WHITE PAPER

Introduction to AMBA® 4 ACE™ and big.LITTLE™ Processing Technology

Ashley Stevens
Senior FAE, Fabric and Systems

June 6th 2011
Updated July 29th 2013

Why AMBA 4 ACE?

The continual requirement for more processing performance whilst maintaining or improving energy consumption to increase battery life and/or reduce energy use, drives the requirement for power-efficient processing. It is well known that multi-processing is a power-efficient way to add additional performance, as opposed to pushing a single processor to ever higher performance by increased use of low-Vt transistors, high-performance but leaky process technology and higher over-drive power-supply voltages. Provided software can make use of parallel hardware it's more power-efficient both in terms of power and area to add additional parallel processor units. Today most ARM Cortex™-A9 implementations are multi-core, either dual or quad core.

An innovative ARM technology known as big.LITTLE™ pairs a high-performance multi-issue out-of-order processor cluster with a high-efficiency in-order processor cluster to deliver optimal performance and energy consumption. The two processors clusters are architecturally compatible, meaning they appear equivalent to software other than in terms of performance. Tasks requiring more modest CPU performance such as email, social media, accelerated video playback and background processes can run on one of the smaller, high-efficiency processor cores. Tasks requiring high performance and which benefit from being run as fast as possible, such as rendering a webpage, run on a core on the high-performance processor cluster, which shuts down when the high performance requirements cease. Power gating the processor removes leakage as well as dynamic power consumption. Benchmarking of smartphone workloads has proven that high-end smartphones typically spend very little time at the highest DVFS (Dynamic Voltage and Frequency Scaling) performance points, spending most time at lower frequencies. Thus most of the time the smaller processor is sufficient.

The first ARM big.LITTLE implementation consists of the powerful Cortex-A15 MPCore™ processor and the highly efficient Cortex-A7 MPCore processor. The Cortex-A7 is around one quarter to one fifth of the power and area of the Cortex-A15, which is from 2 to 3 times more powerful than the Cortex-A7. It can be seen that those tasks that can execute on the Cortex-A7 can do so more efficiently than on the Cortex-A15, but the Cortex-A15 can cope with a much wider range of tasks beyond the single-thread performance of the Cortex-A7. Fast and power-efficient switching between the big and little cores is enabled by cache coherency between the processor clusters. Cache coherency enables cached data to be transferred between the clusters without being written back to external memory, saving time and energy.

Further, it is common to include within an SoC specialized accelerators for heavy-lifting tasks that would be inefficient on general-purpose CPUs, such video encode and decode. Programmable GPUs are prevalent amongst consumer device with a display. To make a complete SoC, each of these multiple CPU implementations, GPUs and hardware accelerators must be able to communicate and share data with one another. With the proliferation of caches, the challenge is to provide consistent views of shared data and to minimize the external memory bandwidth requirement, which is increasingly the main system performance bottleneck. This data-sharing, communications and data movement combined with the presence of multiple caches amongst heterogeneous processing units drives the demand for a flexible cache coherency protocol that can work not just amongst like CPUs (as has been done many times in the past on various processor architectures), but can also work between dissimilar CPUs as used in big.LITTLE technology, GPUs and the hardware accelerators present in today's SoCs.

Evolution of AMBA Specifications

The history of AMBA goes back to 1995 with the launch ASB and APB as AMBA 1. 1999 brought AHB in AMBA 2 and in 2003 AXI was released as part of AMBA 3. In 2010 AMBA 4 brought some minor changes to AXI in the form of AXI4, as well as the new interfaces AXI4-Lite and AXI4-Stream. AXI4 added support for long bursts and dropped support for write interleaving. AXI4-Stream is a point-to-point protocol without an address, just data. AXI4-Lite is a simplified AXI4 for on-chip devices requiring a more powerful interface than APB. In 2011 AMBA 4 phase-2 introduced ACE (AXI Coherency Extensions) and ACE-Lite. This paper focuses on the ACE and ACE-Lite interfaces, which introduce system-level coherency, cache maintenance, distributed virtual memory (DVM) and barrier transaction support. Further details on AXI4, AXI4-Lite and AXI4-Stream can be found on the ARM website by downloading the AMBA Specifications.

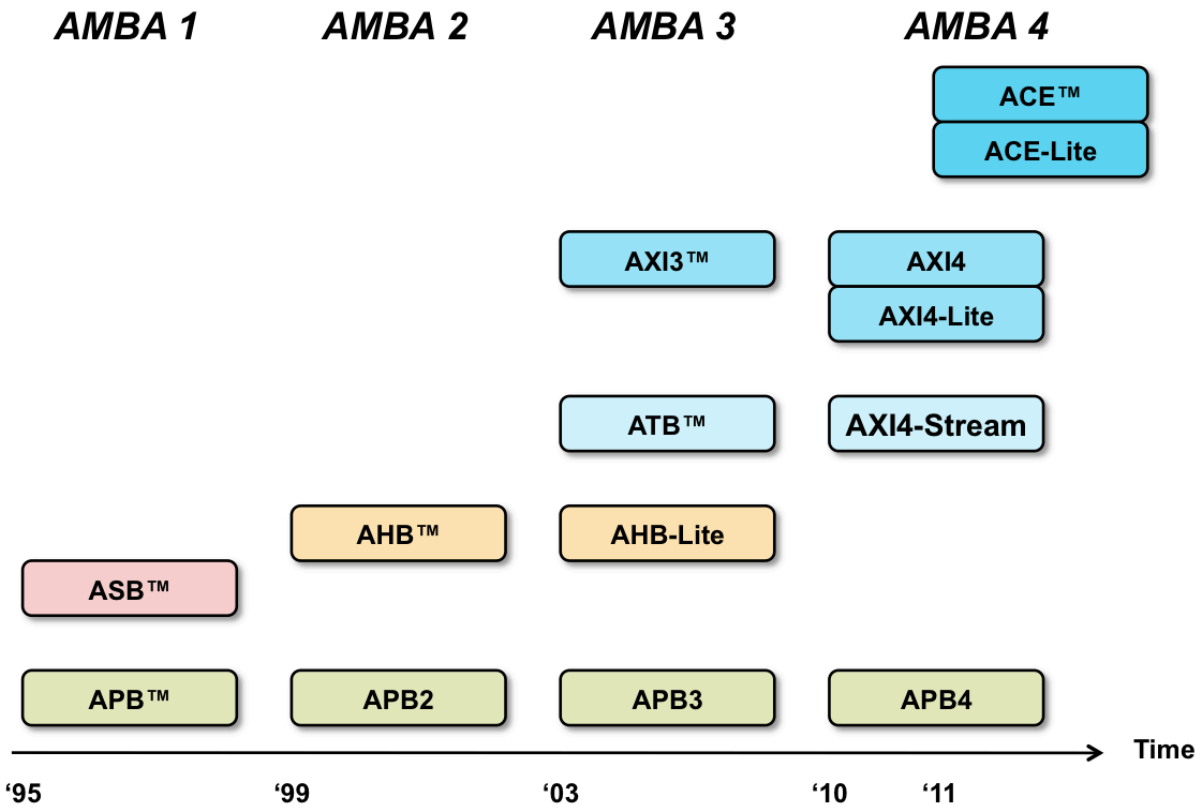


Figure 1 – Evolution of AMBA Standards

The Coherency Challenge

Cache coherency is an issue in any system that contains one or more caches and more than one entity sharing data in a single cached area. There are two potential problems with system that contains caches. Firstly, memory may be updated (by another master) after a cached master has taken a copy. At this point, the data within the cache is out-of-date and no longer contains the most up-to-date data. This issue occurs with any system that contains any kind of cache. Secondly, systems that contain write-back caches must deal with the case where the master writes to the local cached copy at which point the

memory no longer contains the most up-to-date data. A second master reading memory will see out-of-date (stale) data.

Software-Based Coherency Approach

Cache coherency may be addressed with software-based techniques. In the case where the cache contains stale data, the cached copy may be invalidated and re-read from memory when needed again. When memory contains stale data due to a write-back cache containing dirty data, the cache may be cleaned forcing write-back to memory. Any other cached copies that may exist in other caches must be invalidated.

Software-based approaches put the coherency problem onto the programmer. With today's SoC-based product developments, software costs are approaching or even exceeding the costs of hardware development, and regardless of cost software is always on the time-to-market critical path. So anything that can be done to reduce software complexity and timescales is welcome. Invalidating and cleaning caches costs processor time and hence performance and adds a great deal of complexity. If this is done more than necessary then performance and energy-consumption suffers. If it's not done when required then the result is often obscure, hard-to-replicate and track-down bugs that eat into the critical time-to-market window. When SoC systems contained only one cached processor with just small level-1 (L1) caches, software cache maintenance was a reasonable solution. With today's SoCs with multiple CPUs containing L1 and L2 caches (and in some cases even L3 caches), plus other cached masters like DSPs and GPUs, software-based solutions are starting to become untenable. A better solution is required.

Hardware-Based Coherency Approaches

Snooping cache coherency protocols

Hardware-based coherency techniques can be divided into two main approaches. Cache coherency systems based on *Cache Snooping* rely on all masters 'listening in' to all shared-data transactions originating from other masters. Each master has an address input as well as an address output to monitor all other transactions that occur. When the master detects a read transaction for which it has the most up-to-date data it provides the data to the master requesting it, or in the case of a write it will invalidate its local copy. This scheme was originally used in board-level, bus-based systems where the address bus was bidirectional, but is nowadays frequently used for small-scale chip-level multi-processors consisting of 2 to 8 processors. In broadcast snoop systems the coherency traffic is proportional to:

$$N * (N-1) = N^2 - N \qquad \text{where N is the number of coherent masters}$$

since for each master the broadcast goes to all other masters except itself, so coherency traffic for 1 master is proportional to N-1, and since each master can do this then you multiple by N originators to get $N^2 - N$. Clearly purely broadcast-based snooping systems have limits to their scalability.

Directory-based cache coherency protocols

An alternative approach is the *Directory-based* coherency protocol. In a directory-based system there is a single 'directory' which contains a list of where every cached line within the system is held. A master initiating a transaction first consults the directory to find where the data is cached and then directs cache coherency traffic to only those masters containing cached copies. These systems scale better provided data is shared by only a small subset of the total cached masters since cache coherency traffic

is only sent where strictly needed, as opposed to snooping systems where all cache coherency traffic is broadcast to all other masters. In the best case if all shared data is shared only by two masters and we count the directory lookup and the snoop as separate transactions then traffic scales at order 2N. In the worst case where all traffic is shared by all masters, then a directory doesn't help and the traffic scales at order

$N^* ((N - 1) +1) = N^2$ where the '+1' is the directory lookup. In reality, data is probably rarely shared amongst more than 2 masters except in certain special-case scenarios.
The disadvantage of directory-based systems is the cost of the directory and the additional latency of accessing it prior to every shared transaction. If the directory is held in off-chip memory then latency will be poor. If it's held in on-chip memory then there's the cost of a large on-chip RAM.

In reality, these two schemes are really two ends of a continuum of approaches. Snoop based systems can be enhanced with *snoop filters* that can filter out unnecessary broadcast snoops by using partial directories. Snoop filters enable larger scaling of snoop-based systems. A directory-based system is akin to a snoop-based system with perfect, fully populated snoop filters.

AMBA 4 ACE is designed to be a flexible protocol that enables the creation of systems based on either snoop or directory-based approaches and may be used for 'hybrid' approaches such as those with snoop filters. ACE does not restrict the designer to one or other approach, nor does it restrict systems to any particular limit of coherent masters. But the system designer needs to select a suitable approach based on their system size and complexity.

ARM MPCore Intra-Cluster Coherency Technology

ARM introduced MPCore™ multi-core coherency technology in the ARM11 MPCore processor and subsequently in the ARM Cortex-A5 MPCore and Cortex-A9 MPCore processors. The Cortex-A15 MPCore processor also supports MPCore technology but further extends it with AMBA 4 ACE system-wide coherency capability.

MPCore technology enables cache-coherency within a cluster of 2 to 4 processors. MPCore processors implement a modified MESI cache coherency protocol, improving upon MESI by enabling direct cache-to-cache copy of clean data and direct cache-to-cache move of dirty data within the cluster, without write back to memory as would normally be required by a MESI-based processor. Performance is enhanced by the ARM 'Snoop Control Unit' (SCU), which maintains a copy of all L1 data cache tag RAMs acting as a local, low-latency directory, enabling it to direct transfers only to the L1 caches as-needed. This increases performance as unnecessary snoop traffic to L1 caches would increase effective processor L1 access latency by reducing processor access to the L1 cache.

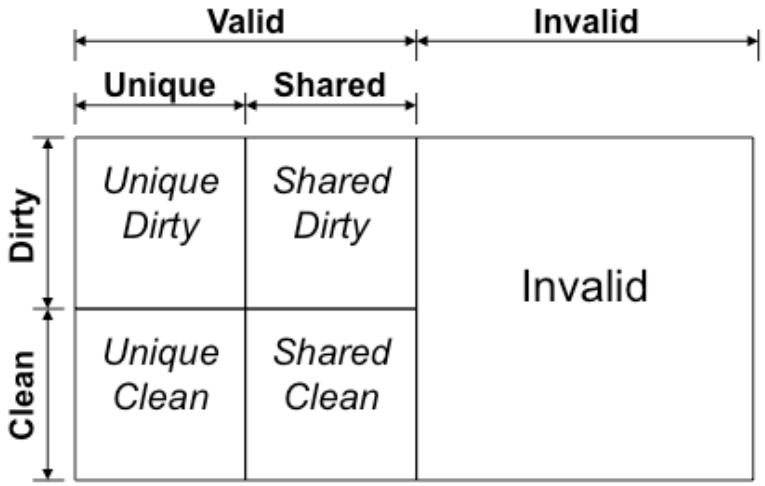
MPCore technology also supports an optional Accelerator Coherency Port, which enables un-cached accelerators access to the processor cache hierarchy, enabling 'one-way' coherency where the accelerator can read and write data within the CPU caches without a write-back to RAM. But ACP cannot support cached accelerators since the CPU has no way to snoop accelerator caches, and the accelerator caches may contain stale data if the CPU writes accelerator-cached data. Effectively the ACP acts like an additional master port into the SCU and the ACP interface consists of a regular AXI3 slave interface.

MPCore technology supports scaling to 4 processor cores. Now the Cortex-A15 processor adds ACE, which not only supports scaling to multiple CPU clusters enabling systems containing more than 4 cores, but also supports heterogeneous systems consisting of multiple CPUs **and** cached accelerators.

AMBA 4 ACE Cache Coherency States

The AMBA 4 ACE protocol is based on a 5-state cache model. Each cache line is either *Valid* or *Invalid*, meaning it either contains cached data or not. If it's *Valid* then it can be in one of 4 states defined by two properties. Either the line is *Unique* or *Shared*, meaning it's either non-shared data, or potentially shared data (the address space is marked as shared). And either the line is Clean or Dirty, *generally* meaning either memory contains the latest, most-up-to-date data and the cache line is merely a copy of memory, or if it's *Dirty* then the cache line is the latest most up-to-date data and it must be written back to memory at some stage. The one exception to the above description is when multiple caches share a line and it's dirty. In this case, all caches must contain the latest data value at all times, but **only one** may be in the *SharedDirty* state, the others being held in the *SharedClean* state. The *SharedDirty* state is thus used to indicate which cache has responsibility for writing the data back to memory, and *SharedClean* is more accurately described as meaning data is shared but there is no need to write it back to memory.

The ACE states can be mapped directly onto the MOESI cache coherency model states. However ACE is designed to support components that use a variety of internal cache state models, including MESI, MOESI, MEI and others. ACE does not prescribe the cache states a component can use. Some components may not support all ACE transactions. Indeed the ARM Cortex-A15 MPCore internally uses MESI states for the L1 data cache meaning the cache cannot be in the *SharedDirty* (Owned) state. To emphasize that ACE is not restricted to the MOESI cache state model, ACE does not use the familiar MOESI terminology.



ARM ACE	MOESI		ACE meaning
UniqueDirty	M	Modified	Not shared, dirty, must be written back
SharedDirty	O	Owned	Shared, dirty, must be written back to memory
UniqueClean	E	Exclusive	Not shared, clean
SharedClean	S	Shared	Shared, no need to write back, may be clean or dirty
Invalid	I	Invalid	Invalid

Figure 2 – ACE Cache Line States and their alternative MOESI naming

ACE Protocol Design Principles

Understanding the basic ACE protocol principles is essential to understanding ACE. Lines held in more than one cache must be held in the Shared state. Only one copy can be in the SharedDirty state, and that is the one that is responsible for updating memory. As stated previously, devices are not required to support all 5 states in the protocol internally. But beyond these fundamental rules there are many options as ACE is very flexible.

The system interconnect is responsible for coordinating the progress of all shared (coherent) transactions and can handle these in various manners. For example, the interconnect may present snoop addresses to all masters in parallel simultaneously, or it may present snoop addresses one at a time serially. The interconnect may choose to perform speculative reads to lower latency and improve performance, or it may choose to wait until snoop responses have been received and it is known that a memory read is required to reduce system power consumption by minimizing external memory reads. The interconnect may include a directory or snoop filter, or it may broadcast snoops to all masters. If it does include a directory or snoop filter it must ensure that it performs all snoops that are required of the protocol, however it's obviously acceptable to perform snoops that are not required, as this results merely in slightly worsened performance and power but is not functionally incorrect.

Unlike early cache coherency protocols and systems (such as board-based MEI snoop coherency where all snoop hits resulted in a write-back), which were concerned primarily with functional correctness, ACE has been designed to enable performance and power optimizations by avoiding wherever possible unnecessary external memory accesses. ACE facilitates direct master-to-master data transfer wherever possible. Since off-chip data accesses are an order of magnitude (10x) or so higher energy than on-chip memory accesses, this enables the system designer to minimize energy and maximize performance.

ACE Additional Signals and Channels

AMBA 4 ACE is backwards -compatible with AMBA 4 AXI adding additional signals and channels to the AMBA 4 AXI interface. The AXI interface consists of 5 channels. In AXI, the read and write channels each have their own dedicated address and control channel. The BRESP channel is used to indicate the completion of write transactions.

ARADDR	Read address and command channel
RDATA	Read data channel
AWADDR	Write address and command channel
WDATA	Write data channel
BRESP	Write response channel

Figure 3 – AXI 5 channel interface

For further information on the AXI interface download the specification from the ARM website.

ACE adds 3 more channels for coherency transactions and also adds some additional signals to existing channels. The ACADDR channel is a snoop address input to the master. The CRRESP channel is used by the master to signal the response to snoops to the interconnect. The CDDATA channel is output from the master to transfer snoop data to the originating master and/or external memory.

ACADDR Coherent address channel. Input to master
CRRESP Coherent response channel. Output from master
CDATA Coherent data channel. Output from master
Figure 4 – ACE additional channels

As well as the additional three channels, there are additional signals added to the existing AXI channels.

Read address channel
ARSNOOP[3:0]
ARBAR[1:0]
ARDOMAIN[1:0]

Read data channel
RRESP[3:2]
RACK

Write address channel
AWSNOOP[2:0]
AWBAR[1:0]
AWDOMAIN[1:0]

Write data channel
WACK

ARSNOOP and AWSNOOP indicate the type of snoop transactions for shareable transactions on the read and write channels respectively. ARBAR and AWBAR are used for barrier signaling. ARDOMAIN indicates which masters should be snooped for snoop transactions and which masters must be considered for ordering of barrier transactions. RRESP has additional bits for shared read transactions that are indirectly driven by CRRESP outputs from a snooped master. In addition to the full ACE interface, the AMBA 4 specification also defines ACE-Lite which has the additional signals on the existing channels but not the new channels. ACE-Lite masters can snoop ACE –compliant masters, but cannot themselves be snooped.

Cortex-A15 Inter-Cluster Coherency, big.LITTLE coherency and I/O Coherency

The Cortex-A15 MPCore processor was the first ARM processor core to support AMBA 4 ACE. ACE enables expansion of the SoC beyond the 4-core cluster of the ARM MPCore technology. Accesses to shared memory are broadcast on the ACE interface enabling coherency between multiple Cortex-A15 clusters, enabling scaling to 8 cores and beyond. But ACE is not limited to coherency between identical CPU clusters, it can also support full coherency between dissimilar CPUs and I/O coherency for accelerators. For example it supports coherency between a Cortex-A15 cluster and a Cortex-A7 cluster (which also supports the ACE coherency protocol) in a big.LITTLE system.

All shared transactions are controlled by the ACE coherent interconnect. ARM has developed the CCI-400 Cache Coherent Interconnect product to support up to two clusters of CPUs and three additional ACE-Lite I/O coherent masters. In this example diagram below we show the Mali-T604 GPU, which is

ACE-Lite –compliant and can therefore snoop the CPU caches. A further ACE-Lite port is dedicated in this case to a coherent I/O (for example a Gigabit Ethernet interface) and the third ACE-Lite interface is shared between other masters. The CCI-400 supports 3 master interfaces connecting to memory controller and peripherals. In this example two master ports connect to a dual-channel DMC-400 Dynamic Memory Controller supporting LP-DDR2 or DDR3 memory and the other is used to support an additional interconnect to support all other on-chip slave peripherals.

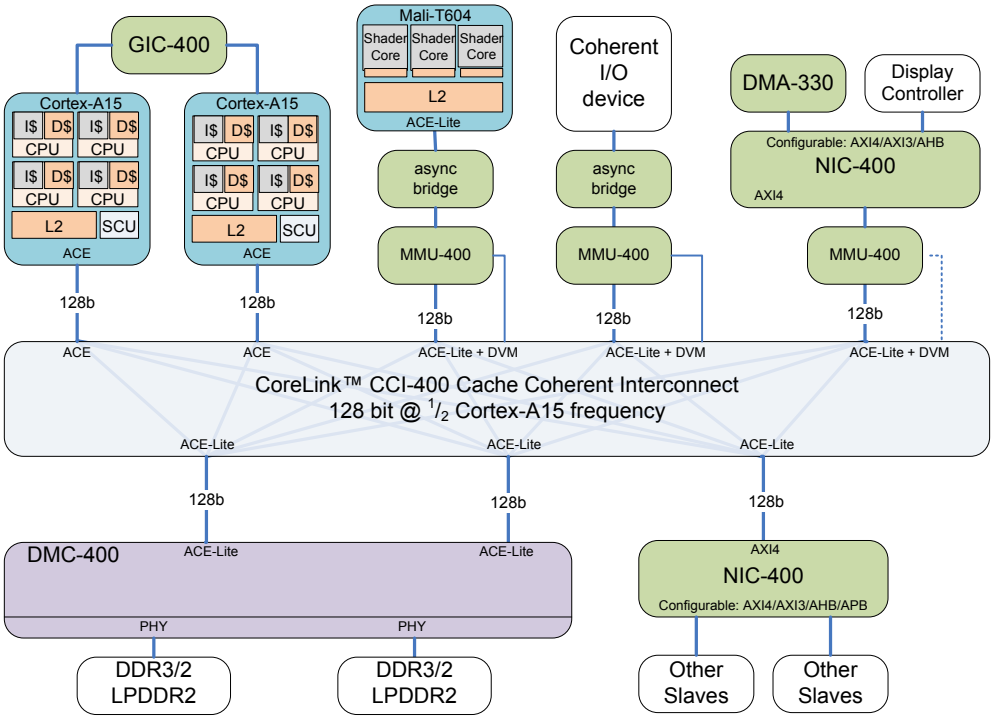


Figure 4 – Example Cortex-A15 Coherent System with CCI-400 Cache Coherent Interconnect

AMBA 4 ACE Transactions

ACE introduces a large number of new transactions to AMBA 4. To understand them, it's best to think of them in groups.

Non-shared

The first group is Non-Shared and this consists of *Read* and *Write* transactions. These are the existing AXI read and write used for non-coherent, non-snooped transactions.

Non-cached

ReadOnce is used by masters reading data that is shared but where the master will not retain a copy, meaning that other masters with a cached copy do not need to change cache state. An example use could be a display controller reading a cached frame buffer.

WriteUnique is used by uncached masters writing shared data, meaning that all other cached copies must be cleaned. Any dirty copy must be written to memory and clean copies invalidated.

WriteLineUnique is like *WriteUnique* except that it always operates on a whole cache line. Since it operates on a whole cache line, other dirty copies need not be written back.

Note: *Read*, *Write*, *ReadOnce* and *WriteUnique* are the only ACE (external) transactions that can operate on data that is not a whole cache line.

Shareable Read

ReadShared is used for shareable reads where the master can accept cache line data in any state

ReadClean is used for shareable reads where the master wants a clean copy of the line. It cannot accept data in either of the dirty states (for example it has a write-through cache).

ReadNotSharedDirty is used for shareable reads where the master can accept data in any cache state **except** the SharedDirty state. It cannot accept data in the SharedDirty state (for example it uses the MESI model rather than the MOESI model). The Cortex-A15 processor core uses the MESI model and therefore uses *ReadNotSharedDirty* for shared read transactions.

Shareable Write

Note: Transactions in the 'shareable write' group are all actually performed on the read channel as they are used for **obtaining the right** to write a line in an internal write-back cache.

MakeUnique is used to clean all other copies of a cache line before performing a shareable write. Once all other copies are cleaned and invalidated the master can allocate a line in its cache and perform the write.

ReadUnique is like *MakeUnique* except it also reads the line from memory. *ReadUnique* is used by a master prior to performing a partial line write, where it is updating only some of the bytes in the line. To do this it needs a copy of the line from memory to perform the partial line write to; unlike *MakeUnique* where it will write the whole line and mark it dirty, meaning it doesn't need the prior contents of memory.

CleanUnique is used for partial line writes where the master already has a copy of the line in its cache. It's therefore similar to *ReadUnique* except that it doesn't need to read the memory contents, only ensure any dirty copy is written back and all other copies are invalidated. Note that if the data is dirty in another cache (ie in the SharedDirty state), the master initiating the *CleanUnique* transaction will be in the SharedClean state **but will have a current up-to-date copy of data**, as the ACE protocol ensures that **all dirty copies of a line in all caches are the same at all times**. It's possible to use *ReadUnique* instead of *CleanUnique* at the expense of an uneccessary external memory read. The Cortex-A15 processor doesn't use *CleanUnique* but uses *ReadUnique*.

Write-back transactions

WriteBack is a transaction to write back an entire dirty line to memory and is normally the result of an eviction of a dirty line as the result of allocating a new line.

WriteClean is like *WriteBack* but indicates the master will retain a copy of the clean line afterwards. This could be the result of a master performing eager writeback, ie speculatively writing back lines when not strictly necessary in the hope that the line will not be updated again prior to eviction. *WriteClean* is provided in addition to *WriteBack* to enable an external snoop filter to track cache contents.

Evict does not write back anything. *Evict* indicates that a clean line has been replaced ('evicted'), for example as the result of an allocation. *Evict* is provided purely as a mechanism for external snoop filters to track what's in the cache.

Cache Maintenance

CleanShared is a broadcast cache clean causing any cache with a dirty copy to write the line to memory. Caches can retain the data in a clean state.

CleanInvalid is similar to CleanShared except that caches must invalidate all copies after any dirty data is written back.

MakeInvalid is a broadcast invalidate. Caches are required to invalidate all copies and do not need to write back any data to memory even if it's dirty data. This can be the result of a CMO (Cache Maintenance Operation) or can be generated by the interconnect to invalidate caches in response to a WriteLineUnique from a master.

Note that ACE-Lite can perform only the transactions within the ***Non-Shared***, ***Non-cached and Cache Maintenance*** transaction groups.

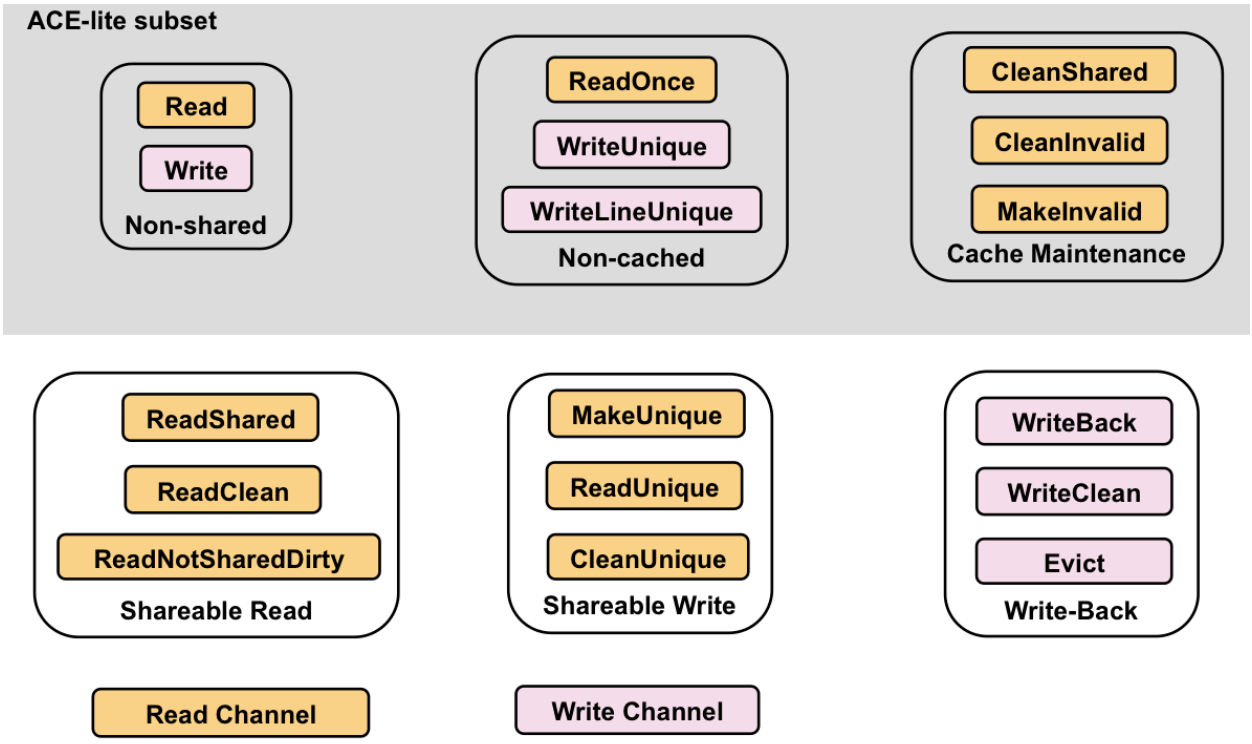


Figure 5 – ACE Transaction Groups

ACE-Lite I/O Coherency

ACE-Lite masters have the additional signals on AXI channels but do not have the additional three ACE snoop channels. ACE-Lite masters can perform transactions only from the *Non-shared*, *Non-cached* and *Cache Maintenance* transaction groups. ACE-Lite enables uncached masters to snoop ACE coherent masters. This can enable interfaces such as Gigabit Ethernet to directly read and write cached data shared within the CPU. Going forwards, ACE-Lite is the preferred technique for I/O coherency and should be used where possible rather than the Accelerator Coherency Port for best power and performance. Cortex-A15 supports an optional ACP primarily for designs including legacy IP that is not ACE-Lite compliant or designs that are upgrades from other MPCore technology-enabled processors.

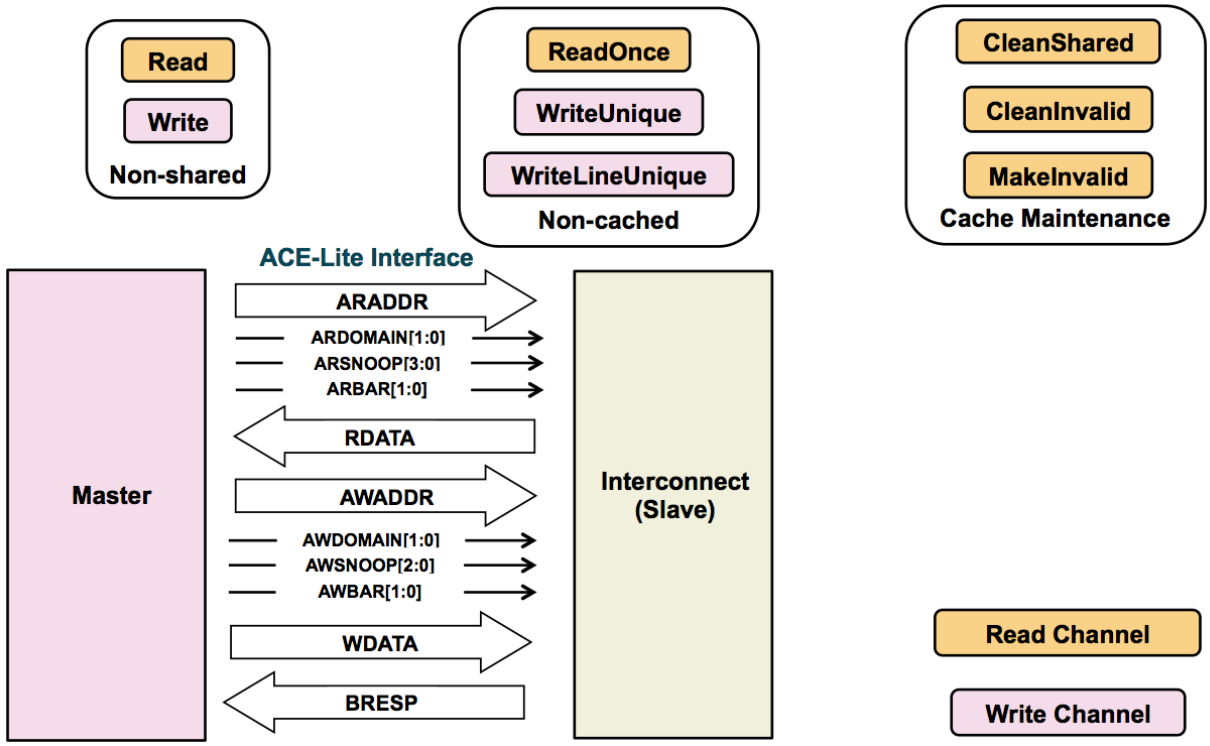


Figure 6 – ACE-Lite sub-set

Benefits of ACE-Lite I/O Coherency for Mali-T604

The ARM Mali-T604 GPU supports ACE-Lite I/O coherency. For each frame, the processor writes GPU commands and optionally vertices to memory. (Vertices are 'optional' as often they don't change from frame to frame). Without ACE-Lite coherency these commands would need to be flushed from the cache. Measurements taken on a dual-core Cortex-A9 at 1GHz with a 256K L2 cache showed that cache flushing can take of the order of 100µs. Cortex-A15 systems are expected to have significantly larger L2 caches, typically from 1MB to 4MB, meaning cache flushing could be expected to be commensurately longer. A 4MB L2 is 16 times larger than 256K L2. Even accounting for the higher performance of Cortex-A15, it is conceivable it could take around 10 times as long or of order 1ms to flush a 4MB L2 cache. (These times are intended to give a rough order of magnitude and are not necessarily accurate for any particular system). At 60 frames per second this needs to be done every frame, so it's possible the CPU could spend up to around 5-6% of the time cache flushing.

With ACE-Lite I/O coherency, processor cache flushing is eliminated. Mali-T604 can read commands and vertices using the *ReadOnce* transaction. If data hits in the CPU cache then it is passed directly from the snoop data channel CDATA via the interconnect to Mali-T604. If the data in the cache was dirty (which it likely was) then it is also written to memory by the interconnect. However the memory read that would be required without ACE-Lite coherency is eliminated meaning that for snoop hits there is just one memory write compared with one write and one read without ACE-Lite. Thus external memory accesses are reduced. Since memory bandwidth is increasingly the main system performance bottleneck and external memory accesses use far more energy than internal accesses, overall system performance and energy-efficiency are enhanced.

Barriers

In systems with shared-memory communications and out-of-order execution, barriers are necessary to ensure correct operation. The ARM architecture defines two types of barrier, DMB and DSB. DMB, or Data Memory Barrier ensures that all memory transactions prior to the barrier are visible by other masters before any after it. Hence the DMB Data Memory Barrier prevents any re-ordering about the DMB. Everything before the DMB must complete before anything after the DMB. With the extension of coherency outside the MPCore cluster, it's necessary to broadcast barriers on the ACE interface. DMB barriers may define a subset of masters that must be able to observe the barrier. This is indicated on the AxDOMAIN signals. These can indicate Inner, Outer, System, or Non-shareable. The Inner domain shares both code and data (ie running the same operating system instance), the Outer domain shares data but not code.

The DSB or data synchronization barrier is used to stall the processor until previous transactions have complete. This contrasts with DMB where the transaction can flow on the pipelined interconnect but no re-ordering is allowed about the DMB. DSB would typically be used when interacting with hardware that has an interface in addition to via memory. For example the DSB could be used to ensure data written to a DMA command buffer in memory has reached its destination before kicking off the DMA via a peripheral register. DSB is the most time-consuming barrier since it stops the processor until transactions are complete, and therefore should only be used where necessary. Barriers are sent on ACE Read and Write channels simultaneously using ARBAR and AWBAR signaling.

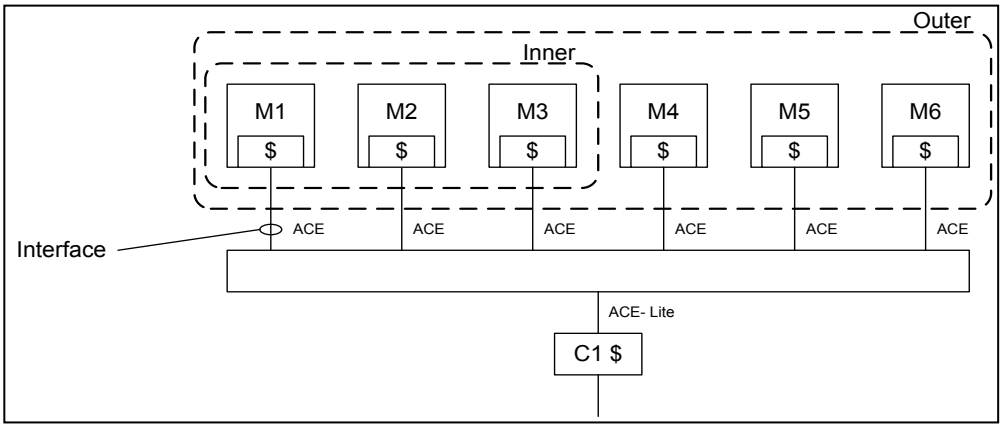


Figure 7 - Domains

Distributed Virtual Memory (DVM)

The ability to build multi-cluster coherent CPU systems sharing a single set of MMU page tables in memory brings the requirements to ensure TLB coherency. A TLB or *Translation Look-aside Buffer* is a cache of MMU page tables in memory. When one master updates page tables it needs to invalidate TLBs that may contain a stale copy of the MMU page table entry. Distributed Virtual Memory support in ACE consists of broadcast invalidation messages. DVM messages can support TLB Invalidation, branch predictor, virtual or physical instruction cache invalidation (for when a processor has written code to memory) and *Synchronization,* which waits for all previous DVM commands to complete. DVM messages are sent on the Read channel using ARSNOOP signaling. A system MMU (SMMU) may make use of the TLB invalidation messages to ensure its entries are up-to-date. Instruction cache and branch-predictor invalidation messages are used for inter-processor communications ensuring instruction-side invalidation occurs in all processors in the system, even across different clusters.

ARM AMBA Cache Coherency Interconnect CCI-400 and Series 400 AMBA 4 Fabric IP

The ARM CCI-400 Cache Coherent Interconnect supports two full ACE coherency ports to enable two CPU clusters enabling the construction of cache-coherent systems up to 8 CPU cores. Each cluster could consist of either Cortex-A15 or Cortex-A7 processors, enabling systems up to 8 Cortex-A15 CPU cores, or big.LITTLE systems with up to four Cortex-A15 cores and four Cortex-A7 cores. The number of Cortex-A15 and Cortex-A7 cores does not need to be the same. The CCI-400 coordinates and controls all coherent accesses between clusters. It also supports 3 ACE-Lite interfaces for ACE-Lite I/O masters like Mali-T604 and other I/O coherent devices. With ACE-Lite support, Mali-T604 can snoop CPU caches. Standard AXI masters may be connected but without cache coherency functionality. The CCI-400 Cache Coherent Interface has 3 master interfaces to connect to 3 slaves. For example, two master ports can be used for a dual-channel LP-DDR2 or DDR3 SDRAM interface with the third connected to a peripheral interconnect for other on-chip peripherals. CCI-400 supports end-to-end quality-of-service (QoS), supporting AXI QoS priority signaling, and QoS Virtual Networks (QVN) which prevents low-priority transactions in flight slowing high-priority QoS transactions due to head-of-line blocking. The GIC-400 interrupt controller supports distributing interrupts to multiple CPU clusters. The MMU-400 provides second-level address translation for master devices in systems making use of ARM Virtualization Extensions present in the Cortex-A15 and Cortex-A7 processors. With virtualization, the first-level address translation controlled by the OS is followed by a second stage translation controlled by the hypervisor. The MMU-400 performs intermediate physical address to physical address translation for masters and is controlled fully by the hypervisor, the OS does not need to know of its existence. The DMC-400 dynamic memory controller supports multiple channels of LP-DDR2 and DDR3 memory. Together, the CCI-400, GIC-400, MMU-400 and DMC-400 along with the NIC-400 AMBA 4 configurable network interconnect create a complete CPU sub-system solution for cache-coherent Cortex-A15 processor-based AMBA 4 systems scaling up to 8 processor cores, and for big.LITTLE systems with up to four Cortex-A15 and four Cortex-A7 processors.

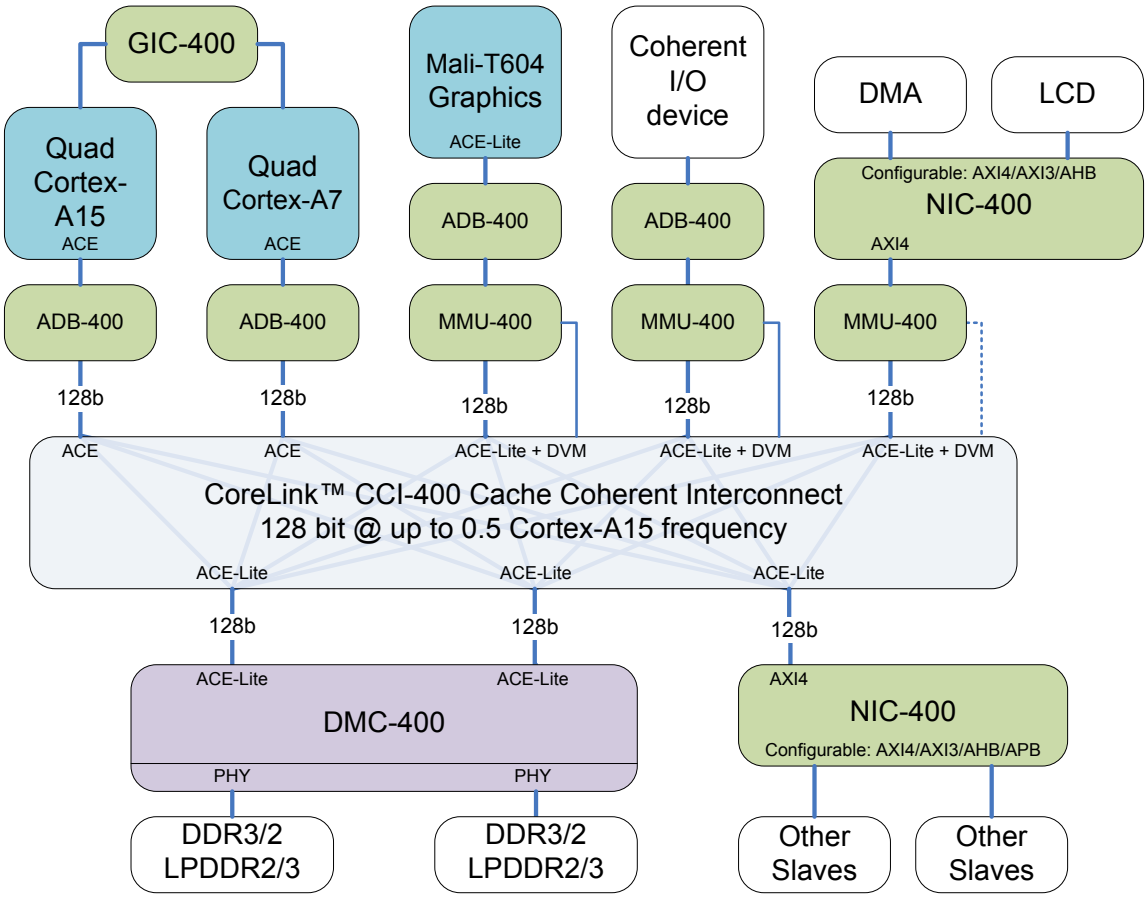


Figure 8 – AMBA 4 400-Series IP for Cache Coherent Compute Sub-Systems

Summary

AMBA 4 ACE adds system-level coherency support to the AMBA 4 specifications. It supports cache-coherent transactions, cache maintenance transactions, barrier signaling and distributed virtual memory (DVM) messages to enable TLB management and invalidation.

ACE has been designed to support a wide range of coherent masters with differing capabilities, not just processor cores such as the Cortex-A15 MPCore processor. ACE supports coherency between dissimilar processors such as the Cortex-A15 and Cortex-A7 processors enabling ARM big.LITTLE technology. It supports I/O coherency for un-cached masters, supports masters with differing cache line sizes, differing internal cache state models, and masters with write-back or write-through caches. ACE has been in development for more than 4 years and benefits from the input and experience of more than 30 reviewing companies. The ACE specification is now available for download on the ARM website via a click-thru licensing agreement, along with SystemVerilog protocol assertions here: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html

The CoreLink 400-Series AMBA 4 IP from ARM together form a complete CPU compute sub-system solution scaling to 8 processor cores supporting cache-coherent big.LITTLE technology and is available for licensing from ARM now. More information on big.LITTLE technology is available on the ARM website: http://www.arm.com/products/processors/technologies/biglittleprocessing.php