# Cache Hierarchy Configurability

Gemini supports a flexible cache hierarchy, allowing the architect significant control of the design. A cache hierarchy is the organization of the caches, coherency components, and memories. It is defined by the connectivity of these components, as well as which addresses are handled by each component.

A simple example of a cache hierarchy is shown below:



Figure 17: Coherent and Non-Coherent Devices

In the diagram above, a master is connected to two slave devices. One is memory slave that is intended to support coherency. The other is an I/O slave that accepts only non-coherent requests. As seen in the diagram, the coherent memory is connected to the master through a coherency controller and a last level cache. For the I/O slave, non-coherent requests can be sent directly from the master to the slave.

## Asymmetric Hierarchies

The cache hierarchy can be asymmetric within the same system.



Figure 18: Asymmetric Cache Hierarchy

In the diagram above, both slaves are enabled for coherency, but only one of them utilizes the cache. Since one of the slaves is an on-chip RAM, it provides no benefit for storing data in the LLC since the on-chip RAM can be accessed just as quickly. It also allows the LLC to improve performance more by only caching lines with a long memory latency.

## Different Ranges for the same Slave

This asymmetry can also apply to the same slave, but allowing different memory ranges to be treated differently.



Figure 19: Different ranges of to the same memory controller

In the diagram above, different ranges of the same memory controller can be handled differently, with one range going through a cache, while the other range goes directly from the CCC to the memory controller.

## Adding Slices for CCC or LLC

For systems with larger bandwidth requirements, it may be necessary to utilize multiple CCCs or LLCs to increase bandwidth. One common method for supporting this is to take an address range and slice it into equal parts, with each CCC or LLC responsible for one of the parts. If requests are well distributed to the different slices, bandwidth will increase proportional to the number of components. Having 4 instances of a cache can get 4x the bandwidth of a single instance.

The slicing function uses specified address bits to assign responsibility to each component. Slicing can be done using a power-of-2 number of slices. This allows a simple decode of address bits.



Figure 20: Uniform sliced address space

In the figure above, the memory controller address space is split into 4 regions, dividing by address bits 7 and 6. Each address region is handled by a separate CCC and a separate LLC. Each CCC slice handles ¼ of the memory addresses, as does each LLC slice. Since both the CCCs and the LLCs are sliced with the same bits, connectivity is limited. Each CCC slice only has a connection to a single LLC slice.

## Different Slicing between CCC and LLC

In the example above, slicing was done in a uniform way between the CCCs and LLCs. This may be a useful design method, but Gemini supports a larger variety of cache hierarchies. The number of CCCs and the number of LLC can vary.



Figure 21: Different Slicing between CCC and LLC

The figure above shows 4 CCCs and 4LLCs, but with a different connectivity. CCC0 and CCC1 can each talk to LLC0 and LLC1, while CCC2 and CCC3 talk to LLC2 and LLC3. This configuration can happen when slice bits are chosen in a different manner.

## Different Number of CCCs and LLCs

Since slicing can be different between CCCs and LLC, Gemini can also support a different number of CCCs and LLCs in the same cache hierarchy. This can be useful if the bandwidth requirements are different or if physical location requires agents to be physically distributed.



Figure 22: Different number of CCCs and LLCs

In the figure above, there are only 2 CCCs while there are 4 LLCs in this cache hierarchy. Additionally, each CCC only talks to two LLCs.

## Combining These Design Choices

Complex cache hierarchies can be built with these various controls, as seen below.



## Non-Coherent Access Point

Gemini’s cache hierarchy support allows for the LLC or Cache to be added as either a Coherent Cache or as a Memory Cache.

A Coherent Cache is a cache that is only accessible by coherent requests. The CCC sends requests to the LLCs, but non-coherent traffic skips the CCC and the LLC and talks directly to memory.



Figure 23: Example Coherency Cache hierarchy

In the diagram above, coherent traffic is sent to IOCB and CCC, which can send coherent requests to the LLC. Non-coherent requests, as embodied by the AXI agent’s traffic, skip past the coherent system and the LLC, and get sent directly to the memory port.

An alternative cache hierarchy design uses a Memory Cache. A memory cache is a cache that is accessible by all traffic, including coherent and non-coherent. The following diagram shows an example memory cache system.



Figure 24: Example Memory Cache hierarchy

There are several tradeoffs between a coherent cache and a memory cache, and choosing the right option for a system can create a significant differentiation.

The decision of a memory cache vs. a coherent cache is decided during construction of the Gemini system. The *add\_llc\_group* or *add\_cache\_group* commands have a memory cache attribute that can be set. If set, the cache will act as a memory cache, and non-coherent traffic to that range will be sent to the LLC. If the attribute is not set, the default will be coherent cache behavior, where only the CCC talks to the LLC.