Memory access ordering part 3 - memory access ordering in the ARM Architecture

Posted by **ARM Leif**, 19 October 2011

In my previous posts, I have introduced the concept of memory access ordering and discussed barriers and their implementation in the Linux kernel. I chose to do it in this order because I wanted to start by communicating the underlying concepts before I went into detail about what the ARM architecture does about memory ordering. This post goes into the juicy bits of what this actually means and how this is handled in the ARM architecture.

Two separate concepts are relevant to memory access ordering in the ARM architecture — *memory types* and *shareability domains*. These progressively made their explicit entry into the ARM architecture in versions 6 and 7, implemented by the ARM11 and Cortex family of processors respectively.

Enter the abstract

When describing many of the concepts mentioned in this post, the ARM Architecture Reference Manual makes frequent use of the words/phrases *observer*/*observers* and *is observed to* or *must observe*. In practice, this refers to Master bus interfaces and how the devices controlling those interfaces, as well as the interconnect, must handle transactions. Only a Master interface can observe a transaction. Since all bus transactions are initiated by a Master, the ordering of accesses arriving at Slave interfaces can be inferred from the Master ordering rules. Note that transaction ordering does not refer simply to the order in which transactions leave a Master interface - they can often be reordered in the memory system, and can be observed in different order by different Masters where not explicitly ordered.

Memory types

Before the ARMv6 architecture, not much explicit was defined about the out-of-ordering of memory accesses — the Sequential Execution Model was assumed to apply to all instructions. Processors that implemented caches and write buffers could mark regions of memory as being *cacheable* or *bufferable* without greater side effects than the obvious ones.

However, for modern processors that implement multiple cores, out-of-order execution or simply permits certain accesses to be buffered and others to happen synchronously, it is vital that certain rules are defined for what constraints apply to:

- the order in which the memory accesses of one core relate to the surrounding instructions
- the order in which the memory accesses of one core can be observed by other cores within a multi-core processor
- the order in which the memory accesses of a processor can be observed by other Masters on the system bus

I will not go into full detail of the different memory types in this post (that is enough information for its own post), but I will give a quick overview of the points relevant for this post. Memory types (and their additional attributes such as cache policy) are configured in the translation tables the operating system sets up for the MMU.

Normal memory

Normal memory is effectively for all of your data and executable code. This memory type permits speculative reads, merging of accesses and (if interrupted by an exception) repeating of writes without side effects. Accesses to Normal memory can always be buffered, and in most situations they are also cached - but they can be configured to be uncached. There is no implicit ordering of Normal memory accesses, beyond pure address dependencies and control dependencies. When not explicitly restricted, the only limit to how out-of-order non-dependent accesses can be is the processor's ability to hold multiple live transactions.

Device memory and Strongly-ordered memory

The Device and Strongly-ordered memory types are used with memory mapped peripherals or other control registers. For the purposes of this post, Device and Strongly-ordered memory are quite similar, and with the ARMv7-A Large Physical Address Extension (LPAE), this becomes even more true since processors implementing the LPAE treat Device and Strongly-ordered memory regions identically. ARMv7-A processors that do not implement the LPAE can set device memory to be Shareable or Non-shareable.

Accesses to these types of memory must happen exactly the number of times that executing the program suggests they should. Two writes to the same location must be performed as two writes, and two reads from the same location must both take place. This is obviously important when you are accessing peripheral control registers. There is however no guarantee about ordering between memory accesses to different devices, or usually between accesses of different memory types.

Barriers

Barriers were introduced progressively into the ARM architecture.

- Some ARMv5 processors, such as the ARM926EJ-S, implemented a Drain Write Buffer cp15 operation, which halted execution until any buffered writes had drained into the external memory system.
- With the introduction of the ARMv6 memory model, this operation was redefined in more architectural terms and became the Data Synchronization Barrier. ARMv6 also introduced the new Data Memory Barrier and Flush Prefetch Buffer cp15 operations.
- ARMv7 evolved the memory model somewhat, extending the meaning of the barriers - and the Flush Prefetch Buffer operation was renamed the Instruction Synchronization Barrier.

- ARMv7 also allocated dedicated instruction encodings for the barrier operations. Use of the cp15 operations is now deprecated and software targeting ARMv7 or later should use the DMB, DSB and ISB mnemonics.
- And finally, ARMv7 extended the Shareability concept to cover both Inner-shareable and Outer-shareable domains (see below). This together with AMBA4 ACE gives us barriers that propagate into the memory system.

So, what are these barriers then, and what do they do?

Instruction Synchronization Barrier (ISB)

The Instruction Synchronization Barrier ensures that any subsequent instructions are fetched anew from cache in order that privilege and access is checked with the current MMU configuration. It is used to ensure any previously executed context changing operations (including cp15 operations) will have completed by the time the ISB completed.

Access type and domain are not really relevant for this barrier. It is not used in any of the Linux memory barrier primitives, but appears here and there in memory management, cache control and context switching code.

Data Memory Barrier (DMB)

The basic functionality of a DMB is as follows:
It prevents reordering of data accesses instructions across itself. All data accesses by this processor/core before the DMB will be visible to all other masters within the specified shareability domain before any of the data accesses after it. It also ensures that any explicit preceding data (or unified) cache maintenance operations have completed before any subsequent data accesses are executed.

The DMB instruction takes two optional parameters: an operation type (stores only - 'ST' - or loads and stores) and a domain. The default operation type is loads and stores and the default domain is System. So, in effect `DMB` is shorthand for `DMB SY`. All possible combinations of types and domains are legal operations on any processor, even if it does not implement the specific functionality described, and can be substituted internally for any stronger barrier.

In the Linux kernel, the DMB instruction is used for the `smp_*mb()` macros.

Data Synchronization Barrier (DSB)

The Data Synchronization Barrier enforces the same ordering as the Data Memory Barrier, but it also blocks execution of *any* further instructions until synchronization is complete. It also waits until all cache and branch predictor maintenance operations have completed for the specified shareability domain. If the access type is load and store then it also waits for any TLB maintenance operations to complete.

In the Linux kernel, the DSB instruction is used for the `*mb()` macros.

Shareability domains

The ordering of memory accesses in the ARM architecture takes place within what is called a *Shareability domain*. Shareability domains define "zones" within the bus topology within which memory accesses are to be kept consistent (taking place in a predictable way) and potentially coherent (with hardware support). Outside of this domain, observers might not see the same order of memory accesses as inside it.

The following table shows the different shareability options available in an ARMv7-A system:

Domain	Abbreviation	Description
Non-shareable	NSH	A domain consisting only of the local agent. Accesses that never need to be syncronized with other cores, processors or devices. Not normally used in SMP systems.
Inner Shareable	ISH	A domain (potentially) shared by multiple agents, but usually not all agents in the system. A system can have multiple Inner Shareable domains. An operation that affects one Inner Shareable domain does not affect other Inner Shareable domains in the system.
Outer Shareable	OSH	A domain almost certainly shared by multiple agents, and quite likely consisting of several Inner Shareable domains. An operation that affects an Outer Shareable domain also implicitly affects all Inner Shareable domains within it (but does not otherwise behave as an Inner Shareable operation). For processors such as the Cortex-A15 MPCore that implement the LPAE, all Device memory accesses are considered Outer Shareable. For other processors, the shareability attribute can be explicitly set (to shareable or non-shareable). Devices within an Outer Shareable domain will normally be complex enough to have a concept of memory management and cache coherency (for example Mali graphics accelerators), although they might not be fully integrated in it.
Full system	SY	An operation on the full system affects all agents in the system; all Non-shareable regions, all Inner Shareable regions and all Outer Shareable regions. Simple peripherals such as UARTs, and several more complex ones, are not normally necessary to have in a restricted shareability domain.

ARMv6 architecture does not support a separate outer shareable domain.

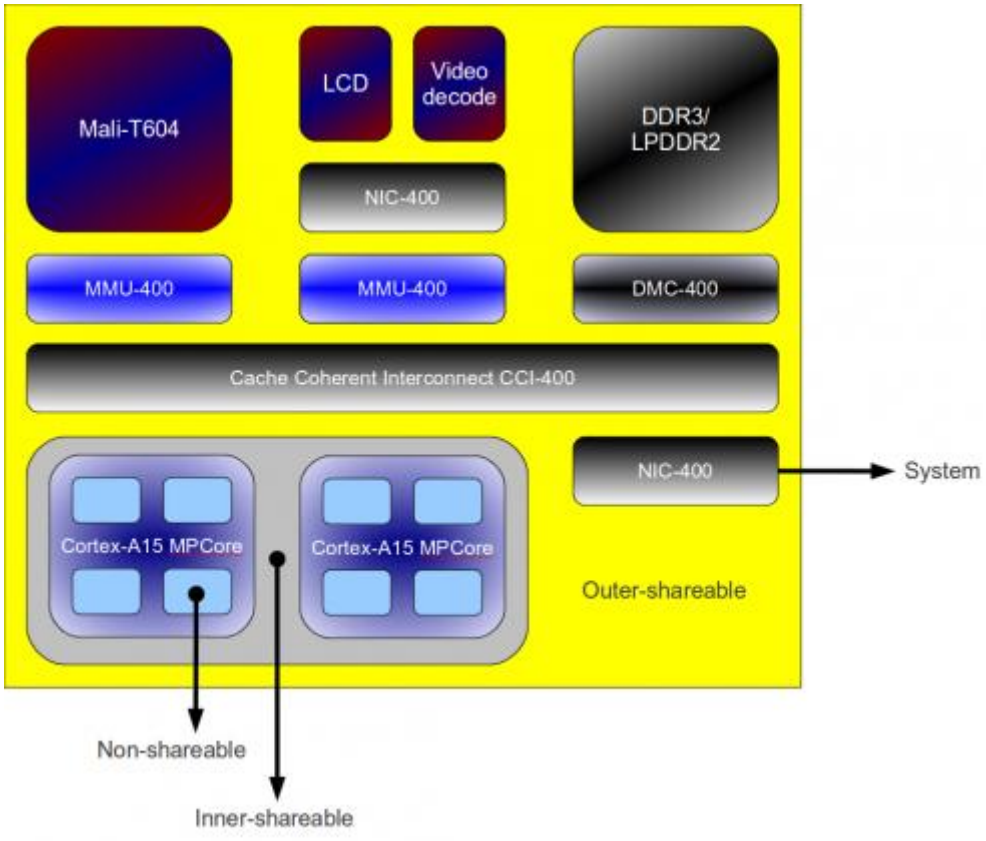
Shareability is effectively assigned to each memory transaction in the system, based on:

- Memory attributes for the region accessed (determined by MMU translation tables)
- Core configuration (can differ between cores within one multi-core processor)
- Implementation of interconnect

- Integration between interconnect and the masters connected to it

But there are also specific operations (instructions or cp15 configuration options) that can be performed with a domain defining their scope.

The diagram below shows an example system and one way the shareability domains can be implemented. Here we have each individual execution unit within the MPCore clusters possessing their own internal Non-shareable domain. The two MPCore clusters have been configured to make up one inner-shareable domain. There is an outer-shareable domain holding graphics and video accelerators and graphics output as well as memory. And finally, anything not in this subsystem is simply part of the System domain.



Barrier domains and levels

Unless any additional parameters are specified, the barriers apply to the System domain. All barrier instructions can take a domain specifier, and it is architecturally defined that any unsupported specifier will be treated as if it specified System. The Data barriers can additionally take a separate parameter (ST for "store") to indicate that this barrier should only affect store accesses, and that loads can be freely reordered around them.
For example, on a processor that does not distinguish between shareability domains `DMB ISHST` will execute simply as if it was `DMB SY`, whereas on one that does, it will be a lot more efficient.

Example:

```
DMB     OSST
```

External caches

With AMBA4 ACE

The Cortex-A15 implements the AXI Coherency Extension, which makes barriers propagate through the interconnect. This makes it easy to maintain ordering within the ACE-aware portion of the system.

Before AMBA4 ACE

Before AMBA4 ACE, there was still a chance that bufferable memory transactions could be reordered in the external memory system even after barrier instructions had ensured that they left the Master interface in the correct order. On the ARM Versatile Express development platform with the 4xCortex-A9 module, accesses can be reordered in the level 2 cache controller. For example, a write to a DMA descriptor can be overtaken by a subsequent write to a control register to initiate the DMA transaction. This is resolved by the `*mb()` macros expanding to perform an explicit "outer sync" with the PL310 when `CONFIG_ARM_DMA_MEM_BUFFERABLE` is set in the kernel configuration file.

Summary

Memory access ordering is a complex topic, but hopefully this 3-part series has provided a useful introduction. For complete and proper information of the memory model of the ARM architecture and the ordering requirements (and tools) for the AMBA interconnect, please see the resources listed below.

References

- Barrier litmus test & cookbook
- ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition
- Cortex-A Series Programmer's Guide
- AMBA4 spec

Leif, Principal Software Engineer, ARM, he first started playing around with UNIX in 1994, installing Linux in 1995, in 1999 graduated to Linux on embedded systems, and moved to doing it for a living in 2001. After four years of that, he packed up and moved to Cambridge, UK, to work for ARM. He is currently working on improving ARM multi-core support in (mostly) open source software. He is way too fond of blue LEDs (but any color will do if they are flashing), and is cowardly refusing to pick sides in the Linux/BSD war.