# Functional Description: Coherent Components

## Coherency IP Components

There are three coherency IP modules that are included in Gemini in order to handle the coherency functions of the chip. These IP modules must be instantiated as agents within the system and connected to the NoC like a typical agent. This section will describe these modules and their expected usage.

## CCC (Cache Coherency Controller)

The CCC is the primary coherency control module. This is where coherent reads are sent to perform coherency lookups and to issue snoops if required. Since coherent requests are sent to the CCC before being sent to memory, the CCC should be instantiated near the most latency sensitive coherent agents. Typically, these will be the CPUs.



Figure 5: CCC Port Diagram

The figure above shows the CCC with two ports. One is the slave port, where the CCC accept coherent requests from CPUs or other coherent agents, and one is the master port, where the CCC issues reads or writes to memory or to a cache in order to satisfy a coherent request.

The slave port protocol for the CCC is called CCCS. The master port protocol is CCCM. The slave port should be configured to be near the latency sensitive coherent requesters such as CPU. The master port should be configured to be near the destination cache or memory.

### Multiple CCCs

In some Gemini systems, multiple coherency controllers can be instantiated. Each CCC will be responsible for a subset of the address space. The assignment of address responsibility has significant flexibility. Each can be assigned their own range or ranges of addresses, or they can share a set of address regions and slice the range between them. For instance, two CCCs could split an address range into even and odd cache lines, with one CCC handling the even lines and one CCC handling the odd lines.

There are several reasons for instantiating multiple coherency controllers.

### Increased Coherent Bandwidth

A single coherency controller has limitations on the bandwidth it can handle. The data interface width at the CCC is 32 bytes wide. While read and write channels can both be active at the same time, the data width limit puts an upper bound on total number of cache lines that can be processed by a single controller. At 1GHz, this puts an upper bound of 32GB/s of read and 32GB/s of write bandwidth on a single CCC.

A different limitation of bandwidth is that coherent requests must access the directory RAM that store the coherent state of the system. Since this is a single-ported RAM, the access of the RAM can be a limit. Cache coherent reads can take two accesses (one for the lookup, and one for saving the new state). Writebacks/Evicts can also require two accesses (one for a lookup, one to save the new state). IO coherent requests will typically only have one access for the lookup but can require an additional access if snoops are sent and state is modified.

Additional coherency controllers increase bandwidth approximately linearly. Two controllers will have 2x the bandwidth (2x the port bandwidth, and 2x the directory bandwidth). This assumes traffic is somewhat uniformly split between the two CCCs, which can be done by using lower order index bits.

### Reducing Directory Size to Decrease Latency

The Directory is a RAM, and so will have an increased latency with more storage. Creating additional CCCs can split the directory size, potentially allowing a decrease in latency or an increase in bandwidth.

### Minimizing Latency with Location

In some systems, multiple coherency controllers may be used by placing them in different locations within the chip and assigning address spaces so that physically close agents will send the bulk of their traffic to the nearby coherency controller. This produces a NUMA (Non-Uniform Memory Access) type of chip design. Since the CCCs are located closer to the agents accessing them, system latency can be reduced.

## IOCB (IO Coherency Bridge)

The IOCB is an IP block that is responsible for IO coherent accesses (WriteUnique, WriteLineUnique, and ReadOnce). It accepts requests from ACE-lite, ACE-lite+DVM and ACE-lite Converted bridges. It can be added with the *add\_iocb* command in NocStudio.

Non-cached traffic is traffic issued by an agent that doesn’t use a cache. This includes non-coherent traffic as well as IO coherent traffic. One of the key issues with these non-cached requests is the need for ordering of the requests. Sharing of data by multiple agents utilizes various sharing models that rely on the order in which request are completed. For instance, writes frequently need to be ordered to ensure that prior writes are visible to an agent before later writes. If a master writes data and then stores a flag to indicate that the data is valid, an agent that reads the flag must be sure that data is visible.

There is frequently a tradeoff between satisfying ordering requirements and achieving high bandwidth. This is because a common method of enforcing ordering between two requests is to wait for the first to complete before issuing the second. If ordering requirements are common, this can have a significant reduction in total bandwidth.

The IOCB is built to enable high bandwidth while allowing frequent ordering requirements. It is an IO coherency accelerator. It takes advantage of the inherent parallelism of coherency to prefetch coherent permissions for the requests. Once permissions are granted, it can commit the operations with high bandwidth without the need for serialization.



Figure 6: IOCB Port Diagram

The diagram above shows the IOCB agent and its two ports. The slave port accepts requests from IO coherent masters (ACE-lite, ACE-lite+DVM, ACE-lite converted). The slave port protocol is specified as IOCBS. The master port issues coherent requests to and from the CCC, and the protocol is specified as IOCBM.

The IOCB data ports can be configured to be either 32 bytes or 64 bytes wide. At 1GHz, this is 32GB/s or 64GB/s respectively of simultaneous read or write bandwidth.

### Multiple IOCBs

A system can have more than one IOCB, just as it can have more than one CCC. There is no relationship between these, however. A system could have multiple CCCs and only one IOCB, or a single CCC and multiple IOCBs, or any combination.

While a CCC is assigned to an address range, the IOCB is assigned to masters. This means a master only ever connects to a single IOCB. If multiple IOCBs are added to the system, each master will be assigned to only one of them.



Figure 7: Multi-IOCB diagram

As the figure above shows, each master only sends traffic to a single IOCB, even when there are multiple IOCBs in the system. However, multiple masters can send to the same IOCB. Sharing the IOCB reduces total hardware costs and allows dynamic sharing of resources.

### Selecting a Target IOCB

Each ACE-lite master (including ACE-lite+DVM, and ACE-lite converted) must connect to an IOCB to process WriteUnique, WriteLineUnique, and ReadOnce requests. A single IOCB can be shared by multiple masters. Multiple IOCBs may be used for increased bandwidth, lower latency, or other system tradeoffs. If so, each master needs to have its target IOCB specified.

The bridge property *cc\_target\_iocb* can be used to specify the target IOCB. By default, this value is marked as unassigned. In a system with a single IOCB, the agents will automatically be mapped to that IOCB. If multiple IOCBs are present, this property must be set for each master.

## DVM (Distributed Virtual Memory Controller)

Distributed Virtual Memory is an independent communication protocol that utilizes the ACE communications channels. The DVM module can be instantiated using the *add\_dvm* command in NocStudio. Details of the DVM protocol can be found in the AMBA ACE specification.

DVM is used to broadcast and synchronize control packets for TLB invalidations, Instruction Cache invalidations, and similar requests. Generally, it is used to maintain non-coherent hardware storage structures when the underlying data changes.

The interconnect has two primary functions related to DVM. The first is that when a DVM message is sent from a master, it is broadcast to all other DVM-enabled agents (ACE or ACE-lite+DVM). The broadcast is sent in the form of a snoop request to these other masters. The second function is a DVM synchronization processes, which includes sending synchronization snoops, gathering completion requests from the DVM agents, and eventually signaling back to the synchronizing master that the request has completed.

To satisfy the DVM protocol requirements, a single DVM agent must be instantiated within the network. This agent broadcasts the snoops, gathers responses, and replies to the original master. It also tracks synchronization requests and verifies that all DVM completion requests have been issued



Figure 8: DVM module port diagram

As seen in the diagram above, the DVM has a single port which accepts DVM requests and issues corresponding snoops as needed. This slave port protocol is specified as DVMS.

Unlike CCC or IOCB, only a single DVM module can exist.

### Systems without DVM

It is possible to construct a coherent system without DVM. This may be desirable when the coherent agents do not support the DVM protocol, so adding a DVM module would be unnecessary. To ensure that a lack of DVM is intentional, NocStudio requires the mesh property *cc\_dvm\_support* to be set to ‘no’. If this property is not explicitly set to ‘no’, the NoC generation will signal an error indicating that the DVM module was not added.

### ACE Masters without DVM

While ACE masters are assumed to support DVM, it is possible to specify that an ACE master does not support DVM. A NocStudio bridge property for the ACE master bridge called *acem\_dvm\_support* can be set to no. When disabled, the ACE master will not receive DVM snoops, cannot generate DVM requests, and will not power up the DVM when it performs a coherency connect.

## Directory Configurations

Gemini utilizes a directory-based coherency protocol. This means that the coherency controller(s) have a storage structure that tracks which lines are in the various coherent caches. A directory is a kind of snoop-filter, and its job is to allow the coherency controller to locally determine the state of a cache line without sending snoops to the caches. This allows lower latency accesses, reduced network bandwidth, reduced snoop bandwidth at the masters, and higher peak bandwidth of the system.

The directory stores cache line addresses, state information about each address, as well as information about which of the coherent caches the line is present in. The directory does not store cache line data. It is not a cache. It only contains address and state information.

The directory is built to use single-ported RAM arrays for the storage. The RAM implementation is not included with Gemini and must be provided by the customer. The RAM design details can be supplied to NocStudio during the Gemini construction so that the final RAM design can be swapped in seamlessly later.

### Building Gemini with no Directory

It is possible to build a Gemini system with no directory. In this system, each request would have to issue snoops to every ACE master in the system and wait for snoop responses before a reply can be returned. This can reduce or limit coherent bandwidth and will likely increase latency of requests, but it can save area for customers who need to minimize the footprint of the hardware.

When an ACE master is added in NocStudio, it can be marked as having no snoop filter support using the bridge property *cc\_snoop\_filter\_support*. If snoop filter support is disabled, it means coherent requests will have to send snoops to each master. If all ACE masters are set up that way, no directory would exist at all.

The CCC is designed to be able to send one snoop to a single destination per cycle. This means that the coherent bandwidth can be significantly reduced if the number of ACE masters is large. If a system had 4 ACE masters, it could only process a request ¼ cycles at most, since it would take that long to send all of the snoops. To get more coherent bandwidth, more CCCs can be used. However, there is a finite amount of snoop traffic that an ACE master can receive. It can only accept one snoop per cycle because of the single snoop port. So coherent bandwidth has a fundamental limit with this snooping configuration.

### Mixing Directory And Snoop Agents

Gemini allows a mix of coherent agents that do or do not use the directory. This can lead to more complex design tradeoffs.

When some agents are tracked by the directory, and other agents are snoop-only, the benefits of the directory have the potential to be significantly reduced. A directory provides more coherent bandwidth and lower request latency. But if snoops need to be sent to some agents on every request, latency will go up and bandwidth will be reduced and ultimately limited by the snoop ports. Without additional modifications, this combination would behave poorly.

To avoid these issues, Gemini can support the Inner Shareable and Outer Shareable domains. The behavior of the system obeys the following rules:

Table 2: Inner and Outer Shareable Domain Visibility

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Requesting Agent | Behavior of Different Domain Type | | | |
| Inner Sharable Domain | | Outer Sharable Domain | |
| Directory lookup? | Snoop sending to snoop-only agent? | Directory lookup? | Snoop sending to snoop-only agent? |
| ACE (Directory support) | Yes | No | Yes | Yes |
| ACE (Snoop only) | Yes | Yes | Yes | Yes |

As the table shows, every access will look up the directory, since this incurs no latency or bandwidth cost. The snoop-only agents only get snooped under some conditions. Other requests from snoop-only agents will always send the snoops. But for directory supporting ACE agents, only their Outer Shareable accesses will send snoop. So, directory agents can realize the benefits of a directory even when there are snoop-only agents by accessing regions as Inner Shareable.

#### Possible Uses of Mixing Directory and Snoop-only Agents

There are a lot of different combinations that could be useful for mixing these systems. For instance, a GPU might have a very large coherent cache and keeping directory state of that cache could be expensive. In that configuration, the GPU could be marked as snoop-only. This would allow the GPU to still access coherent memory and view the CPU cache contents, but it would enable the CPUs to ignore the GPU in address regions it knows the CPU is not using.

Another possible use is in a system with an external link to a coherent subsystem. Since the coherent subsystem is not part of the chip and could change over time, it may be desirable to satisfy requests to that external system without directory overhead. But the latency costs for sending snoops may be substantial. Using the Inner Shareable domain would allow local agents to interact quickly while limiting the amount of global communication in the Outer Shareable domain.

### Sizing the Directory

The first step in configuring the directories is to determine their capacity and associativity.

Any line present in a coherent cache must be tracked by the directory. If the directory doesn’t have room for a new address, it will have to evict an existing address. To guarantee inclusivity, this will result in a cache line flush being sent to the caches to remove the evicted address. This can lead to the caches being underutilized. This is called a directory conflict and can result in a reduction in performance due to the underutilization of the caches. To avoid this, the directory needs to be sized appropriately.

The first recommendation is that the capacity of the directory is at least the capacity of the caches has to track. For instance, if there are 2 caches with 1MB each, the directory should be able to track 2MB of total cache capacity. Anything less than the cache capacity will result in underutilization of the caches.

Gemini supports significant heterogeneity in the system. It allows different coherent caches to be sized differently.

For example, one cache might have 2MB, another might have 512KB, while a third only has 128KB. The total cache capacity to be tracked is 2.625MB.

The directory utilizes a set-associative organization. This means it will have a power-of-2 number of sets, with some number of ways per set. This format puts a practical limitation on the granularity of the capacity. To ensure directory capacity >= cache capacity, the directory can be oversized if necessary. Configuring the sets and ways appropriately can minimize how much oversizing is necessary. For instance, a 2MB directory may be organized as 8 ways and 4096 sets. If 2.1 MB is needed, the directory can choose different options, such as 8 ways and 8192 sets (4MB), or 10 ways and 4096 sets (2.5 MB of cache). Varying the number of ways can increase capacity on a non-power-of-2 scale.

The capacity of the caches is not the only factor that should drive the sizing of the directory. Another similar factor arises from the fact that the directory is a shared data structure for all caches.

To truly guarantee that any combination of lines within the caches can fit within the directory, the directory would have to have the same indexing method as well as associativity of all of the caches. For instance, if 4 caches each have 8 ways, the directory would need a minimum of 4x8=32 associative ways to guarantee the lines in the caches can all fit in the directory. With less associativity, some combinations won’t fit in the directory and conflicts can occur.

The Gemini directory is not intended to match the associativity or indexing method of the caches. Instead, it is implemented as a shared structure with limited associativity. It utilizes several innovations to reduce the likelihood of directory conflicts, as well as methods of reducing the cost of these conflicts when they occur. However, since conflicts are still possible, additional steps can be taken to decrease conflicts even more.

One available tool to avoid conflicts is to increase the directory capacity beyond the capacity of the caches. With a directory that has a larger capacity than the caches, there will be unused entries in the directory. This means that when the caches put added pressure on a set in the directory, the directory will be able to absorb additional lines without the need for a conflict. So, to increase performance of the system, the directory can be slightly oversized.

There are two properties to set in order to size the directory. One describes the number of sets of the directory, while the other describes the associativity. Both are properties of the CCC host. The first is “cc\_directory\_associativity” and the second is “cc\_directory\_index\_width”.

The associativity must be between 8-16 and must be a multiple of 2. This is because the directory is organized as 2 tag arrays, with equal number of associative ways.

The index width is the log2 (#sets). A directory with 4096 sets will have an index width of 12 bits.

If not defined, these properties may be automatically set up using the setup\_coherency command, described in a later section of this document.

#### Selecting Index and Tag Bits

Two related properties are available for customer to configure: *cc\_directory\_index\_bits* and *cc\_directory\_tag\_bits*. These properties specify the actual address bits that are used for index bits and for tag bits. This allows the use of higher order bits for indexing if desired.

However, the most common choice of index bits and tag bits will use the lowest order bits for indexing, and higher order bits for tagging. These properties will automatically be configured with the setup\_coherency command, and therefore they should not normally be set by the user.

#### Minimizing Storage

The directory entries are typically very small because they only hold the tag bits and some state bits. This means that the total RAM size is highly sensitive to each additional bit. For instance, a system with a 40-bit address size and 2MB of cache could have a directory with 8 ways and 4096 sets. This means 12 bits of index, 6 bits of offset, and the remaining 22 bits for tag. If 5 more state bits are needed, the total goes up to 27 bits per entry.

Because the number of bits is relatively small, removing any unneeded bits can have a significant impact on the area.

Removing a single bit can reduce the directory size by about 4% in this example.

There are two sets of bits that can usually be removed from the directory. The first is the address slice bits, when multiple CCCs are used in a sliced manner. If there are 4 CCCs, sliced using bits [7:6] of the address, then those two bits do not need to be included in either the index or tag of the directory.

The second common source of unused bits exists because of the address map. Typically, only a portion of the address space is coherent. Usually only the DRAM space, and sometimes a much smaller region used for RAM or ROM. This means that most of the address space is not coherent, and so the directory will never receive requests with these addresses. Since DRAM is typically placed near the bottom of the address map (near 0x0), there are usually higher order bits that are unused. For instance, in a 40-bit address, it may be that coherent address space always has Addr[39:36] equal to 4’b0000. This still leave 64GB of address space that can be used for coherent addresses. Removing these additional 4 bits can further reduce the directory size.

Since Gemini allows for a programmable address map, this reduction cannot happen automatically. Instead, the configuration must explicitly indicate that not all of the address bit will be used by the directory.

### Directory RAM Specification

The directory storage is intended to be implemented using single-ported RAM arrays. RAM is needed because of the quantity of storage needed which can be significant. The RAM is single-ported in order to make it as generic as possible and easier to implement.

The directory RAM is organized as logical arrays. The actual RAM design could further divide these arrays, but logically they will be treated as two arrays.



Figure 9: Directory organized as two tag arrays

The two RAMs are accessed in parallel during a lookup but can be written individually. The index of the access can be different for the two RAMs in the same cycle, which requires that they be separate RAM instances.

While multiple ways are contained in the RAM, an entire row is accessed at a time. So, the RAM design is not aware of the number of associative ways. It only sees an array of data blocks. The width of the data block is determined by NocStudio based on the required number of address bits, associativity, etc.

Since the RAM arrays are implemented by the customer, certain characteristics of the RAM must be provided to NocStudio to configure the hardware to access it correctly. The logic must know the latency of a read request, as well as the bandwidth of the RAM accesses. These are two different properties that can be set for the directory in the CCC host property list.

cc\_directory\_latency

cc\_directory\_bandwidth\_denominator

The first value specifies the latency of the RAM lookup. The RAM design is expected to have a flopped input and a flopped output. The latency value describes the number of cycles it takes to perform the RAM lookup. A latency of 1 means that the RAM will take a full cycle to do its lookup, but that data will actually arrive after two flop points. The following diagram shows a RAM latency of 1.



Figure 10: Example RAM latency == 1

As noted, when a RAM latency is set to 1, it means that if the request is sent on cycle N, the data is available during cycle N+2 since both inputs and outputs are flopped by the RAM.

The second property, “cc\_directory\_bandwidth\_denominator”, is used to indicate the bandwidth of the RAM. Depending on the frequency of the design and the size of the directory, it may be necessary to build the directory RAM with less bandwidth, such as only being able to perform an access once every other cycle.

This property is a bandwidth denominator. The bandwidth can be stated as 1 access/N cycles, where N is the value of this property. A value of 1 will indicate that a new access can happen each cycle. A value of 2 means one access every 2 cycles, etc. The maximum value is 4.

The following diagram shows a pipeline description of a directory RAM with a latency of 2 and a bandwidth denominator of 2.



Figure 11: Example of 2 cycle latency, 1/2 cycle bandwidth

It is possible for the latency and bandwidth numbers to differ. A RAM might have a long latency due to travel time to the sub-arrays but may have some ability to be pipelined.



Figure 12: Example directory RAM with 3 cycle latency, 1/2 cycle bandwidth

As seen in the diagram above, the RAM may have a latency of 3 cycles, but a bandwidth of 1access every 2 cycles. With these two properties, the logic can adapt to most RAM design requirements.

While the directory has two logical arrays, the arrays must be implemented identically. That means the latency and bandwidth properties are common to both arrays.

### ECC Support in Directory

The Directory supports ECC protection for the RAM. The ECC generation and checking happens in the CCC control logic, and the RAM itself only changes by having additional storage bits per entry. ECC can be added during configuration time with the following CCC host property:

cc\_directory\_ecc\_enabled

NocStudio will calculate the minimal number of additional bits needed to support ECC and add them to the RAM data width automatically. The approximate size of the ECC bits is log2(entry width)+1. So, 150 bits of data would require 9 bits of ECC overhead.

The ECC algorithm allows for single bit error correction, and double bit error detection.

#### ECC Algorithm

If the directory is configured to have ECC (Error-Correcting Code), the IP will implement a customized ECC algorithm. Additional bits will be added to the directory RAM array width to hold ECC information. The directory control logic will handle generating ECC values and checking the directory read results to confirm that there is no error.

The ECC algorithm uses a hamming code with an additional parity bit, sometimes referred to as SECDEC (single error correction, double error detection). The algorithm adds the ECC checkbits to the protected data block, so all bits are protected with single-bit correction and double-bit detection.

The hardware supports a register mechanism to directly access the directory RAM, including the ECC checkbits. It supports multiple variants, including a method to take an existing directory entry and flip one or more bits before writing it back into the array. This can be used to test ECC logic within the system.

### Parity Support in Directory

Instead of ECC protection of the Directory RAM, Gemini can configure the system to use Parity protection. While ECC can both detect single or double bit errors and correct single-bit errors, the Parity protection can only detect single-bit errors, and has no correction capabilities. It uses only a single additional bit, instead of the log2 bits needed for ECC, so this can provide some protection at a smaller cost.

The following CCC host property can be used to add parity to the directory.

cc\_directory\_parity\_enabled

### BIST and DFT Support for RAMs

Since the directory RAM is implemented by the customer, the BIST and DFT hooks for the RAM must also be provided by the customer.

A functional RTL model of the RAM is instantiated in Gemini RTL hierarchy. For the primary RAM functionality, the module within the CCC can be simply replaced with the more accurate customer RAM RTL module. However, to add additional hooks for BIST or DFT, additional signals must be made available to that module. To enable this, additional properties are created within NocStudio to route additional signals through the RTL hierarchy to the RAM instance. These properties are cc\_mem0\_in\_width, cc\_mem0\_out\_width, cc\_mem1\_in\_width, and cc\_mem1\_out\_width.

A generic bit vector can be passed to and from each RAM, with the width of this vector specified by these properties. If DFT/BIST needs 15 signals coming from the top level to the RAM, and 1 output, the values can be set to 15 for the in\_width, and 1 for the out\_width properties.

### Capacity Implications of CCC Slicing

When CCC slicing is used to increase bandwidth, it is usually not necessary for each directory instance to have the full capacity of the cache. Generally, the cache capacity can be divided by the number of slices. So, a 2MB cache capacity could be implemented with 4 CCC slices, each with a directory capacity of .5MB.

There are two reasons for this. The first is when address slice bits are the same as the cache index bits, then it is guaranteed that only ¼ of the lines in the cache can reside in a single CCC slice. So, if bits [7:6] are used for slicing and for indexing, then ¼ of the lines in the cache will have bits [7:6]==2’b01, so the CCC that is mapped by [7:6]==2’b01 will only need ¼ of the capacity.

Even if the cache uses different address bits, a random distribution of addresses will typically only need ¼ of the cache capacity in each of the 4 CCC slices. If each slice supported the full capacity, they would typically be only ¼ utilized.

So, for most designs, multiple CCCs can collectively match the capacity of the caches. This means the cost of slicing the CCC is only the overhead control logic in the CCC, and the directory RAM which will tend to dominate the area will not need to increase.

## Snoop Filter Support

The normal Gemini model utilizes a directory and expects ACE masters to provide snoop-filter support, as specified in section C10: Optional External Snoop Filtering in the AMBA AXI and ACE Protocol Specification. This means that when an ACE master replaces a cache line, it is expected to send an Evict request for any clean lines that it is dropping. This allows the directory to know which addresses it no longer need to track.

Gemini can also be configured to support ACE masters without the snoop-filter support (snoop-only agents), including a combination of both types.

Having snoop-only agents can impact both the latency and bandwidth of the coherent system. Latency is increased because coherent requests will always need to send snoops to the snoop-only agents and the round-trip plus processing delay will be added to the latency. Bandwidth is also reduced as snoops are sent one at a time from each CCC. Multiple snoop-only agents will reduce bandwidth by even more.

There may still be useful cases for snoop-only traffic. One example is when a master with a large cache, like a GPU, wants to be coherent but doesn’t want to increase the size of the directory. Another example is an external port where the size of the cache may vary.

To allow these cases to exist without significant performance loss, it is possible to distinguish between Inner Shareable and Outer Shareable domains. When snoop-only agents are created, Gemini will be automatically configured to create three Shareability domains: Inner-Shareable Snoop-Only; Inner-Shareable Directory Agents; Outer Shareable All Agents.

With these regions, directory supporting agents making Inner Shareable requests can benefit from the lower latency of the directory by not snooping the Snoop-only agents. When data may be shared with the snoop-only agents, the Outer Shareable domain can be used.

To specify a master as snoop-only, the master bridge property *cc\_snoop\_filter\_support* can be set to ‘no’.

## ACE Master Cache Capacity

ACE masters can have their cache capacity specified using the bridge property *cc\_cache\_capacity*. This indicates, in MB, how much cache is supported by that ACE master. This value is only used when the ACE master has snoop filter support. In that case, this value indicates to NocStudio how much cache each agent has that needs to be tracked by the directory. This can be used to automatically size the directory for the system.

## Speculative Fetch Control

Gemini provides an optional performance feature. When a coherent read is sent to the coherency controller, it has two choices.

The first choice is for the read to perform the directory lookup, send snoops if required, and wait for the snoop responses. If data was returned by any of the snoops (or by a WriteBack/WriteClean), it can be forwarded to the requesting agent without the need for making a request to LLC or Memory.

The alternative is to perform a speculative fetch. The speculative fetch issues a read to the next stage of the memory hierarchy (LLC or memory) as soon as possible, even before the directory has completed its lookup. This can significantly reduce the latency of the request since it doesn’t have to wait for the directory access, snoops, or snoop responses. However, it may lead to memory reads that turned out to be unnecessary, wasting some memory bandwidth.

Since the speculative fetch reduces latency but potentially increases bandwidth costs, it should be selectively used. Typically, it should be enabled for latency sensitive requests, and disabled for requests that are less latency sensitive in order to save the bandwidth costs.

The bridge property cc\_axi4m\_speculative\_fetch can be set to yes or no. By default, ACE masters will be set to yes, as they are often latency sensitive. Other masters are set to no, as IO coherent agents are often more latency insensitive.

The speculative fetch value here only sets the default behavior for the interconnect. These values can be overwritten using programmable registers in the CCC(s).

## Fast Tap Support

Under certain circumstances, Gemini can support a reduced latency mechanism called a fast tap. The fast tap is more direct connection between an ACE master and the coherency controller. By having a direct connection, it is able to skip some of the network overhead of the system. This overhead includes packetization, arbitration and travel through the network, as well as de-packetization. And that cost can exist for both the AR and R packets.



Figure 13: Fast Tap

The Fast Tap creates a dedicated path to the CCC and back, skipping packetization and de-packetization, as well as any arbitration steps. As seen in the diagram above, the fast tap connection creates additional dedicates paths to the CCC. Other traffic can be sent through the bridge to the normal paths through the NoC. The Fast Tap is only used for coherent reads and Cache Maintenance operations that target the CCC.

Fast Tap does require dedicated wires as well as some additional storage, so it does not come for free. However, for many systems, the latency reduction may be worth the additional costs.

Fast Tap has a number of restrictions.

1. Only ACE masters can use the Fast Tap.
2. Fast Tap is limited to at most 2 ACE masters.
3. Fast Tap is limited to interfaces with a 16 or 32-byte R channel.
4. Fast Tap can only exist in a system with a single CCC.

The property used to specify that an ACE master should utilize the fast tap is *cc\_fast\_tap\_enabled*, which defaults to “no”.

## CD Channel configuration

### CD Enable/Disable

ACE protocol specifies that the CD channel (snoop data response) is an optional bus. An ACE master can be built without this channel, requiring any data to be sent back using a WriteBack command on the AW and W channels.

The property *cc\_cd\_channel\_enabled* is used to specify whether the ACE master has a CD channel or not. By default, it is specified as “yes” since CD channel is usually included. Marking this as no will remove the CD channel signals from the bus.

### CD Channel Width

If the CD channel present on an ACE master port, the data width of that channel can be independently controlled from the R and W channel widths, allowing a different snoop data response bus width. This is an interface property for the “crcd” channel, called data\_width. The following is an example of how to set this property:

ifce\_prop cpu0/mprt.crcd.out data\_width 128

## Transaction handling

### Request Splitting

CFG Gemini will split all reads and writes that target CCC, IOCB, or LLC if the requests cross a natural 64-byte boundary interface. These components process requests on a 64B cache line granularity, requiring any larger requests to be split into smaller requests that can be processed.

For other requests in the system, request splitting can be controlled by the axi4m\_request\_split\_size bridge property of the master.

### ACE Agent 64-byte Limitation

Due to an issue in the ACE specification, full ACE agents are restricted to coherent accesses within a 64-byte boundary. Coherent requests that cross a 64-byte boundary will be rejected by the NoC. This only applies to full ACE agents. AXI or ACE-lite agents have no such restriction and are instead limited by the 4KB AXI granularity requirement.

### Write-Evict Handling

In Rev E of the ACE protocol specification, the Write-Evict requests is added to the protocol. This is to be used in conjunction with a last-level cache. A configuration without a last-level cache should avoid issuing Write-Evict requests, as they can cause unnecessary requests to the CCC or to memory. In ARM processors, this can usually be prevented by disabling UniqueClean Evictions. In cortex-A15, for instance, this can be found in the L2ACTLR register.

### Bufferable Requests

CFG Gemini does not provide early acknowledgments for write requests. If a request is marked as bufferable, Gemini will not treat it any differently from non-bufferable requests.

### Barrier Handling

CFG Gemini supports barriers in the ACE specification in multiple ways. For most barriers, the bridge will simply wait for prior requests to complete, which occurs when the BRESP or RRESP packet returns. For coherent memory barriers (Inner Shareable or Outer Shareable), barriers sent from ACE-lite or ACE-lite+DVM agents will be forwarded to the CFG IOCB, to allow greater performance. ACE master bridges locally complete barriers sent from ACE masters.

Since barriers are not broadcasted to slave devices, any early acknowledgments from the slaves will allow the barriers to complete.

### Read Data Interleaving

AXI protocol allows read requests to have their data responses interleaved by the slave and through the network. CFG Gemini does not support this. If slaves do interleave data, a data de-interleaver can be added to the slave’s bridge in NocStudio.

### Command Mapping

In the ACE protocol, there are many different command variants. This section describes how these different commands are processed by Gemini. The table below shows which destination the command is sent to be processed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ARSNOOP | ARDOMAIN | ARBAR | Transaction Type | Target |
| 4’b0000 | 00, 11 | x0 | ReadNoSnoop | Slave |
| 4’b0000 | 01, 10 | x0 | ReadOnce | IOCB or CCC\* |
| 4’b0001 | 01, 10 | x0 | ReadShared | CCC |
| 4’b0010 | 01, 10 | x0 | ReadClean | CCC |
| 4’b0011 | 01, 10 | x0 | ReadNotSharedDirty | CCC |
| 4’b0111 | 01, 10 | x0 | ReadUnique | CCC |
| 4’b1011 | 01, 10 | x0 | CleanUnique | CCC |
| 4’b1100 | 01, 10 | x0 | MakeUnique | CCC |
| 4’b1000 | 00,01,10 | x0 | CleanShared | CCC |
| 4’b1001 | 00,01,10 | x0 | CleanInvalid | CCC |
| 4’b1101 | 00,01,10 | x0 | MakeInvalid | CCC |
| 4’b0000 | 01,10 | 01 | Coherent Memory Bar | IOCB or local bridge\*\* |
| 4’b0000 | 01,10 | 11 | Coherent Sync Bar | local bridge |
| 4’b0000 | 00,11 | x1 | Non-coherent Bar | local bridge |
| 4’b1110 | 01, 10 | x0 | DVM Complete | DVM |
| 4’b1111 | 01,10 | x0 | DVM Message | DVM |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AWSNOOP | AWDOMAIN | AWBAR | Transaction Type | Target |
| 3’b000 | 00, 11 | x0 | WriteNoSnoop | Slave |
| 3’b000 | 01, 10 | x0 | WriteUnique | IOCB or CCC\* |
| 3’b001 | 01, 10 | x0 | WriteLineUnique | IOCB or CCC\* |
| 3’b010 | 01,10 | x0 | WriteClean | CCC |
| 3’b010 | 00 | x0 | Non-Share WriteClean | Slave |
| 3’b011 | 01,10 | x0 | WriteBack | CCC |
| 3’b011 | 00 | x0 | Non-Share WriteBack | Slave |
| 3’b100 | 01,10 | x0 | Evict | CCC |
| 3’b101 | 01,10 | x0 | WriteEvict | CCC |
| 3’b101 | 00 | x0 | Non-Share WriteEvict | Slave |
| 3’b000 | 01,10 | 01 | Coherent Memory Bar | IOCB or local bridge\*\* |
| 3’b000 | 01,10 | 11 | Coherent Sync Bar | local bridge |
| 3’b000 | 00,11 | x1 | Non-coherent Bar | local bridge |

\* ACE masters send these requests to CCC. ACE-lite, ACE-lite+DVM, ACE-lite converted bridges send the request to IOCB.

\*\*Barriers are sent to IOCB from ACE-lite or ACE-lite+DVM agents. For ACE agents, they are locally completed.

In the tables above, there are 5 possible destinations for a request to be sent to. Non-coherent requests can target the destination slave, without going through other coherency components. Some other requests target the IOCB, which can make request to the CCC if necessary. Some request will go straight to CCC for processing. DVM requests will go to the DVM module for processing. And finally, some of the barrier instructions will be satisfied locally at the bridge.

Note that a configuration may have multiple slaves, multiple CCCs, etc. Additional address map information is used to identify which specific agent a request will be sent to. Additionally, if an address is not mapped to a device, a decode error will be generated in the local bridge.

## Deadlock Avoidance Unit

With today’s systems supporting different standard protocols, it is vital to make sure the interoperability and dependencies are taken care of. PCIe introduce some specific requirements that have to be designed for to avoid unforeseen dependencies with other protocols. The deadlock-avoidance unit is designed for this specific purpose and provides decoupling of protocols necessary to allow interfacing between a coherent subsystem with cores using the AMBA-ACE protocol and a PCIe root complex.

### ACE + PCI Deadlock

ACE protocol has deadlock incompatibilities with PCI-based protocols (PCI, PCIe, HT, etc.). Ultimately this is because the ACE interface uses the same channel (AW) to issue Copy-backs and non-cacheable requests. This allows non-cacheable requests to block the WriteBacks from happening. The following diagrams shows the ACE+PCI deadlock.



**Figure 14: ACE/PCI Deadlock**

As seen in the diagram, the deadlock loop happens when the PCI is bi-directional with upstream writes being IO coherent. When this happens, snoops may be sent to the ACE master, which may require a writeback to drain in order to satisfy the snoop. If the ACE master has sent traffic to the NPC downstream PCI path (usually for configuration accesses), this creates a dependency loop that ultimately causes deadlock. The red path shows the loop.

From a configuration perspective, deadlock can occur when PCI upstream writes are IO coherent, PCI network is bi-directional (root complex), and ACE master can issue writes to PCI.

### Architecture

The key issue in the ACE/PCI deadlock is the AW channel of the ACE agent. All other dependencies in the system are requirements of the traffic flows. This is an artificial dependency that exist because of the interface specification. Consequently, the work-around for this deadlock is to alleviate the dependency on that channel.

To avoid that mechanism, a hardware component can be added after the AW channel of the ACE master that has enough storage to guarantee that all PCI writes from that ACE master are allowed to drain and get out of the way of CopyBacks from the ACE master. Logically, this storage can reside anywhere within the NoC. For convenience of keeping storage in a single location for implementation efficiency, the hardware component would connect at the interface with the AXI<->PCI bridge.



**Figure 15: Deadlock Avoidance Unit in the System**

In the diagram above, the Deadlock Avoidance Unit is present at the interface to AXI/PCI bridge. The unit has two ports connecting to the NoC.

* One port that is dedicated to ACE master traffic. This allow a dedicate path for ACE master traffic to PCI to drain even if the PCI bridge itself is back-pressured.
* Second port is used for all other traffic, such as writes from non-ACE masters.

#### Motivation for second port

The two ports are needed to minimize total storage in the unit. The only required storage is enough to drain ACE master writes even when the PCI is back-pressured. If non-ACE master traffic came into this unit on the same port as ACE master, that port could get flow-controlled and prevent the ACE writes from being able to drain. That can be avoided by adding additional storage to guarantee all writes aimed at PCI can drain, but that is expensive. Having separate flow control is better. This limits the total storage required by this unit to only the ACE master write traffic.

The second port is only needed when there are non-ACE masters that can send writes to the PCI. If all traffic comes from ACE masters, no traffic would target the second port.

#### Buffer optimization

The buffering requirements to guarantee traffic can drain from ACE masters is one buffer for every possible write outstanding by the ACE masters. This can get expensive. One optimization is if the master can recognize the PCI space, and limit the number of writes it is able to issue. This would require that two *max\_write\_outstanding* properties be set.

* One would be for the total number of writes.
* The other would be any request limit specifically to the PCI.

### DAU RAM structure

The write queue within the DAU can be implemented using a single-ported RAM structure.



**Figure 16: RAM based write queue**

* The RAM can only be accessed once per cycle.
* The access to the RAM uses strict priority, where reading data out of the RAM is higher priority than writes to the RAM. This will backpressure the NoC, but only when draining is possible, so there’s no deadlock issues.

#### RAM Bypass

The incoming data can bypass the RAM and go directly to the FIFO if there is an available spot in the FIFO, if the RAM-based queue is empty, and if a read to the RAM isn’t already in progress (since that entry will be higher priority). If data is written to the RAM every time, the bandwidth of the interface will be ½, since half the RAM bandwidth will be used for writing and the other half for reading. By bypassing in normal conditions, the Queue can still allow full throughput.

#### Flop Array

Instead of a RAM array, a flop array can be used. This can make sense of the storage needed is too small for a RAM, or to make the design synthesizable.

## Coherency Connect/Disconnect

ACE masters and ACE-lite+DVM masters can receive snoops from CCC or DVM. To shut down one of these ports, the coherency IP must be alerted that the agent wants to be disconnected from there coherency protocol so they can stop sending snoop requests to that agent. To provide this functionality, these agents have a connect/disconnect mechanism for coherency.

These agents can be configured in three ways in NocStudio. They can be configured to have a control pin that selects whether the agent is connected or disconnected from the coherency, or they can be controlled through registers.

If pins are configured, the NoC will create two new signals to the port called SYSCOREQ and SYSCOACK. The SYSCOREQ is driven by the master to connect to the coherency protocol. The SYSCOACK provides a status response. These signals are part of a 4-state handshake. At reset, the agent starts disconnected, and SYSCOREQ and SYSCOACK are deasserted. While disconnected, the agent cannot issue coherent requests or DVM requests. When it wants to connect, it will assert the SYSCOREQ signal. It must then wait for the SYSCOACK signal to indicate that the connection has been made. The SYSCOREQ should not be deasserted until SYSCOACK is raised.

Once connected, the agent can receive snoops and may issue coherent requests including DVM requests.

If the agent decides to disconnect, it must go through a disconnect process. The first thing is must do is stop making new coherent requests. The ACE master must then flush its caches of any modified lines. This is because once it is disconnected, no other agents will be able to snoop it and any data it has will be lost. Once the cache is flushed, and all CopyBacks have received their responses, the agent can signal that it wants to disconnect by deasserting the SYSCOREQ. Even after SYSCOREQ is deasserted, the agent must still accept snoops and respond to them appropriately. This is because there may already be snoops outstanding in the network, and they must be successfully completed.

Eventually, the SYSCOACK will indicate that the coherent agent has successfully disconnected and that no new snoops are outstanding. At this point, the agent can power down or reset without breaking the rest of the coherent system.

The coherent agent can instead be set up to have register control of the coherency connect/disconnect mechanism. Instead of having pins to control this, the master bridge will have internal registers that generate or sync the SYSCOREQ and SYSCOACK signals. To disconnect a connected agent, a register writes to the SYSCOREQ register of that bridge should be made. When the SYSCOACK register match the SYSCOREQ value, the change has taken place. Like the pins, SYSCOREQ shouldn’t change until after the SYSCOACK is matching it.

If this register control mode is select, the user can configure an agent to come out of reset either connected or disconnected. If connected, the SYSCOREQ signal will start of asserted, and the connect sequence will start immediately. If disconnected, the agent will start of disconnect and only become connected when the SYSCOREQ register is written.

The coherency connect/disconnect mechanism can be used for powering down an agent. It can also be used for error handling, to allow an agent do be reset without disrupting the coherent system.

The ECC detection and correction can also be disabled via register access.

## Invalidation Engine

The directory supports an invalidation engine that can invalidate the full content of the directory. The engine runs through every set of the array and overwrites all bits in that set to zero, including ECC bits if present. No stale content will be left in the directory after the invalidation engine has completed.

The invalidation engine is triggered on the deassertion of reset. While executing the invalidation, any coherent requests that need to access the directory will be stalled until the sequence has completed.

The invalidation engine is built to issue writes to both directory arrays in parallel, to speed up the time to completion. A directory set contains multiple entries, all of which are invalidated in a single write.

The invalidation engine can also be triggered by a register access. This must be carefully controlled. The invalidation will cause a loss of coherent state information, so any agents holding a copy of a line will no longer be tracked. To avoid this, all caches should be invalidated if the directory is invalidated.

While the invalidation engine will overwrite the directory content, any outstanding requests may modify the directory state after the invalidation engine has completed. To avoid this effect, all requests to the CCC must be completed and the CRT entries should be checked to confirm there is no outstanding requests.

## QoS overrides for derivative coherent transactions

Coherency requires extra points for checks and balances in the life of a transaction, to ensure that the coherent intent of the transaction is respected and maintained. As a result, derivative transactions are injected to send messages and or transfer data. The QoS requirements for these derivative transactions are not specified by the user and is usually not explicitly available. Depending on the system use case incorrect QoS values could make or break the performance of the coherent system. NocStudio adds support to allow the user to control the QoS for all the derivative coherent transaction based on the system needs. This allows a system to define varied QoS for below transactions, giving the user an intelligent way to control the sharing of available bandwidth.

* Flow through transactions - Read requests, Snoop responses, and Write requests initiated by the CCC
* Derivative transactions - Direct evicts that trigger write requests