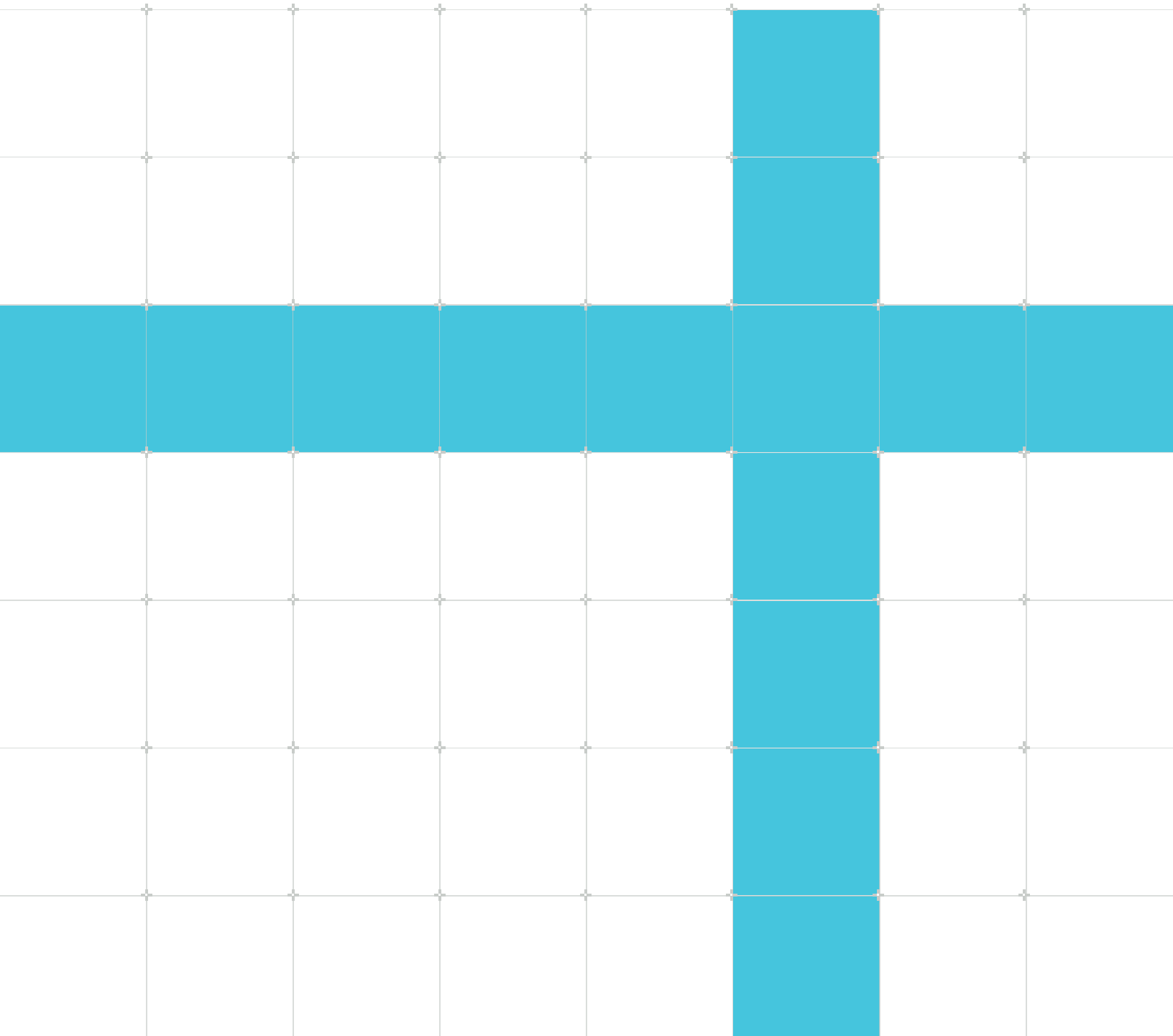
arm

Learn the architecture - ARMv8-A memory systems

Version 1.1

Issue 02
100941_0101_02_en



Learn the architecture - ARMv8-A memory systems

Release information

Document history

Issue	Date	Confidentiality	Change
0101-02	1 April 2022	Non-Confidential	Initial release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Armv8-A memory systems

This guide introduces memory systems in the Armv8-A architecture. These systems are detailed through The memory model, Memory types, Memory attributes and Barriers.

You must understand the operation of the memory system and access ordering in cases where your code interacts directly either with the hardware or with code executing on other cores, or if it directly loads or writes instructions to be executed, or modifies translation tables.

If you are an application developer, hardware interaction on an OS such as Linux is probably through a device driver. The interaction with other cores is through Pthreads or another multithreading API and the interaction with a paged memory system is through the operating system. In this case, the memory ordering issues are taken care of by the relevant code, however, this is not the case for all operating systems and you must check whether the same is true for the OS you work with.

However, if you are, for example, writing an operating system kernel or device drivers, or implementing a hypervisor, you must have a good understanding of the memory ordering rules of the ARM architecture.

Some reordering required when your code requires explicit ordering of memory accesses to be seen by cores or devices in the system.

2. The memory model

Compilers give you a wide range of options that aim to increase the speed, or reduce the size, of the executable files they generate. For each line in the source code, there are many possible choices of assembly instructions that could be used.

The Armv8-A architecture employs a weakly ordered model of memory. This means that the order of memory accesses is not necessarily required to be the same as the program order for load and store operations.

During the optimization process, the processor and system elements can reorder memory read operations with respect to each other to improve data throughput. Writes can also be reordered. This means that the required bandwidth between the processor and external memory can be reduced and the long latencies that are associated with such external memory accesses are hidden.

To ensure that reordering can take place, there must be memory types that allow such optimizations to take place in them.

Hardware can reorder reads and writes to Normal memory. Reads and writes can also be ordered by address dependencies, and half barriers. However, the existence of either data dependencies or explicit memory barrier instructions can override this. Certain situations require stronger ordering rules. You can provide information to the core about this through the memory type attribute of the translation table entry that describes that memory.

High-performance systems can support techniques such as speculative memory reads, multiple issuing of instructions, or out-of-order execution and these, along with other techniques, offer further possibilities for hardware reordering of memory access:

Multiple issue of instructions

Processors can issue and execute multiple instructions per cycle. Some instructions can reach the execution stage of the pipeline in parallel, as a result they may execute in a different order to their order in the program.

Out-of-order execution

Many processors support out-of-order execution of non-dependent instructions. Because of the multiple issue of instructions, some instructions can stall in the execution stage, while they wait for others to complete, but these will not stop non-dependent instructions from completing.

This can also change the program order.

Speculation

When the processor encounters a conditional instruction, such as a branch, it can begin to execute instructions before it knows for certain whether that particular instruction is executed or not.

The result is therefore available sooner if conditions prove that the speculation was correct.

Instruction fetch speculation is the fetch of instructions that are not defined by the program execution order.

Speculative loads

If a load instruction that reads from a Cacheable location is speculatively executed, this can result in a cache linefill and potential eviction of an existing cache line.

Load and store optimizations

As reads and writes to external memory can have a long latency, processors can reduce the number of transfers for example, by merging together several stores into one larger transaction.

External memory systems

In many System on Chip (SoC) devices, there are several agents capable of initiating transfers and multiple routes to the slave devices that are read or written.

Some of these devices, such as a DRAM controller, are capable of accepting simultaneous requests from different masters. Transactions can be buffered, or reordered.

This means that accesses from different masters can therefore take varying numbers of cycles to complete and might overtake each other.

Cache coherent cluster processing

In a cluster, hardware cache coherency can migrate cache lines between cores.

Different cores might see updates to cached memory locations in a different order to each other.

Also, these might not be coherent with external memory.

Optimizing compilers

An optimizing compiler can reorder instructions to hide latencies or make best use of hardware features.

It can often move a memory access forward, to make it earlier, and give it more time to complete before the value is required.

They can also have instruction scheduling that can take advantage of specific core multi-issue pipelines.

In a single core system, the effects of such reordering are transparent to the programmer, because the individual processor can check for hazards and ensure that data dependencies are respected. However, in cases where you have multiple cores that communicate through shared memory, or share data in other ways, memory ordering considerations become more important.

3. Memory types

The ARMv8-A architecture defines two mutually exclusive memory types, Normal and Device and all regions of memory are configured as one or the other of these two types.

Normal memory

Normal memory is used for all code and for most data regions in memory. Examples of Normal memory include areas of RAM, Flash, or ROM in physical memory. This kind of memory provides the highest processor performance as it is weakly ordered and the compiler can perform more optimizations. The processor can reorder, repeat, and merge accesses to Normal memory.

The processor can speculatively access address locations that are marked as Normal, so that data or instructions can be read from memory without being explicitly referenced in the program, or before the actual execution of an explicit reference. Such speculative accesses can occur as a result of branch prediction, speculative cache linefills, out-of-order data loads, or other hardware optimizations.

For best performance, always mark application code and data as Normal. In circumstances where enforced memory ordering is required, do this by using explicit barrier operations. Normal memory accepts weakly ordered memory accesses without any issues. There is no requirement for Normal accesses to complete in order with respect to either other Normal accesses or to Device accesses.

However, the processor must always handle hazards that are caused by address dependencies. For example, consider the following simple code sequence:

```
STR X0, [X2]
LDR X1, [X2]
```

A single processor running a single thread always ensures that the value that is placed in X1 is the value that was written from register X0 through to the address stored in X2.

This applies to more complex dependencies. Consider the following code:

```
ADD X4, X3, #3
ADD X5, X3, #2
...
STR X0, [X3]
STRB W1, [X4]
LDRH W2, [X5]
```

In this case, the accesses take place to addresses that overlap each other. The processor must ensure that the memory is updated as if the `STR` and `STRB` occurred in order, so that the `LDRH` returns the most up-to-date value. It would still be valid for the processor to merge the `STR` and `STRB` into a single access that contained the latest, correct data written.

Device memory

The Device memory type is used with memory-mapped peripherals and all memory regions where an access might have a side effect. For example, a read to a timer is not repeatable, as it returns

different values for each read. A write to a control register can trigger an interrupt. The Device memory type imposes more restrictions on the core.

Accesses to these types of memory must occur 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 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.

Speculative data accesses cannot be performed to regions of memory that are marked as Device.

Trying to execute code from a region marked as Device is UNPREDICTABLE.

When an instruction can result in UNPREDICTABLE behavior, the ARM architecture can specify a narrow range of permitted behaviors. This is defined as a number of **CONSTRAINED UNPREDICTABLE** behaviors. The implementation can either handle the instruction fetch as if it were to a memory location with the normal Non-cacheable attribute, or it can take a permission fault.

There are four different types of device memory, defining the rules which memory accesses must obey.

As the memory type weakens those rules are relaxed:

- Device-nGnRnE is the most restrictive.
- Device-nGnRE
- Device-nGRE
- Device-GRE least restrictive

The letter suffixes refer to the following three properties:

Gathering or non-Gathering

Gathering or non-Gathering (G or nG) determines whether multiple accesses can be merged into a single transaction for this memory region. If the address is marked as non-Gathering (nG), then the number and size of accesses that are performed to that location must exactly match the number and size of explicit accesses in the code. If the address is marked as Gathering (G), then the processor can, for example, merge two bytes writes into a single halfword write.

Reordering

Reordering (R or NR) determines whether accesses to the same device can be reordered with respect to each other. If the address is marked as non-Reordering (NR), then accesses within the same block always appear on the bus in program order. The size of this block is **IMPLEMENTATION DEFINED**. Where the size of this block is large, it could span several table entries. In this case, the ordering rule is observed with respect to any other accesses also marked as NR.

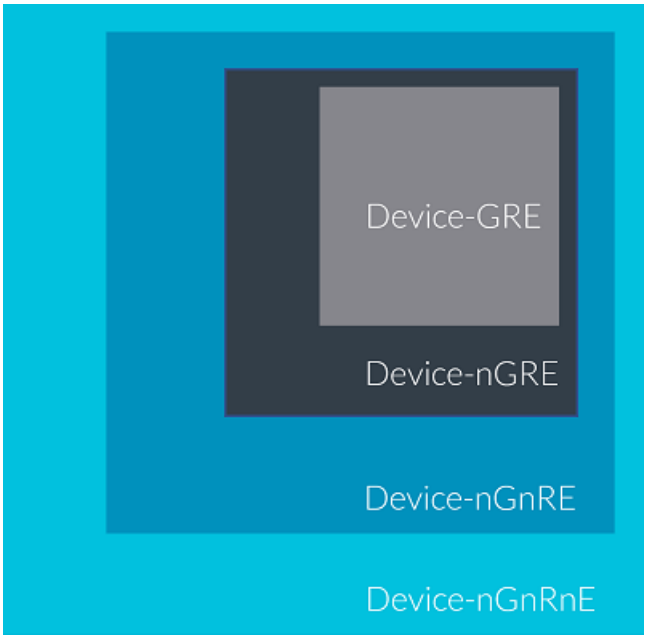
Early Write Acknowledgment

Early Write Acknowledgment (E or nE) determines whether an intermediate write buffer between the processor and the device being accessed is allowed to send an acknowledgment of a write completion.

If the address is marked as non-Early Write Acknowledgment (nE), then the write response must come from the peripheral. If the address is marked as Early Write Acknowledgment (E), then it is a buffer in the interconnect logic can signal write acceptance, before the write actually being received by the end device. This is essentially a message to the external memory system.

The following figure shows the four different types of device memory:

Figure 3-1: Device memory

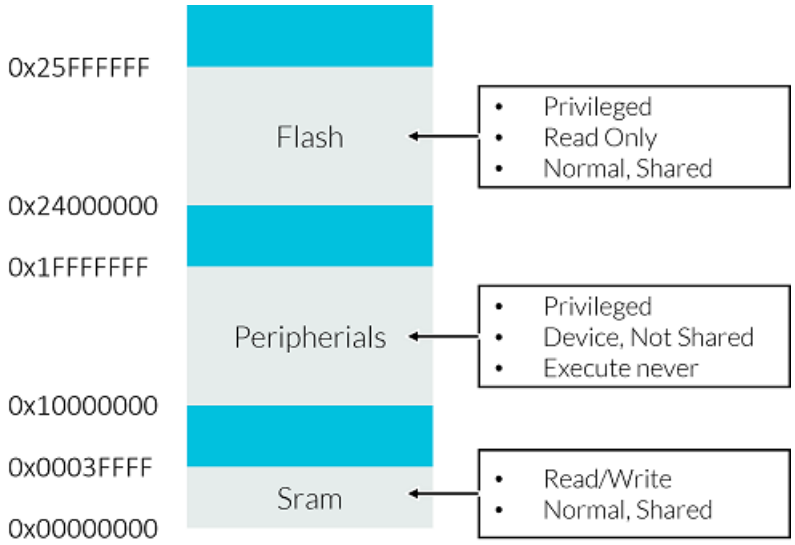


4. Memory attributes

The memory map of a system can be divided into several regions. Each region can have different memory attributes, such as access permissions that include read and write permissions for different privilege levels, memory type, and cache policies.

The following figure shows an example system memory map:

Figure 4-1: Example memory map



Functional pieces of code and data are grouped in the memory map and the attributes for each of these areas are controlled separately by the Memory Management Unit.

In addition to the memory type, memory attributes also provide control over cacheability, shareability, access, and execution permissions. Shareable and cache properties apply only to Normal memory. Device regions are always Non-cacheable and Outer-shareable. For Cacheable locations, you can use attributes to indicate cache allocation policy to the processor.

Cacheable and shareable memory attributes

Regions of memory that are marked as Normal can be specified as either cached or non-cached. Memory caching can be separately controlled through inner and outer attributes, for multiple levels of cache. The division between inner and outer is **IMPLEMENTATION DEFINED**, but typically the inner attributes are used by caches in the processor. The outer attributes are used by external memory where they can be used by caches external to the core or cluster.

The shareable attribute is used to define whether a location is shared with multiple cores. Marking a region as Non-shareable means that it is only used by a particular core, whereas marking it as Inner Shareable or Outer shareable, or both, means that the location is shared with other observers, for example, a GPU or DMA device might be considered another observer.

The division between inner and outer is also **IMPLEMENTATION DEFINED**. These attributes define sets of observers for which the shareability attributes make the caches transparent for data accesses. This also means that the system must provide hardware coherency management so that cores in the Inner Shareable domain see a coherent copy of locations that are marked as Inner Shareable.

If a processor or other master in the system does not support coherency, then it must treat the shareable regions as Non-cacheable.

Domains

Data memory accesses can take longer and consume more power with cache coherency hardware than they otherwise would do. This overhead can be minimized by maintaining coherency between a smaller number of masters while ensuring that they are physically close in the processor. For this reason, the architecture splits the system into domains, and makes it possible to limit the overhead to those locations where the coherency is required.

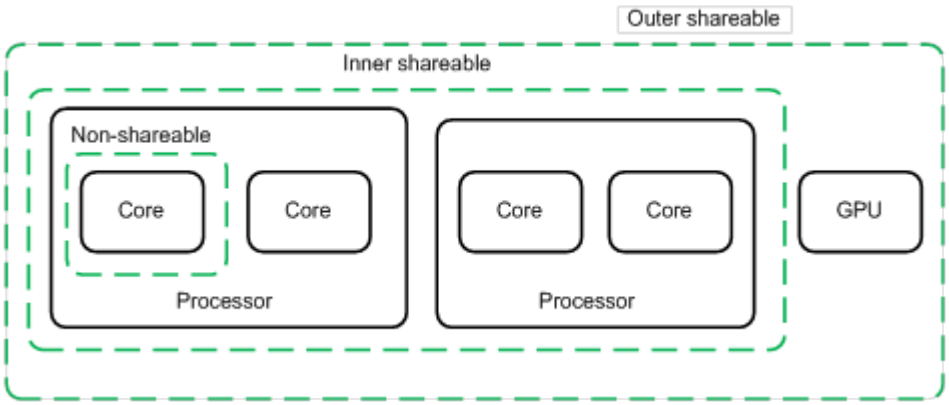
Shareability is assigned to each memory transaction in the system, based on:

- Memory attributes for the region accessed (determined by MMU translation tables).
- Core configuration (can differ between cores in a cluster).
- Implementation of interconnect.
- Integration between interconnect and the masters that are connected to it.

But there are also specific operations that can be performed with a domain defining their scope.

The following diagram shows shareability domain options:

Figure 4-2: Domains



The following shareability domain options are available:

Domain option	Description
Non-shareable	A domain consisting only of the local agent. Accesses that never require synchronization with other cores, processors, or devices. This domain is not typically used in SMP systems. **Note:** Symmetric Multi-Processing (SMP) is a software architecture that dynamically determines the roles of individual cores. Each core in the cluster has the same view of memory and of shared hardware. Any application, process, or task can run on any core and the operating system scheduler can migrate tasks between cores to achieve optimal system load.
Inner Shareable	A domain that is shared by other agents, but not necessarily all agent in the system. A system can have several Inner Shareable domains. An operation that affects one Inner Shareable domain does not affect other Inner Shareable domains in the system.
Outer Shareable	A domain that is shared by multiple agents that can consist of one or more Inner Shareable domains. An operation that affects an Outer Shareable domain also affects all Inner Shareable domains inside it. However, it does not otherwise behave as an Inner Shareable operation.
Full system	An operation on the full system affects all observers in the system.

5. Barriers

The Arm architecture includes barrier instructions to force access ordering and access completion at a specific point.

Barriers are used to prevent unsafe optimizations from occurring and to enforce a specific memory ordering. Use of unnecessary barrier instructions can therefore reduce software performance. Consider carefully whether a barrier is necessary in a specific situation, and if so, which is the correct barrier to use.

There are three types of barrier instruction.

Instruction Synchronization Barrier

Instruction Synchronization Barrier(ISB) is used to guarantee that any subsequent instructions are fetched, so that privilege and access are checked with the current MMU configuration. It is used to ensure any previously executed context-changing operations, such as writes to system control registers, have completed by the time the ISB completes.

In hardware terms, for example, this might mean that the instruction pipeline is flushed. Typical uses of this would be in memory management, cache control, and context switching code, or where code is being moved about in memory.

The following example shows how to enable the floating-point unit and SIMD, which you can do in AArch64 by writing to bit [20] of the CPACR_EL1 register. The ISB is a context synchronization event that guarantees that the enable is complete before any subsequent FPU or NEON instructions are executed.

```
MRS X1, CPACR_EL1               // Copy contents of CPACR to X1
ORR X1, X1, #(0x3 << 20)  // Write to bit 20 of X1. (Enable FPU and SIMD)
MSR CPACR_EL1, X1               // Write contents of X1 to CPACR
ISB
```

An ISB flushes the pipeline and ensures that the effects of any completed context-changing operation before the ISB are visible to any instruction after the ISB. Instructions from the cache or memory are refetched.

It also ensures that any context-changing operations after the ISB instruction only take effect after the ISB has completed and are not seen by instructions before the ISB.

This does not mean that an ISB is required after each instruction that modifies a processor register. For example, reads or writes to PSTATE fields, ELRs, SPs, and SPSRs always occur in program order relative to other instructions.

Data Memory Barrier

Data Memory Barrier (DMB) prevents reordering of data accesses instructions across the DMB instruction. Depending on the barrier type, certain data accesses, that is, loads or stores, but not

instruction fetches, performed by this processor before the DMB, are visible to all other masters within the specified shareability domain before certain other data accesses after the DMB.

For example:

```
    LDR X0, [X1]                        // Must be seen by the memory system before
the
                                                // STR below.
    DMB ISHLD
    ADD X2, #1                         // May be executed before or after the memory
                                                //system sees LDR.
    STR X3, [X4]                       // Must be seen by the memory system after the
                                                // LDR above.
```

It also ensures that any explicit preceding data or unified cache maintenance operations have completed before any subsequent data accesses are executed.

For example:

```
    DC CSW, X5                 // Data clean by Set/way
    LDR x0, [X1]              // Effect of data cache clean might not be seen by
                                        // this instruction
    DMB ISH
    LDR X2, [X3]              // Effect of data cache clean are seen by this
                                        // instruction
```

Data Synchronization Barrier

Data Synchronization Barrier(DSB) enforces the same ordering as the Data Memory Barrier, but it also blocks execution of any further instructions, not just loads or stores, until synchronization is complete. This can be used to prevent execution of a SEV instruction, for instance, that would signal to other cores that an event occurred. It waits until all cache, TLB, and branch predictor maintenance operations that are issued by this processor have completed for the specified shareability domain.

For example:

```
    DC ISW, X5                 // operation must have completed before DSB can
                                        // complete STR
    STR X0, [X1]              // Access must have completed before DSB can complete
    DSB ISH
    ADD X2, X2, #3            // Cannot be executed until DSB completes
```

Using barriers

The DMB and DSB instructions take a parameter which specifies the types of access to which the barrier operates, before or after, and a shareability domain to which it applies.

The available options are listed in the following table.

<option>	Description	Ordered Accesses (before - after)	Shareability Domain
OSHLD	Operation that waits only for loads to complete, and only to the outer shareable domain	Load - Load, Load - Store	Outer Shareable

<option>	Description	Ordered Accesses (before - after)	Shareability Domain
OSHST	Operation that waits only for stores to complete, and only to the outer shareable domain.	Store - Store	Outer Shareable
OSH	Operation only to the outer shareable domain.	Any - Any	Outer Shareable
NSHLD	Operation that waits only for loads to complete and only out to the point of unification.	Load - Load, Load - Store	Non-shareable
NSHST	Operation that waits only for stores to complete and only out to the point of unification.	Store - Store	Non-shareable
NSH	Operation only out to the point of unification.	Any - Any	Non-shareable
ISHLD	Operation that waits only for loads to complete, and only to the Inner Shareable domain	Load -Load, Load - Store	Inner Shareable
ISHST	Operation that waits only for stores to complete, and only to the Inner Shareable domain.	Store - Store	Inner Shareable
ISH	Operation only to the Inner Shareable domain.	Any - Any	Inner Shareable
LD	Operation that waits only for loads to complete.	Load -Load, Load - Store	Full system
ST	Operation that waits only for stores to complete.	Store - Store	Full system
SY	Full system operation. This is the default and can be omitted.	Any - Any	Full system

The ordered access field specifies which classes of accesses the barrier operates on. There are three options.

Ordered Accesses (before - after)	Description
Load - Load/ Store	This means that the barrier requires all loads to complete before the barrier but does not require stores to complete. Both loads and stores that appear after the barrier in program order must wait for the barrier to complete.
Store - Store	This means that the barrier only affects store accesses and that loads can still be freely reordered around the barrier.
Any - Any	This means that both loads and stores must complete before the barrier. Both loads and stores that appear after the barrier in program order must wait for the barrier to complete.

A more subtle effect of the ordering rules is that the instruction interface, data interface, and MMU table walker of a core are considered as separate observers. This means that you might need, for example, to use DSB instructions to ensure that an access to one interface is guaranteed to be observable on a different interface.

If you execute a data cache clean and invalidate instruction, DC CVAU, X0 for example, you must insert a DSB instruction after this. You must add this to be sure that subsequent translation table walks, modifications to translation table entries, instruction fetches, or updates to instructions in memory, can all see the new values.

For example, consider an update of the translation tables:

```
STR X0, [X1]                        // update a translation table entry
DSB ISHST                                 // ensure write has completed
TLBI VAE1IS, X2                  // invalidate the TLB entry for the entry that
                                          // changes
DSB ISH                          // ensure that TLB invalidation is complete
```

```
    ISB                                         // synchronize context on this
processor
```

A DSB is required to ensure that the maintenance operations complete and an ISB is required to ensure that the effects of those operations are seen by the instructions that follow.

The processor might speculatively access an address that is marked as Normal at any time. So when considering whether barriers are required, consider more than just explicit accesses that are generated by load or store instructions.

One-way barriers

A64 adds new load and store instructions with implicit barrier semantics. The instructions are less restrictive than either DMB or DSB instructions. They also require that all loads and stores before or after the implicit barrier are observed in program order.

Instruction	Description
Load-Acquire (LDAR)	All loads and stores that are after an `LDAR` in program order, and that match the shareability domain of the target address, must be observed after the `LDAR`.
Store-Release (STLR)	All loads and stores preceding an `STLR` that match the shareability domain of the target address must be observed before the `STLR`.

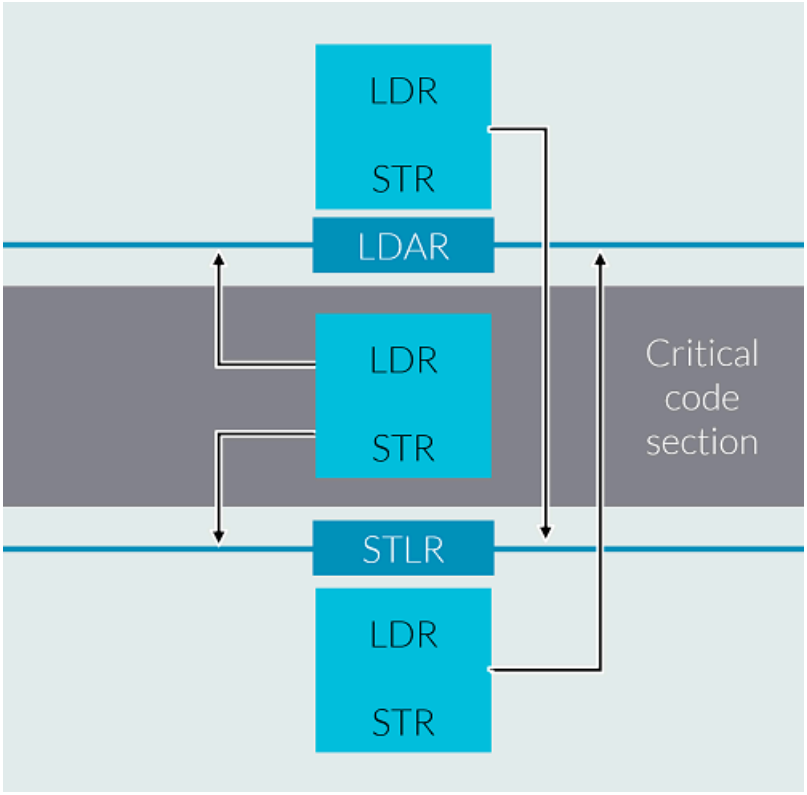
There are also exclusive versions of the above, `LDAXR` and `STLXR`, available.

Unlike the data barrier instructions, which take a qualifier to control which shareability domains see the effect of the barrier, the `LDAR` and `STLR` instructions use the attribute of the address accessed.

An LDAR instruction guarantees that any memory access instructions after the `LDAR`, are only visible after the load-acquire. A store-release guarantees that all earlier memory accesses are visible before the store-release becomes visible and that the store is visible to all parts of the system capable of storing cached data at the same time.

The following figure shows how accesses can cross a one-way barrier in one direction but not in the other:

Figure 5-1: One way barriers



Use of barriers in C code

The C11 and C++11 languages have a good platform-independent memory model that is preferable to intrinsics.

All versions of C and C++ have sequence points, but C11 and C++11 also provide memory models. Sequence points only prevent the compiler from reordering C++ source code. There is nothing to stop the processor reordering instructions in the generated object code, or for read and write buffers to reorder the sequence in which data transfers are sent to the cache. In other words, they are only relevant for single-threaded code. For multi-threaded code, then either use the memory model features of C11 / C++11, or other synchronization mechanisms such as mutexes which are provided by the operating system. Examples of sequence points in code include function calls and accesses to volatile variables.

The C language specification defines sequence points as follows:

At certain specified points in the execution sequence, called sequence points, all side effects of previous evaluations shall be complete, and no side effects of subsequent evaluations shall have taken place.

Barriers in Linux

The Linux kernel includes several platform-independent barrier functions. See the Linux kernel documentation in the memory-barriers.txt file for more details.