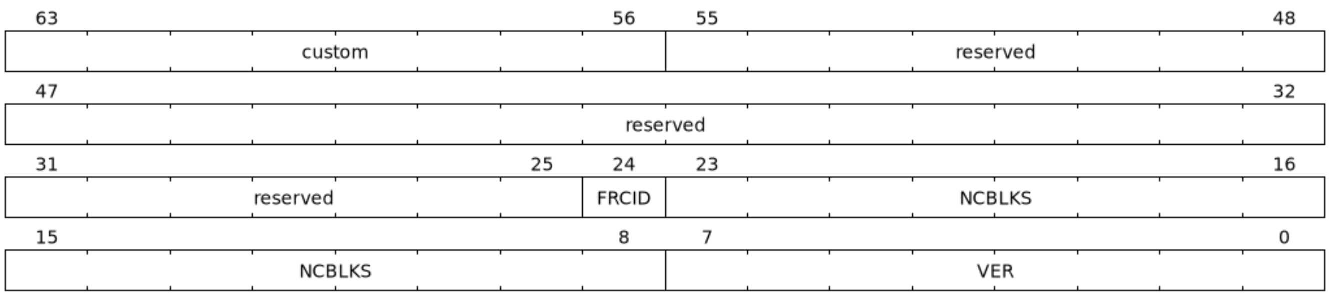

Figure 2. Capabilities register fields

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 multiples of an allocatable capacity block.

For example, a cache controller that supports capacity allocation by ways may report the number of ways as the number of allocatable capacity blocks.

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

3.2. 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.

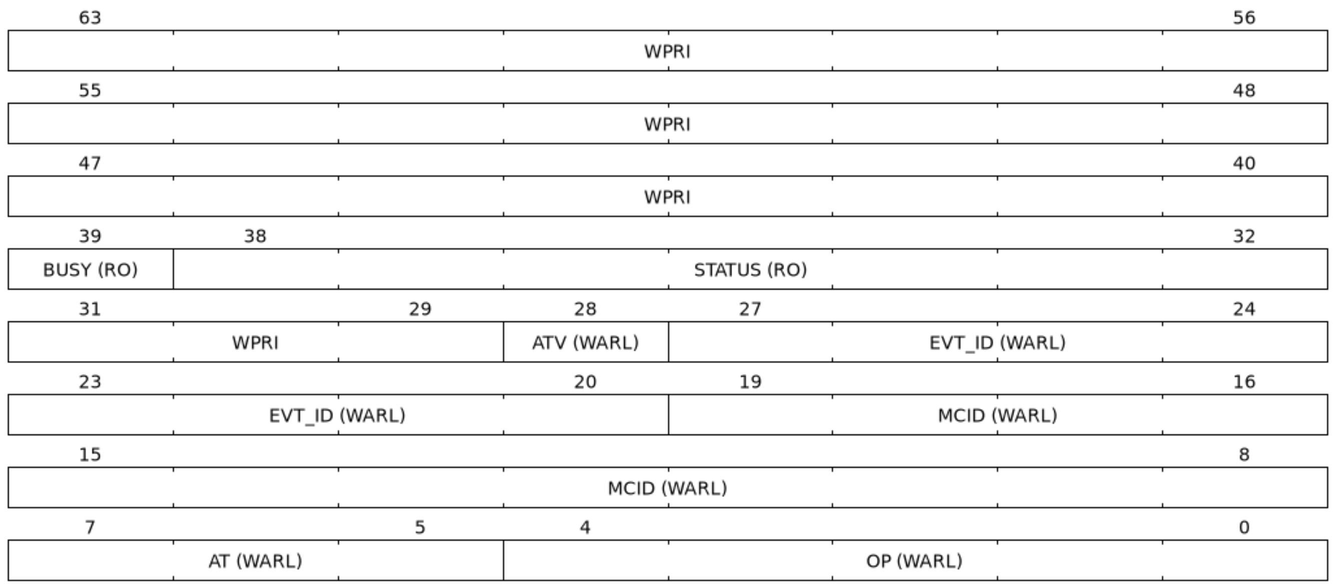

Figure 3. Usage monitoring control (cc_mon_ctl)

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

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 is used to program the access-type to count.

When the EVT_ID for a MCID is selected as 0, the counter retains its current value but stops counting. When EVT_ID is programmed with a non-zero and legal value, the counter resets to 0 and starts counting the programmed events for requests with matching MCID and AT(if ATV is 1). When ATV is 0, the counter counts requests with all access-types and the AT value is ignored.

A controller that does not support monitoring by access-type may hardwire the ATV and AT fields to 0.

Table 3. 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.

Table 4. 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 capacity block. The counter is decremented by 1 when that capacity block is de-allocated.
—	2-127	Reserved for future standard use.
—	127-128	Designated for custom use.

When the cc_mon_ctl 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 BUSY bit to 1 indicating the controller is performing the requested operation. When the BUSY bit reads 0 the operation is complete and the STATUS field provides a status value (Table 5) of the requested operation.

Table 5. cc_mon_ctl.STATUS field encodings

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

Behavior of writes to the cc_mon_ctl when BUSY is 1 is UNSPECIFIED. Some implementations may ignore the second write and others may perform the operation determined by the second write. Software must verify that BUSY is 0 before writing cc_mon_ctl.

3.3. Monitoring counter value (`cc_mon_ctr_val`)

The `cc_mon_ctr_val` is a read-only register that holds a snapshot of the counter requested by `READ_COUNTER` operation.

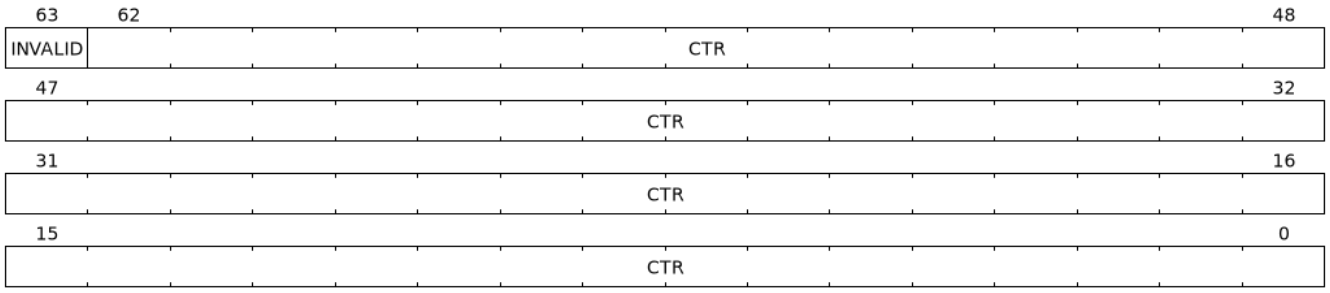

Figure 4. Usage monitoring counter value (`cc_mon_ctr_val`)

The counter is valid if the `INVALID` field is 0. The counter may be marked `INVALID` if it underflows or the controller for `UNSPECIFIED` reasons determine the count to be not valid. The counters marked `INVALID` may become valid in future.

A counter may underflow when capacity blocks are de-allocated following a reset of the counter to 0. This may be due to the `MCID` being reallocated to a new workload while the capacity controller still holds capacity blocks allocated by the workload to which the `MCID` was previously allocated. The counter value should typically stabilize to reflect the usage of the new workload after the workload has executed for a short duration following the counter reset.

Some implementations may not store the `MCID` of the request that caused the capacity block to be allocated with every capacity block 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 blocks.

Techniques such as set-sampling in caches have been shown to provide statistically accurate cache occupancy information with a relatively small sample size such as 10% [3].

When the controller has not observed enough samples to provide an accurate value in the monitoring counter the controller may report the counter as being `INVALID` till more accurate measurements are available.

3.4. Capacity allocation control (`cc_alloc_ctl`)

The `cc_alloc_ctl` register is used to control allocation of capacity to a `RCID` per `AT`. If a controller does not support capacity allocation then the register is read-only zero. If the controller does not support capacity allocation per access-type then the `AT` field is read-only zero.

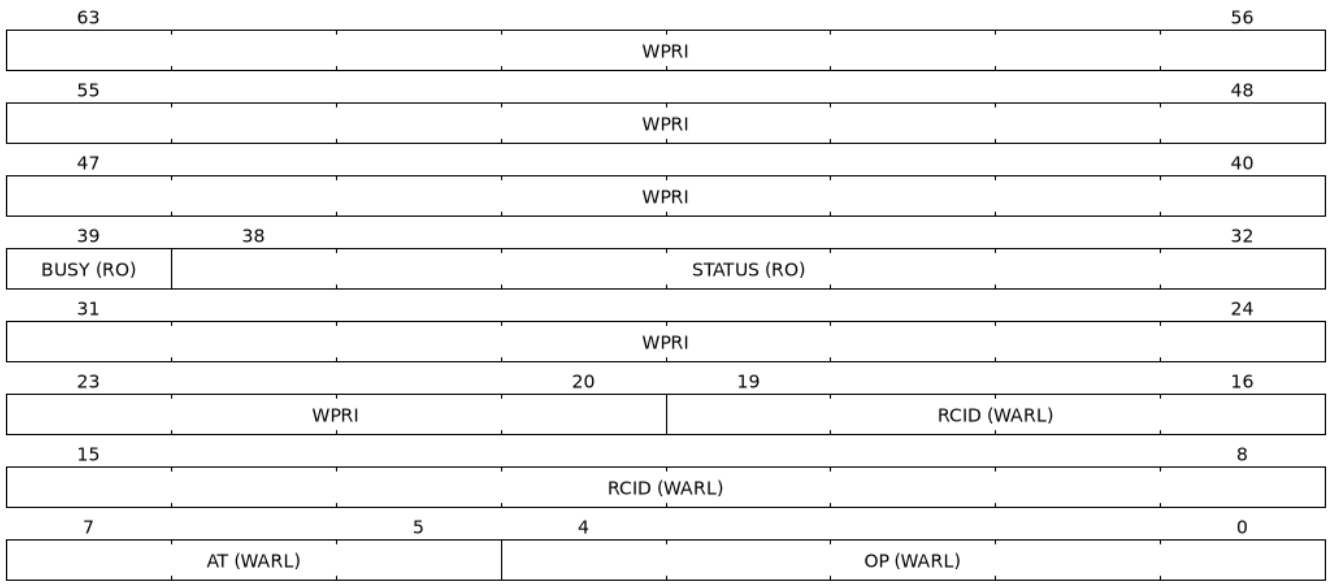

Figure 5. Capacity allocation control (`cc_alloc_ctl`)

The `OP` field is used to instruct the capacity controller to perform an operation listed in Table 6. The `cc_alloc_ctl` register is used in conjuction with the `cc_block_mask` register to perform capacity allocation operations. When the requested operation uses the operands configured in `cc_block_mask`, software must first program the `cc_block_mask` register with the operands for the operation before requesting the operation.

Table 6. Capacity allocation operations (`OP`)

Operation	Encoding	Description
—	0	Reserved for future standard use.
`CONFIG_LIMIT`	1	The `CONFIG_LIMIT` operation is used to establish a limit for capacity allocation for requests by `RCID` and of access-type `AT`. The capacity allocation is specified in `cc_block_mask` register.
`READ_LIMIT`	2	The `READ_LIMIT` operation is used to read back the previously configured allocation limits for requests by `RCID` and of type `AT`. The current configured allocation limit is written to `cc_block_mask` register on completion of the operation.
`FLUSH_RCID`	3	The `FLUSH_RCID` operation requests the controller to deallocate the capacity allocated for use by the specified `RCID` for access-type specified by `AT`. The `cc_block_mask` register is not used for this operation.
—	4-23	Reserved for future standard use.
—	24-31	Designated for custom use.

Capacity controllers enumerate the allocatable capacity blocks in `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 to be allocated.

A capacity allocation must be configured for each supported access-type by the controller. Identical limits may be configured for two or more access-types if different capacity allocation per access-type is not required. The behavior is `UNSPECIFIED` if capacity is not allocated for each access-type supported by the controller.

A cache controller that supports capacity allocation indicates the number of allocatable capacity blocks in `cc_capabilities.NCBLKS` field. For example, 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 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

The `FLUSH_RCID` operation may incur long latency to complete. New requests to the controller by the `RCID` being flushed are allowed. The controller is allowed to deallocate capacity blocks that were 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 may 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 blocks allocated by the old workload and thus for a short warmup duration the capacity controller may be enforcing capacity 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 may be used to flush and evict capacity allocated to the old workload.

When the `cc_alloc_ctl` 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 `BUSY` bit to 1 indicating the controller is performing the requested operation. When the `BUSY` bit reads 0 the operation is complete and the `STATUS` field provides a status value (Table 7) of the requested operation.

Table 7. `cc_alloc_ctl.STATUS` field encodings

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

Behavior of writes to the cc_alloc_ctl or cc_block_mask when BUSY is 1 is UNSPECIFIED. Some implementations may ignore the second write and others may perform the operation determined by the second write. Software must verify that BUSY is 0 before writing cc_alloc_ctl or cc_block_mask.

3.5. Capacity block mask (cc_block_mask)

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

The register has NCBLKS Section 3.1 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 width in bits is determined as

$$BMW = \frac{NCBLKS + 63}{64} x64$$

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

To configure capacity allocation limit for an RCID and AT, the cc_block_mask is first programmed with a bit-mask identifying the capacity blocks to be allocated and then the cc_alloc_ctl register 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 a capacity block if the capacity block mask programmed for RCID and AT associated the request has the corresponding bit set to 1 in the capacity block mask. At least one capacity block must be allocated using cc_block_mask when allocating capacity. Allocating overlapping capacity block masks among RCID and AT is allowed.

A set-associative cache controller that supports capacity allocation by ways may advertize NCBLKS to be the number of ways per set in the cache. Allocating capacity in such a cache for an RCID and AT involves selecting a subset of ways and programming the mask of the selected ways in cc_block_mask when requesting the CONFIG_LIMIT operation.

To read out the capacity allocation limit for an RCID and AT, the READ_LIMIT operation is requested using cc_alloc_ctl. On successful completion of the operation the cc_block_mask holds the configured capacity 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 register programming 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 registers fields.

- `bc_mon_ctl` - `BUSY` and `STATUS` fields
- `bc_alloc_ctl` - `BUSY` and `STATUS` fields

The reset value is `UNSPECIFIED` for all other registers and/or fields.

The bandwidth controllers at reset allocate all available bandwidth to `RCID` value of 0. When the bandwidth controller supports bandwidth allocation per access-type, one of the supported access-types for `RCID=0` is allocated all available bandwidth and all other access-types share the bandwidth allocation with that access-type. The bandwidth allocation for all other `RCID` values is `UNSPECIFIED`.

4.1. Capabilities (`bc_capabilities`)

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

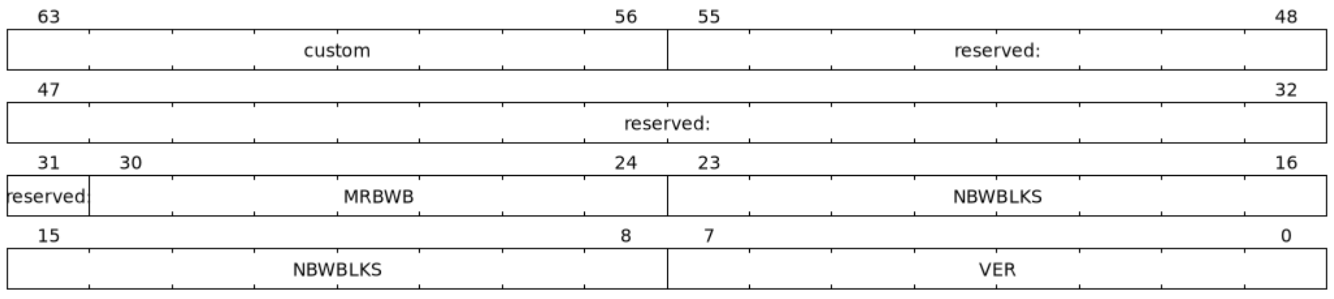

Figure 6. Capabilities register fields

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 `NBWBLKS` field holds the total number of available bandwidth blocks in the controller. The bandwidth represented by an available bandwidth block is UNSPECIFIED. The bandwidth controller supports reserving bandwidth in multiples of a bandwidth block and thereby support proportional allocation of bandwidth.

Bandwidth controllers may limit the maximum bandwidth that may be reserved by allocating to workloads to a fraction of the NBWBLKS. The MRBWB field holds a number between 1 and 100 to represent this fraction. This number is termed as MaxRsvdBWBlocks.

$$MaxRsvdBWBlocks = \text{round}\left(\frac{MRBWBxNBWBLKS}{100}\right)$$

The bandwidth controller needs to meter the bandwidth usage by a workload to determine if its exceeding its allocations and if required invoke necessary measures to throttle the bandwidth usage by the workload. The instantaneous bandwdith use by a workload may thus undershoot or overshoot the configured allocation. The QoS capabilities are statistical in nature and are typically designed to enforce the configured bandwidth over larger time windows. Not allowing all available bandwidth blocks to be reserved for allocation may allow the bandwidth controller with handle such transient inaccuracies.

4.2. Usage monitoring control (bc_mon_ctl)

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

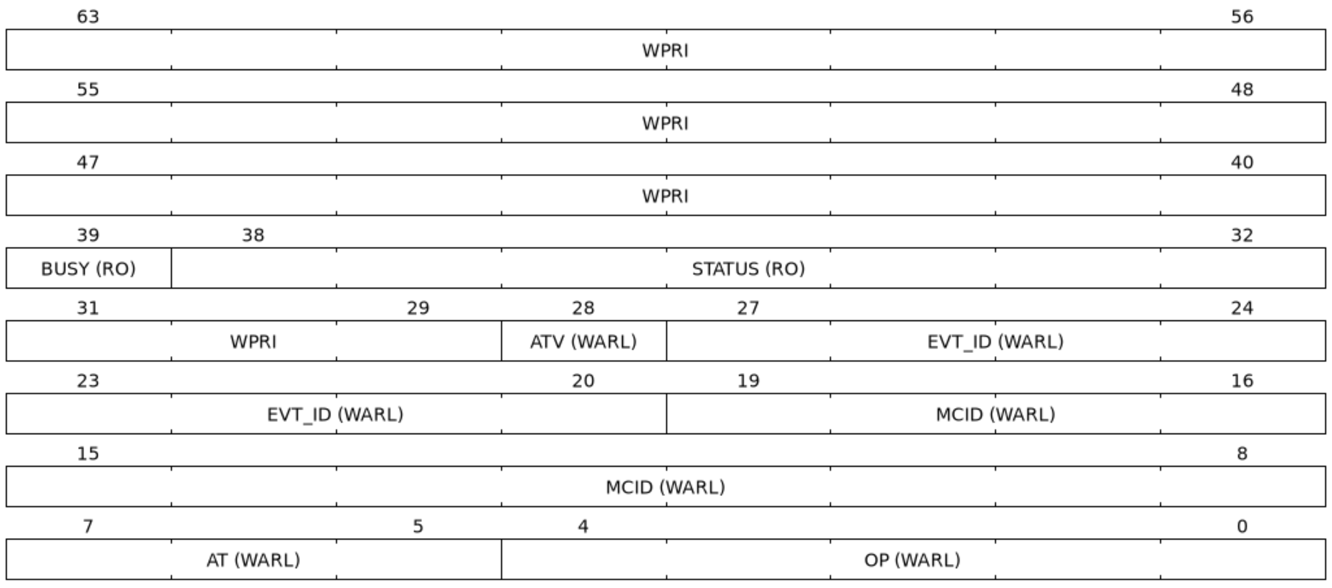

Figure 7. Usage monitoring control (bc_mon_ctl)

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

The EVT_ID field and is to program the identifier of the event to count in the counter selected by MCID. The AT field is used to program the access-type to count.

When the EVT_ID for a MCID is selected as 0, the counter retains its current value but stops counting.

When EVT_ID is programmed with a non-zero and legal value, the counter resets to 0, the OVF bit is cleared to 0, and the counter starts counting the programmed events for requests with MCID and AT (if ATV is 1). When ATV is 0, the counter counts requests with all access-types and the AT value is ignored.

A controller that does not support monitoring by access-type may hardwire the ATV and AT fields to 0.

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.
—	24-31	Designated for custom use.

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.
—	127-128	Designated for custom use.

When the bc_mon_ctl 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 BUSY bit to 1 indicating the controller is performing the requested operation. When the BUSY bit reads 0 the operation is complete and the STATUS field provides a status value (Table 11) of the requested operation.

Table 11. bc_mon_ctl.STATUS field encodings

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

Behavior of writes to the bc_mon_ctl when BUSY is 1 is UNSPECIFIED. Some implementations may ignore the second write and others may perform the operation determined by the second write. Software must verify that BUSY is 0 before writing bc_mon_ctl.

4.3. Monitoring counter value (bc_mon_ctr_val)

The bc_mon_ctr_val is a read-only register that holds a snapshot of the counter requested by READ_COUNTER operation.

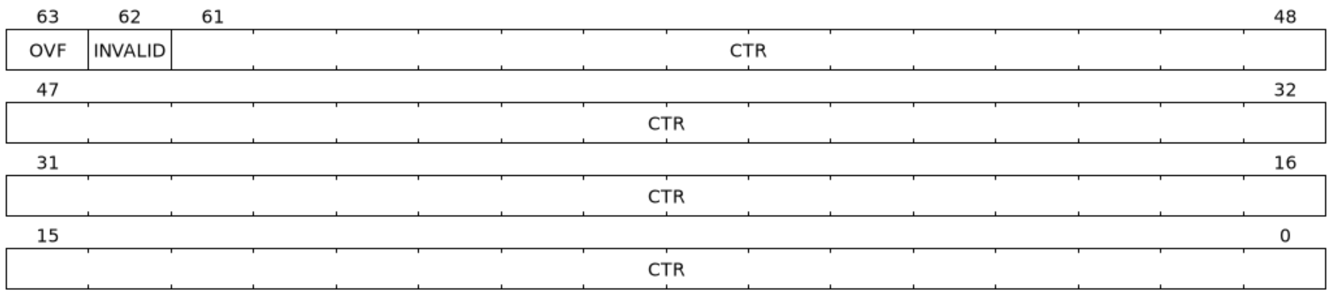

Figure 8. Usage monitoring counter value (bc_mon_ctr_val)

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

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

The counter provides the byte transferred by requests matching the EVT_ID as the requests 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. If the value of the counter at time T1 was B1 and at time T2 is B2 then the bandwidth is as follows. The frequency of the time source is T_{freq}.

$$Bandwidth = T_{freq} \times \frac{B2 - B1}{T2 - T1}$$

The width of the counter is UNSPECIFIED.

The width of the counter is UNSPECIFIED but is recommended to be wide enough to not cause more than one overflow per sample when sampled at a frequency of 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 allocation of bandwidth to a RCID per AT. If a controller does not support capacity allocation then the register is read-only zero. If the controller does not support capacity allocation per access-type then the AT field is read-only zero.

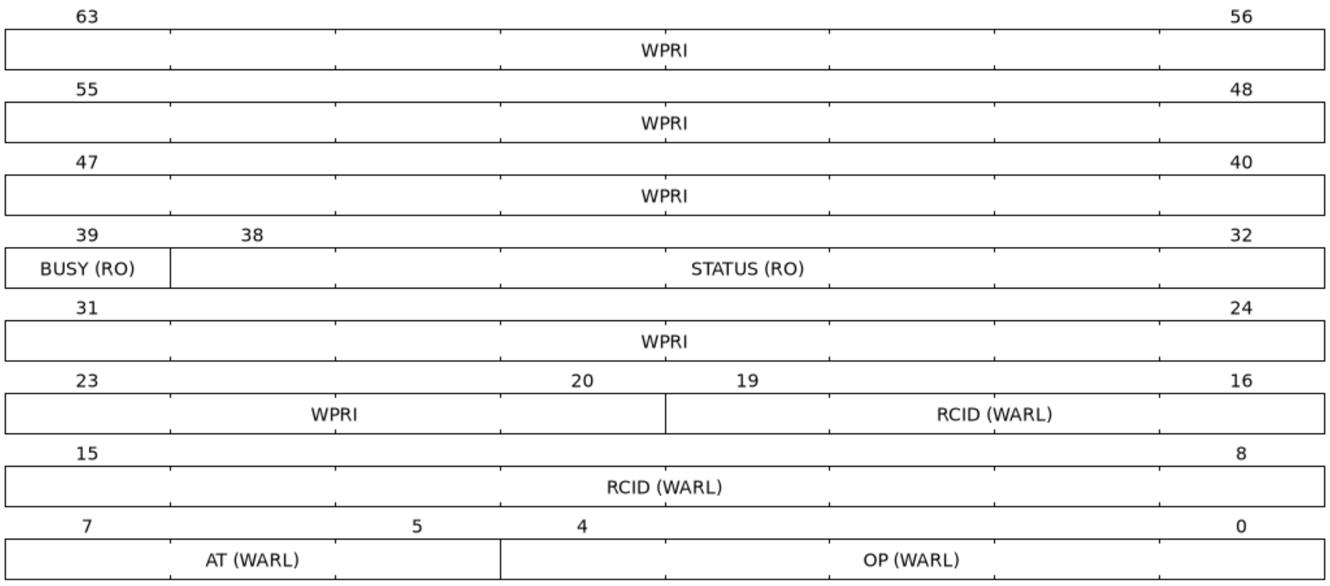

Figure 9. Bandwidth allocation control (bc_alloc_ctl)

The OP field is used to instruct the bandwidth controller to perform an operation listed in Table 12. The bc_alloc_ctl register is used in conjuction with the bc_bw_alloc register to perform bandwidth allocation operations. When 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	The CONFIG_LIMIT operation is used to establish 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	The READ_LIMIT operation is used to read back the previously configured bandwidth allocation for requests by RCID and of type AT. The current configured allocation is written to bc_bw_alloc register on completion of the operation.

Operation	Encoding	Description
—	3-23	Reserved for future standard use.
—	24-31	Designated for custom use.

A bandwidth allocation must be configured for each supported access-type 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 configured to share the allocation with the default access-type. The behavior is `UNSPECIFIED` if bandwidth is not allocated for each access-type supported by the controller.

When the `bc_alloc_ctl` 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 `BUSY` bit to 1 indicating the controller is performing the requested operation. When the `BUSY` bit reads 0 the operation is complete and the `STATUS` field provides a status value (Table 13) of the requested operation.

Table 13. `bc_alloc_ctl.STATUS` field encodings

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

4.5. Bandwidth allocation (`bc_bw_alloc`)

The `bc_bw_alloc` is used to program reserved bandwidth blocks (`Rbwb`) for an `RCID` for requests of access-type `AT`. The bandwidth is allocated in multiples of bandwidth blocks and the value in `Rbwb` must be at least 1 and must not exceed `MaxRsvdBWBlocks` else the operation fails with `STATUS=5`. The sum of `Rbwb` allocated across all `RCID` must not exceed `MaxRsvdBWBlocks` else the operation fails with `STATUS=5`.

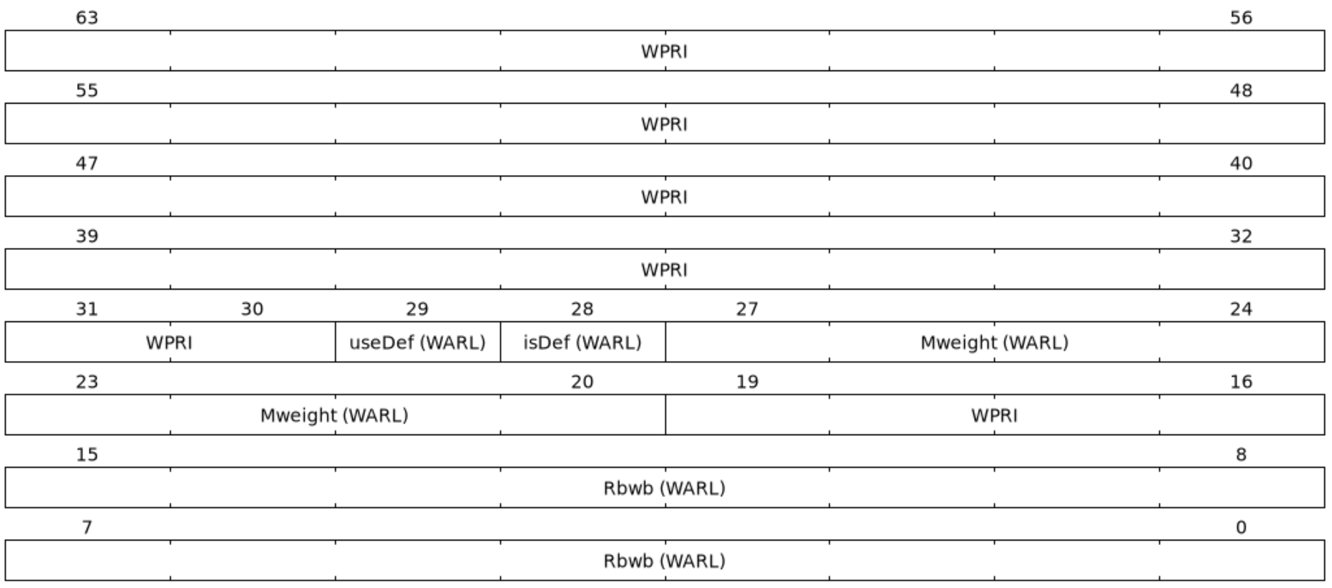

Figure 10. Bandwidth allocation (`bc_bw_alloc`)

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 is `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 then additional bandwidth may be made available to `RCID` that contend for that bandwidth. The non-reserved or unused bandwidth is proportionately shared by the contending RCIDs using the configured `Mweight`. The `Mweight` parameter is a number between 0 and 255. A larger weight implies a greater fraction of the bandwidth. The sharing of non-reserved bandwidth is not differentiated by access-type. 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. The share of unused bandwidth made available to `RCID=x` when it contents with another `RCID` is determined by the `Mweight` of `RCID=x` divided by the sum of `Mweight` of all other contending `RCID`. This ratio `P` is as follows:

$$P = \frac{Wweight_x}{\sum_{r=1}^{r=n} Mweight_r}$$

The bandwidth enforcement is typically work-conserving and allows unused bandwidth to be used by requestors even if they have consumed their `Rbwb`.

When contending for unused bandwidths the weighted share is typically computed among the `RCIDs` that are actively generating requests in that accounting interval.

When `isDef` is 1, the bandwidth allocation programmed for that access-type is considered the default bandwidth allocation. If unique bandwidth allocation is not required for an access-type then the `useDef` may be used to indicate that requests of this access-type share bandwidth allocated for the default access-type. Only one access-type may be designated as the default access-type. The `isDef` and `useDef` fields are reserved if the controller does not support bandwidth allocation per access-type. If more than one access-types have been configured with `isDef` programmed to 1 then the implementation may associate an access-type programmed with `useDef` with any of those but the behavior is otherwise well defined. If an access-type is configured with `useDef` as 1 but no other access-type is configured to be the default access-type then the implementation may select any of the access-types as the default access-type but the behavior is otherwise well defined.